Debugging Applications that use TI-RTOS Technical Overview

Todd Mullanix

TI-RTOS Apps Manager

Texas Instruments

Agenda

- 30 Second Advertisement
- Stack Overflow
- Device Exception
- Memory Mismanagement
- Debugging Lab (separate PPT)

Pre-work: Please familiarize yourself with the following information prior to this training:
- CCS
- TI-RTOS

Texas Instruments

30 Second Advertisement

Since we know there a number of customers that will not want to use an RTOS for various reasons. Here's some key point to remember:

- **TI-RTOS is developed and supported by TI:** If you write your own little scheduler, you have to write it, maintain it, port it if you move to a new device, etc. Is your job to deliver a smaller scheduler or a real product on time?
- **Includes Power Management:** For the low power devices, TI has power management included in TI-RTOS. Look at the device's power management…it is hard. Do you really want to deliver a power manager (and power aware drivers) or a real product on time?
- **Portable:** Want to move to another device? Hope you factored this in when you wrote their own little scheduler and drivers.
- **Scalable**: Want to add system-level functionality into the application? Hope you factored this in when you wrote their own little scheduler and drivers.
- **Don't want to learn an RTOS**: TI-RTOS's kernel has "standard" OS components: Tasks, interrupts, semaphore, queues, etc. It also supports POSIX (also called Pthreads).
- **Overhead**: Yes, TI-RTOS takes space. So does your little scheduler. What is the threshold (other than "smaller")? For the smallest CC1310 device (32KB flash), TI-RTOS can be set-up to only use ~3KB (~6KB with full Power Management) of the flash and this still includes almost all the kernel's functionality. Note: CC13xx/CC26xx has the kernel's .text in ROM.
- **Debugging Facilities**: Hey this is a good lead-in…

Texas Instruments

Stacks Overflow

Stacks are used to place information like local data storage, return state, parameter passing, etc. Stacks grow as more subroutines are called. Finding a “good” value for the stack size is im… u make it too large, you waste memory. Worse though is if you make it too small…

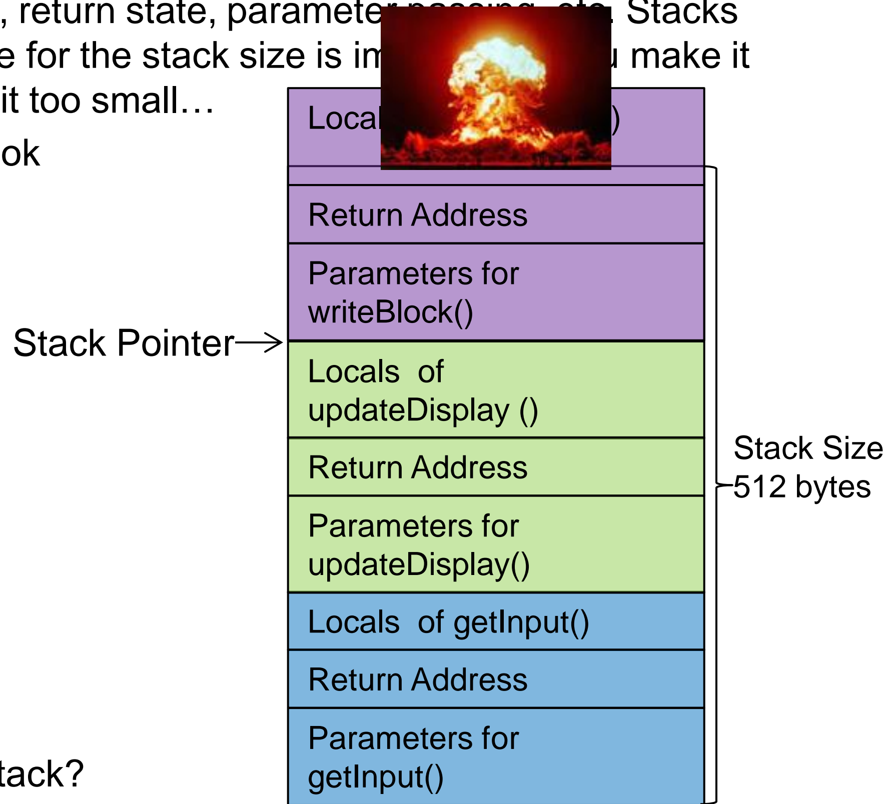
Here’s code executing and let’s see what the stack might look like before the calling `writeBlock()` in `updateDisplay()`.

```
void getInput(int foo, int bar)
{
  …
    retVal = updateDisplay(buffer, BASE_X, BASE_Y);

int updateDisplay(char *bitmap, int x, int y)
{
    …
    writeBlock(&bitmap[i], xoffset, yoffset);

int writeBlock(char *block int x, int y)
{
    char tempBuf[256];
```

What’s going to happen when tempBuf is placed onto the stack?



Texas Instruments

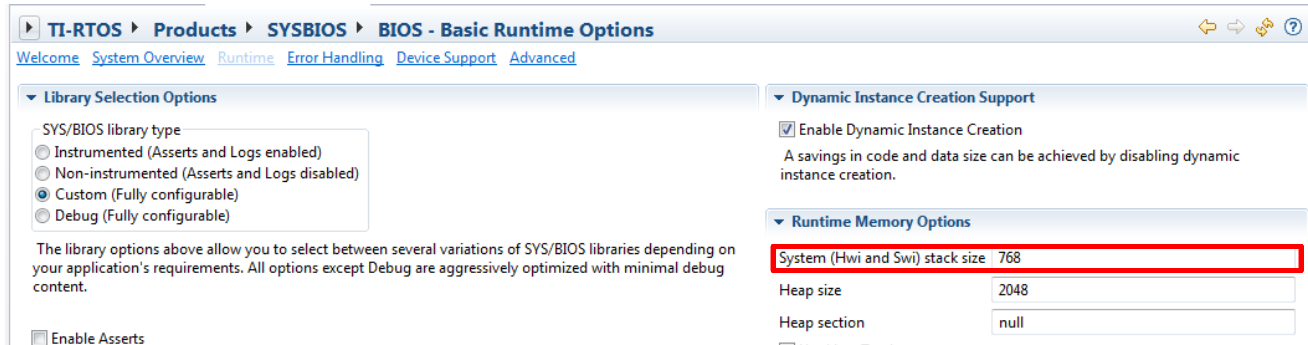
Stacks in TI-RTOS

With TI-RTOS there are two different types of stacks

System Stack: Hardware Interrupts (Hwi) and Software Interrupts (Swi) use a single system stack. The size of this stack is configurable via the .cfg file (with IAR, you set it in the linker file).

```
Program.stack = 768;
```

Or graphically



Task Stack: Each Task has it's own stack. The size of this stack is specified when you create a task.

Peak Usage of Stacks in TI-RTOS

The kernel will initialize the stacks with 0xBE values if the initStackFlags are set to true in the .cfg file (the default is true).

```
var Task = xdc.useModule('ti.sysbios.knl.Task');
var halHwi = xdc.useModule('ti.sysbios.hal.Hwi');
Task.initStackFlag = true;
halHwi.initStackFlag = true;
```

If you set these to false, you save ~140 bytes of code and booting is slightly faster.
If you use true, you can get the peak usage in RTOS Object Viewer (ROV) in CCS and IAR.

RTOS Object View (ROV)

HeapMem
Hwi
Idle
Queue

Basic | Detailed | Module | Exception | Raw

address	options	activeInterrupt	pendingInterrupt	exception	hwiStackPeak	hwiStackSize	hwiStackBase
0x2000160c	...	0	0	none	540	768	0x20001764

RTOS Object View (ROV)

Semaphore
Startup
Swi
System
Task
Timer (ti.sysbios.family.arm.m3)
Timer (ti.sysbios.family.arm.msp432)

Basic | Detailed | CallStacks | ReadyQs | Module | Raw

address	label	priority	mode	fxn	arg0	arg1	stackPeak	stackSize	stackBase	curCoreId
0x20001644	ti.sysbios.knl.Task.IdleTask	0	Running	ti_sysbios_knl_Idle_loop__E	0x0	0x0	124	512	0x200009c0	n/a
0x20000fc0	console	1	Blocked	consoleFxn	0x0	0x0	128	128	0x200003f0	n/a

Overrun!

Caveat: when opened, these ROV tabs slow down single stepping in the debugger.

Texas Instruments

Runtime Checking of Stacks in TI-RTOS

The kernel will perform runtime checks if desired.

```
var Task = xdc.useModule('ti.sysbios.knl.Task');
var halHwi = xdc.useModule('ti.sysbios.hal.Hwi');
Task.checkStackFlag = true;
halHwi.checkStackFlag = true;
```

If you set these to **false**, you save ~290 bytes of code.

If you use **true**:

- Whenever there a task context switch, the kernel will check the stack peaks of the new and old tasks to make sure it is still 0xBE. If it is not, an error* is raised.
- If the Idle task executes, it will call the "ti_sysbios_hal_Hwi_checkStack" function to make sure the system stack is ok. If the stack is blown, an error* is raised.

* Refer to the xdc.runtime.Error module for details on how to plug in an Error handler.

Texas Instruments

Stacks: Recommendations

For new development, it's recommended you enable both the initialization of the stack and the runtime checking.

```
Task.initStackFlag = true;
halHwi.initStackFlag = true;

Task.checkStackFlag = true;
halHwi.checkStackFlag = true;
```

Once you have the application to a stable point, you can then turn them off if you are tight on space or need to squeeze out a tiny bit more performance. If these are not a concern, you can leave them enabled and plug in an Error* handler that can act accordingly if the stacks are blown (e.g. dump memory to be analyzed later and restart the device).

* Refer to the xdc.runtime.Error module for details on how to plug in an Error handler.

Texas Instruments

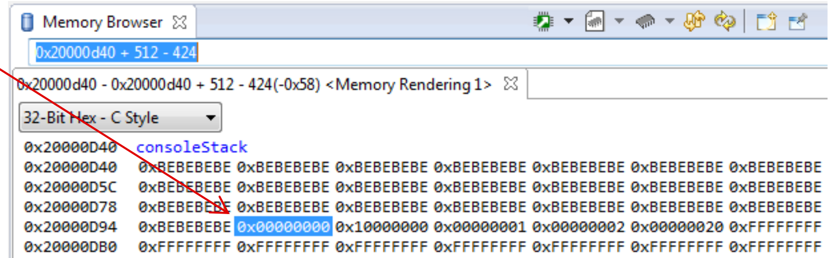
Additional Techniques to Size Stacks

Hardware Watchpoints: HW watchpoints in CCS are great for seeing what caused the stack peak. You can run the application and determine the stack peak with ROV.

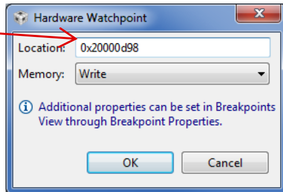
Basic | Detailed | CallStacks | ReadyQs | Module | Raw

address	label	priority	mode	fxn	arg0	arg1	stackPeak	stackSize	stackBase
0x200019c4	ti.sysbios.knl.Task.IdleTask	0	Running	ti_sysbios_knl_Idle_loop__E	0x0	0x0	336	1024	0x20000940
0x20001340	console	1	Blocked	consoleFxn	0x40	0x0	424	512	0x20000d40

If you look at the memory, you can see the peak



Then simply set a HW Watchpoint for a write to that address.



Restart the application. It will be hit quickly (since you have stack initialization turned on). The next time you hit the breakpoint, you can look at the call stack to see what caused the peak. Please note: the quality of the call task trace is dependent on the device, the symbols compiler options you have enabled/disabled, and compiler toolchain.

Texas Instruments

Additional Techniques to Size Stacks

Call_graph: Call_graph analyzes stacks based on a .out file (i.e. statically determined as opposed to runtime). This can be useful in trying to find places that use a large amount of stack space. Here is a write-up: http://processors.wiki.ti.com/index.php/Code_Generation_Tools_XML_Processing_Scripts

The call_graph tool does not work through function pointers and assembly that is not instrumented for it. For example, below shows UART_write being 8 bytes, where in reality it is more since it calls UARTMSP432_write via a function pointer.

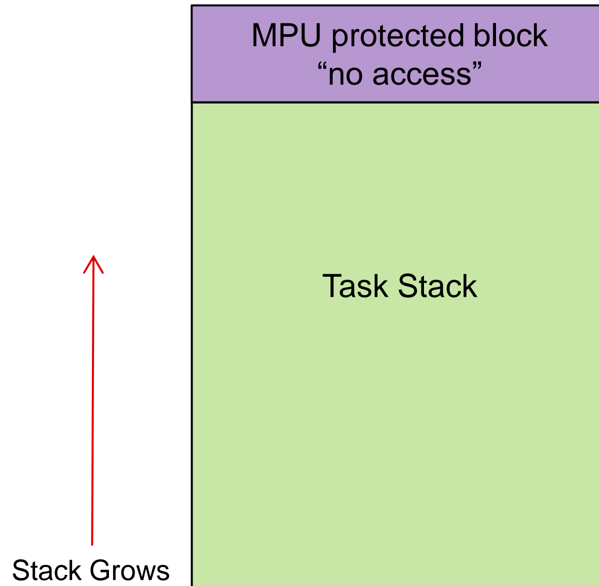
```
consoleFxn : wcs = 320
|  malloc : wcs = 120
|  |  <repeat ...>
|  free : wcs = 0
|  |  <repeat ...>
|  simpleConsole : wcs = 296
|  |  UARTUtils_getHandle : wcs = 8
|  |  UART_read : wcs = 8
|  |  UART_write : wcs = 8
|  |  ti_sysbios_heaps_HeapTrack_printTask__E : wcs = 256
|  |  |  ti_sysbios_heaps_HeapTrack_Object__first__S : wcs = 0
|  |  |  ti_sysbios_heaps_HeapTrack_Object__get__S : wcs = 0
|  |  |  ti_sysbios_heaps_HeapTrack_Object__next__S : wcs = 0
|  |  |  ti_sysbios_heaps_HeapTrack_printTrack__I : wcs = 216
|  |  |  |  ti_sysbios_knl_Task_Handle__label__S : wcs = 16
|  |  |  |  |  xdc_runtime_Core_assignLabel__I : wcs = 8
|  |  |  |  |  |  xdc_runtime_Text_cordText__E : wcs = 0
|  |  |  |  xdc_runtime_System_printf__E : wcs = 168
|  |  |  |  |  <repeat ...>
|  |  ti_sysbios_knl_Task_self__E : wcs = 0
|  |  ti_sysbios_knl_Task_sleep__E : wcs = 168
```

```
UARTMSP432_write : wcs = 208
|  HwiP_disable : wcs = 0
|  |  <repeat ...>
|  ( HwiP_restore
     ti_sysbios_family_arm_m3_Hwi_restoreFxn__E ) : wcs = 0
|  Power_setConstraint : wcs = 16
|  |  <repeat ...>
|  SemaphoreP_pend : wcs = 184
|  |  ti_sysbios_knl_Semaphore_pend__E : wcs = 176
|  |  |  ti_sysbios_knl_Clock_addI__E : wcs = 16
|  |  |  |  ti_sysbios_knl_Queue_put__E : wcs = 0
|  |  |  ti_sysbios_knl_Task_blockI__E : wcs = 32
|  |  |  ti_sysbios_knl_Task_restore__E : wcs = 80
|  |  |  |  <repeat ...>
|  UART_clearInterruptFlag : wcs = 0
|  UART_disableInterrupt : wcs = 8
|  UART_enableInterrupt : wcs = 0
```

Additional Techniques to Catch Stack Overflow

Memory Protection Unit (MPU) Module

There is a MPU module in the TI-RTOS kernel for selected ARM Cortex-A and Cortex-M devices. You can have a small region (e.g. 32 bytes) at the top of the stack where its attributes are no-access. If the stack grows into the protected region an exception occurs.



Texas Instruments

Exceptions

What is an exception?

Really short-story…not a good thing!

Short-story…a condition that the device cannot handle. For example, bus error, executing an unknown instruction, etc.

TI-RTOS supports exception handling for the ARM and C64+ devices. For this presentation, we are going to focus on the exception handling for the MCU (M3, M4, M4F) devices.

Texas Instruments

Exceptions

We are going to look at what happens when the following code is executed on the EK-TM4C1294XL board. Note: line 68 in the `heartBeatFxn()` is going to cause an exception!

```
63 Void heartBeatFxn(UArg arg0, UArg arg1)
64 {
65     while (1) {
66         Task_sleep((unsigned int)arg0);
67         GPIO_toggle(Board_LED0);
68         asm("    .word    0x4567f123   "); /* undefined instruction */
69     }
70 }
```

When an exception occurs, the device jumps to the exception handler. TI-RTOS allows different types of handlers for exceptions to be plugged in:

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

The next slides will show how to select which exception handler to use and its benefits.

Texas Instruments

Exceptions: User Supplied Handler

User Supplied: If you want to be master of your domain and supply the exception handler yourself, you can set the following and your handler is called (instead of going into the spin-loop).

```
var m3Hwi = xdc.useModule('ti.sysbios.family.arm.m3.Hwi');
m3Hwi.excHandlerFunc = "&myExceptionHandler";
```

Here is some pseudo-code for the user supplied handler

```
Void myExceptionHandler(UInt *excStack, UInt lr)
{
    // do stuff like write RAM to flash, flash LEDS, phone home, etc.
    // reset device
```

Benefits

- You're in charge.
- You still know you have an exception from ROV but no reliable decoding

RTOS Object View (ROV)

empty_EK_TM4C1294XL_TI_TivaTM4C1294NCPDT.out
- Viewable Modules
  - BIOS
  - Boot
  - Clock
  - Diags
  - Event
  - GPIO
  - GateHwi
  - GateMutex
  - HeapMem
  - Hwi

Basic | Detailed | Module | Exception | Raw

address	options	activeInterrupt	pendingInterrupt	exception	hwiStackPeak	hwiStackSize	hwiStackBase
0x2000...	...	3	35	Yes	456	768	0x200038e0

Texas Instruments

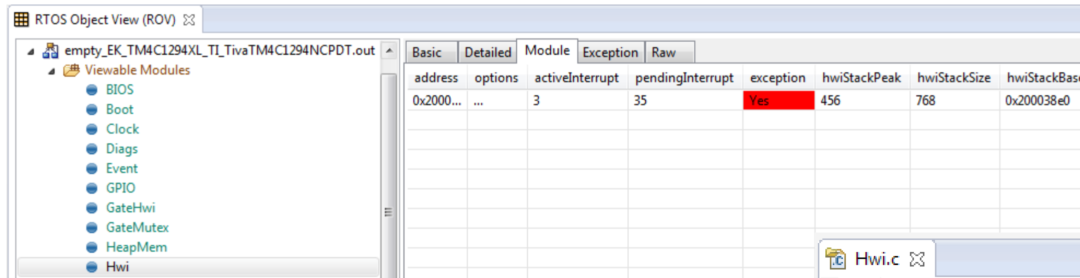
Exceptions: TI-RTOS Spin Loop Handler

TI-RTOS Spin Loop Handler: You can configure TI-RTOS to use a spin-loop handler instead

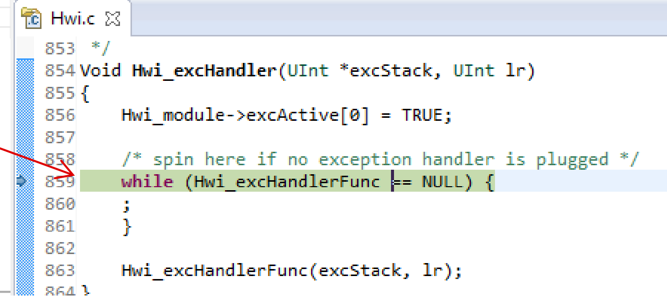
```
var m3Hwi = xdc.useModule('ti.sysbios.family.arm.m3.Hwi');
m3Hwi.excHandlerFunc = null;
```

Benefits

- Smallest footprint for the handlers
- You still know you have an exception from ROV but no reliable decoding.



- If you halt the target, you will be in the spin-loop



Texas Instruments

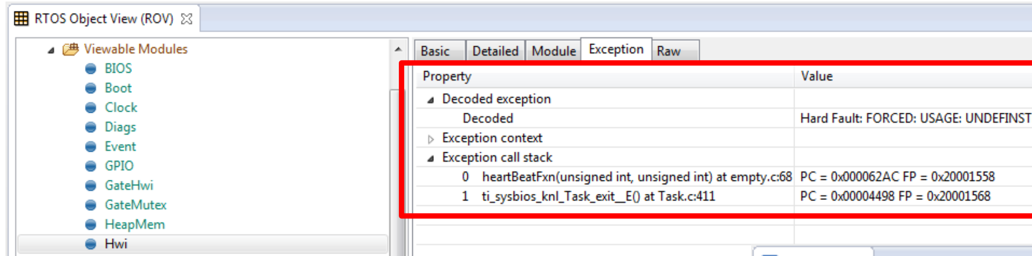
Exceptions: TI-RTOS Minimal Exception Decoding Handler

TI-RTOS Minimal Exception Decoding Handler: If you disable the enhanced exception handling and use the TI-RTOS minimal handler instead.

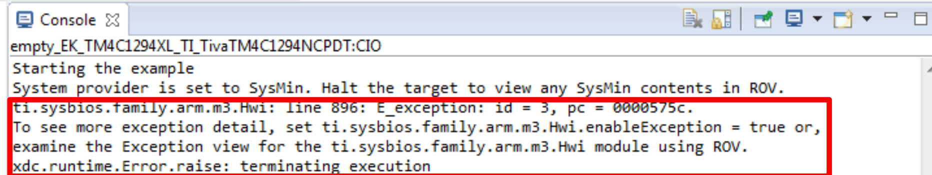
```
var m3Hwi = xdc.useModule('ti.sysbios.family.arm.m3.Hwi');
m3Hwi.enableException = false;  //true for enhanced
```

Benefits

- ROV decodes the exception and give a back trace. Note the “heartBeatFxn” name, file name and line number!



- The CCS Console will have some information (if application is configured to output to CCS Console).
- You can set excHookFunc to execute before decoding.



- However, slightly larger footprint when compared to the spin-loop (~400 bytes more).

Please note, the quality of the back trace is dependent on the device, the symbols compiler options you have enabled/disabled, and compiler toolchain.

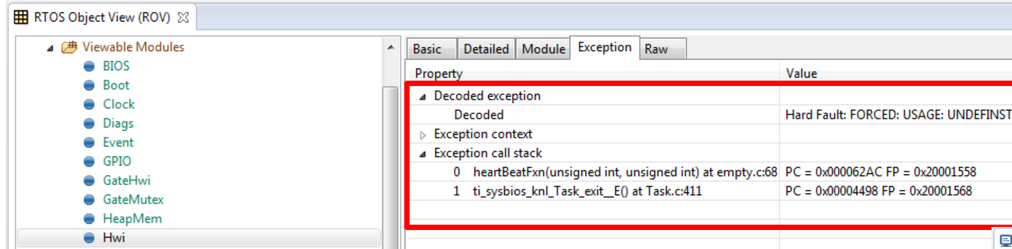
Exceptions: TI-RTOS Enhanced Exception Decoding Handler

TI-RTOS Enhanced Exception Decoding Handler: If you accept the default configuration (shown below), you get the TI-RTOS enhanced exception decoder.

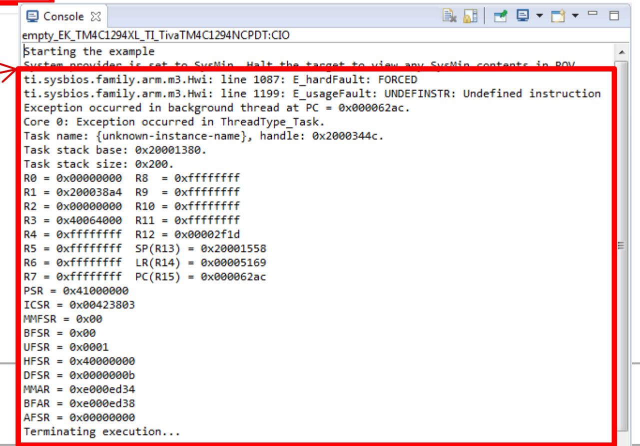
```
var m3Hwi = xdc.useModule('ti.sysbios.family.arm.m3.Hwi');
m3Hwi.enableException = true;
```

Benefits

- ROV decodes the exception and gives a back trace.



- The CCS Console will have the decoded exception (if application is configured to output to CCS Console).
- You can set excHookFunc to execute before decoding. (refer to the Additional Details slide at the end for more details)
- However, ~3K larger footprint when compared to the "minimal"



Exceptions: Handlers Summary

You have several options with TI-RTOS for handling exceptions

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

Handler	ROV	CCS Console	excHookFunc
User Supplied	Exception is flagged	Up to user code	Not Available
Spin-loop	Exception is flagged	Nothing	Not Available
Minimal Decoder	Decoded & Back Trace	Notification	Available
Enhanced Decoder	Decoded & Back Trace	Decoded & Registers	Available

Texas Instruments

More Exception Information…

excHookFunc: For the enhanced and minimal TI-RTOS decoding exception handlers, you can plug in a function that will be called during the handling of the exception. This gives you an opportunity to perform any needed actions. Refer to the ti.sysbios.family.arm.M3.Hwi module for more details.

More Exception Details: There is more information about exceptions here: http://processors.wiki.ti.com/index.php/SYS/BIOS_FAQs#4_Exception_Dump_Decoding_Using_the_CCS_Register_View

Texas Instruments

Memory Allocation

Doing dynamic memory allocation in an embedded device has its risks. TI-RTOS offers a way to easily add a smart heap on top of the system/default heap. This heap is called **HeapTrack**. It helps with the following areas

- **Over-writing the end of allocated buffers**
- **Freeing the same block twice**
- **Memory leaks**
- **Sizing the heap**

To enable **HeapTrack**, simply set the following in the .cfg file:

```
BIOS.heapTrackEnabled = true;
```

Or graphically

usbmousedevice.cfg

TI-RTOS ▸ Products ▸ SYSBIOS ▸ BIOS - Basic Runtime Options

Welcome System Overview Runtime Error Handling Device Support Advanced

Library Selection Options

SYS/BIOS library type
- Instrumented (Asserts and Logs enabled)
- Non-instrumented (Asserts and Logs disabled)
- Custom (Fully configurable)
- Debug (Fully configurable)

The library options above allow you to select between several variations of SYS/BIOS libraries depending on your application's requirements. All options except Debug are aggressively optimized with minimal debug content.

Enable Asserts
Enable Logs
Custom Compiler Options r_speed=2 --program_level_compile -o3 -g --optimize_with_debug

Dynamic Instance Creation Support

Enable Dynamic Instance Creation
A savings in code and data size can be achieved by disabling dynamic instance creation.

Runtime Memory Options

System (Hwi and Swi) stack size 768
Heap size 1024
Heap section null
Use HeapTrack
The heap configured above is used for the standard C malloc() and free() functions or when the 'heap' argument to Memory_alloc() is NULL.

Memory Allocation: HeapTrack Details

For every memory allocation from the system heap, HeapTrack adds this small structure at the end of the allocated block.

```
struct Tracker {
    UInt32 scribble;    // = 0xa5a5a5a5 when in use
    Queue_Elem queElem; // next and prev pointers
    SizeT size;
    UInt32 tick;
    Task_Handle taskHandle;
}
```

Note: this may require you to slightly increase the size of your system heap since a little extra memory is used for every allocated block.

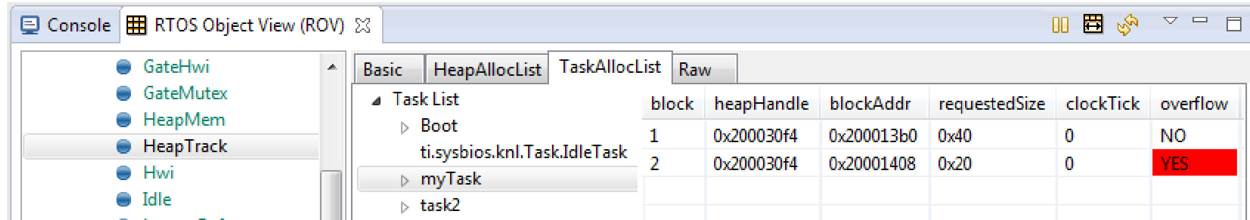
This structure is analyzed both during via ROV and runtime execution…

Texas Instruments

Memory Allocation: HeapTrack ROV

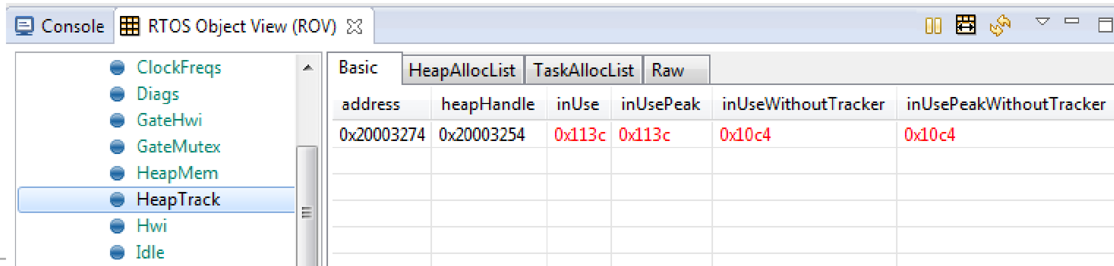
HeapTrack in ROV displays all the allocated blocks by the task that allocated the blocks and by the heap. Here are the things that HeapTrack in ROV helps find

Writing past the block: If the block has a corrupted scribble word, it is denoted with red. Note: the runtime check only happens when freeing the block. ROV shows it when it is still allocated.



Memory Leak: By looking at the timestamp and Task owner, you generally can spot memory leaks pretty easily.

Peaks: You can see the high-watermark for the heap also (both with and without the Tracker struct).



Texas Instruments

Memory Allocation: HeapTrack Runtime

When the allocated block is freed, the following two checks are done if kernel asserts are enabled.

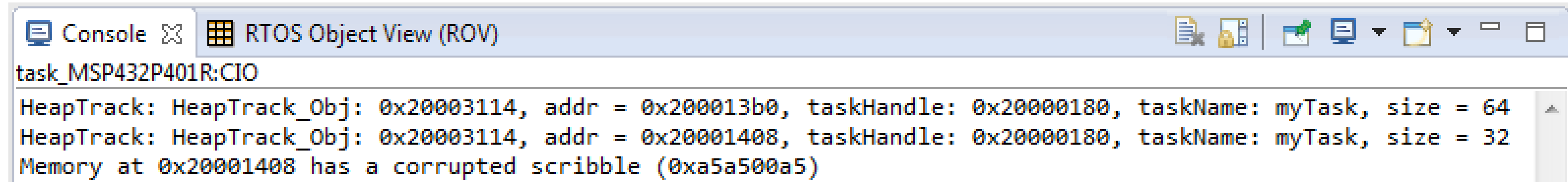
- **Double free**: In free, an assert checks to see that you are not trying to free a free block.
- **Writing past the block**: In the free, an assert check makes sure the scribble word is valid. If you accidently write past the end of the block, the scribble gets corrupted.

HeapTrack has a **two APIs that can be called by the applicatio**n to output (via System_printf) the allocated blocks.

```
Void HeapTrack_printHeap(HeapTrack_Object *obj);

Void HeapTrack_printTask(Task_Handle task);
```

Here is an example of the HeapTrack_printTask output. The task has allocated two blocks of size 64 and 32. The application has overwritten the scribble word (on purpose☺). This is shown in the output.



Memory Allocation: Recommendations

You can quickly enable **HeapTrack** and run your application. Then using ROV and/or runtime checks you can quickly find

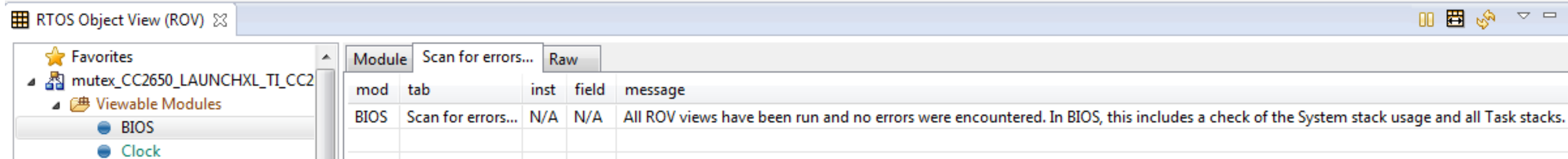
- **Over-writing the end of allocated buffers**
- **Freeing the same block twice**
- **Memory leaks**
- **Sizing the heap properly**

After the problem is fixed, simply turn **HeapTrack** off to minimize the slight performance and size impact.

Recommendation Summary

So…something weird is going on with your application. Here are some easy steps to do…

1. **Check System and Task stack peaks in ROV or "Scan for Errors…":** A quick and easy way to see if there are any issues detected is select "BIOS->Scan for errors…" in ROV. Stack overflows will show up here as well as Hwi and Task.



2. **Turn on TI-RTOS "Minimal" or "Enhanced" Exception Handling**.
3. **Enable HeapTrack if you have a dynamic allocation.**

Resources

- www.ti.com Web Page:
  - www.ti.com/tool/ti-rtos
- e2e Forum - TI-RTOS Forum:
  - http://e2e.ti.com/support/embedded/tirtos/default.aspx
- Additional Training & Support Resources
  - Main Product Page: http://processors.wiki.ti.com/index.php/TI-RTOS
  - TI-RTOS online training: https://training.ti.com/ti-rtos-workshop-series
  - Support direct link (includes Apps projects, extended release notes, FAQ, training, etc.) http://processors.wiki.ti.com/index.php/TI-RTOS_Support
- Download page:
  - http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/tirtos/index.html

Texas Instruments

Thank you