Intro to the TI-RTOS Kernel Workshop

Day 1

1. Welcome
2. Intro to CCS
3. Intro to TI-RTOS Kernel
4. Configuring the Kernel

Day 2

5. Using Hwi
6. Using Swi
7. Using Clk
8. Using Tasks
9. Inter-thread Comm

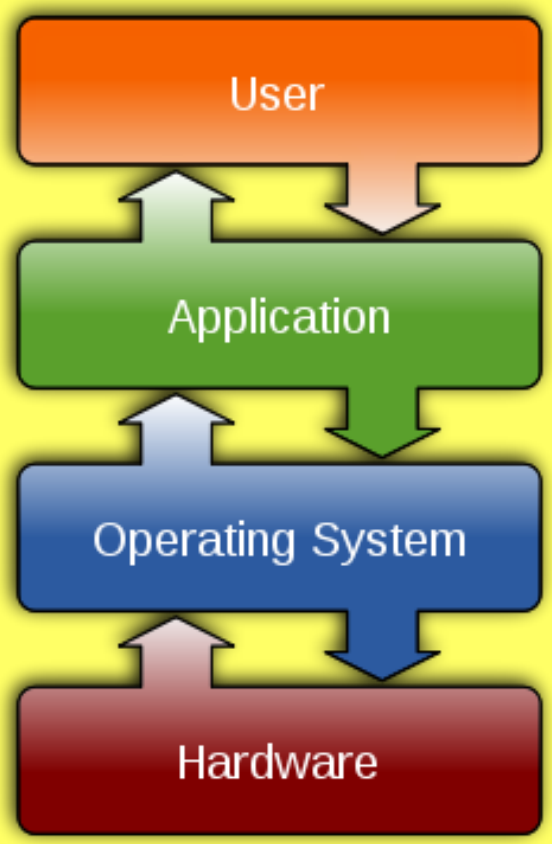
GrabBag Using Dynamic Mem

Objectives

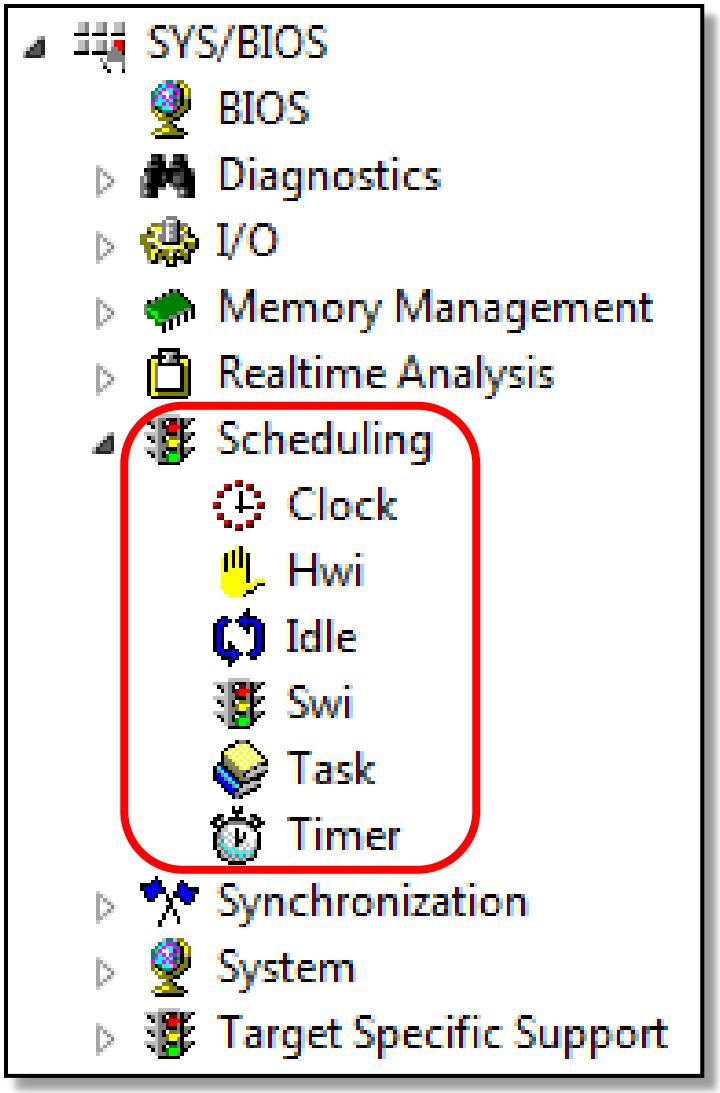
- Answer the question – "*why use an RTOS*?"
- Explore the basics concepts of the *TI-RTOS real-time kernel (SYS/BIOS)*
- Define kernel *thread types* and how the scheduler prioritizes threads in a system
- Explain the types of *debugging tools* built into the TI-RTOS kernel
- *Quiz* – Schedule the threads in a motor-control application

Outline

- What is the TI-RTOS Kernel?
- Scheduler
- Adding Tasks…
- BIOS Environment
- BIOS Debugging Tools
- For More Info…
- Chapter Quiz



TI-RTOS Kernel Services



TI-RTOS Kernel (or SYS/BIOS) is a ***library of services*** that users can add to their system to perform various tasks:

- Memory Mgmt (stack, heap, cache)
- Real-time Analysis (logs, graphs, loads)
- Scheduling (various thread types)
- Synchronization (e.g. semaphores, events)

What does the DNA of this kernel look like?

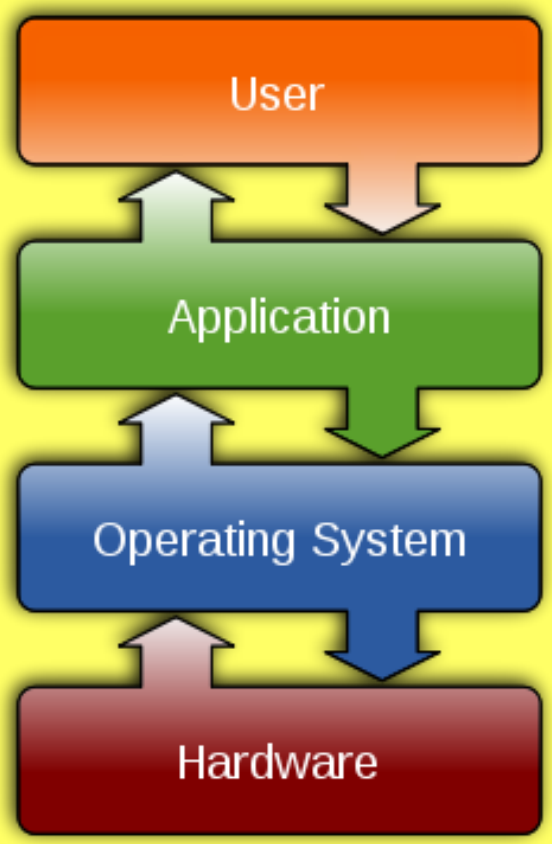
TI-RTOS Kernel – Characteristics

- RTOS means "Real-time O/S" – so the intent of this O/S is to provide common services to the user WITHOUT disturbing the real-time nature of the system
- The TI-RTOS Kernel (SYS/BIOS) is a PRE-EMPTIVE scheduler. This means the highest priority thread ALWAYS RUNS FIRST. Time-slicing is not inherently supported *(can change PRI dynamically if desired)*.
- The kernel is EVENT-DRIVEN. Any kernel-configured interrupts or user calls to APIs such as `Swi_post()` will invoke the scheduler. The kernel is NOT time-sliced although threads can be triggered on a time bases if so desired.
- The kernel is OBJECT BASED. All APIs (methods) operate on self-contained objects. Therefore when you change ONE object, all other objects are unaffected.
- Being object-based allows most RTOS kernel calls to be DETERMINISTIC. The scheduler works by updating event queues such that all context switches take the same number of cycles.
- Real-time Analysis APIs (such as Logs) are small and fast – the intent is to LEAVE them in the program – even for production code – aides field testing

Let's take a closer look at one of the most useful parts of the kernel - the SCHEDULER...

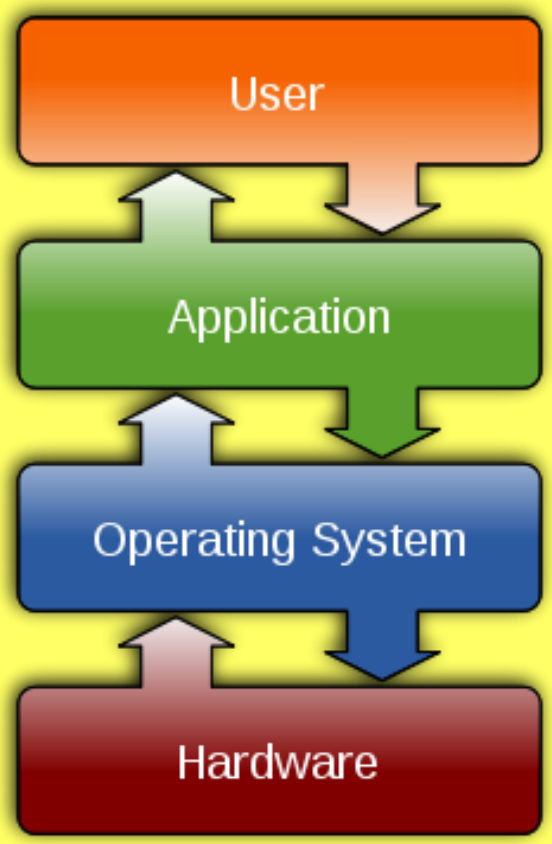
Outline

- What is the TI-RTOS Kernel?
- Scheduler
  - Problem
  - Solutions
- Adding Tasks…
- BIOS Environment
- BIOS Debugging Tools
- For More Info…
- Chapter Quiz

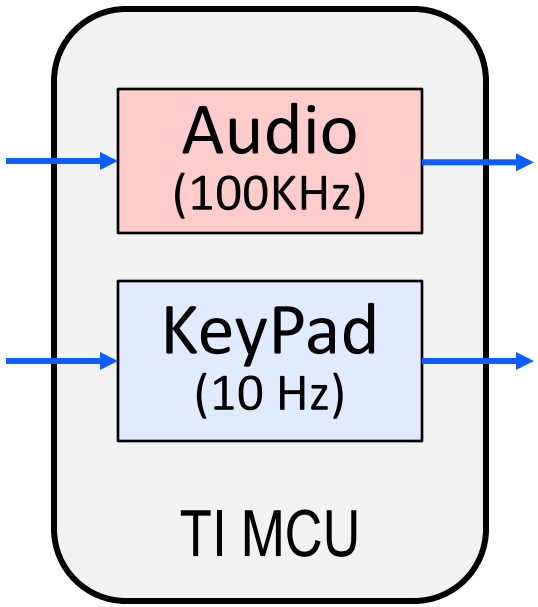


Outline

- What is the TI-RTOS Kernel?
- Scheduler
  - Problem
  - Solutions
- Adding Tasks…
- BIOS Environment
- BIOS Debugging Tools
- For More Info…
- Chapter Quiz



Scheduling Problem – Two Threads



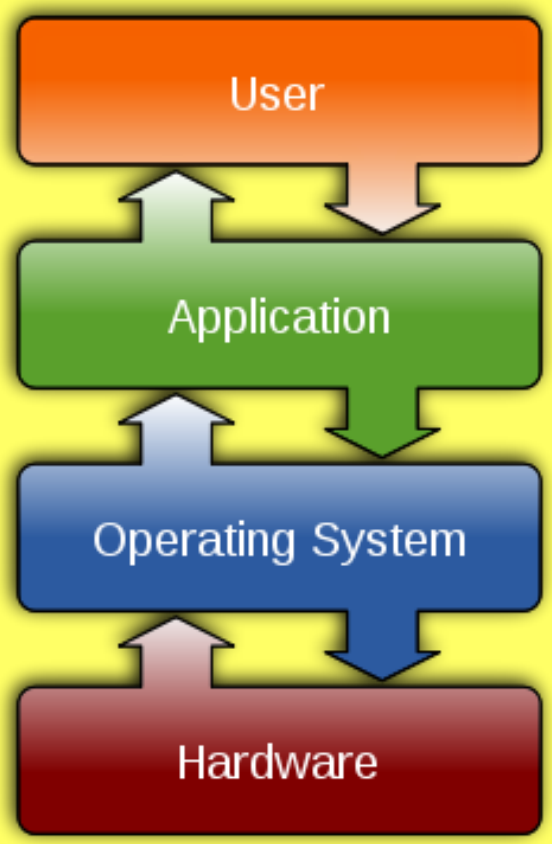
Problem Definition: you have two different threads that need to be serviced independently

- Will one routine conflict with the other?
- How do you SCHEDULE each thread?
- Is one "thread" higher PRIORITY than the other?

Let's explore a few options we can use to SCHEDULE these two threads…

Outline

- What is the TI-RTOS Kernel?
- Scheduler
  - Problem
  - Solutions
- Adding Tasks…
- BIOS Environment
- BIOS Debugging Tools
- For More Info…
- Chapter Quiz



Solution #1 – Super Loop

```
main()
{
   while(1)
   {
      Audio

      KeyPad

   }
}
```

Solution #1 – put each algo into an endless loop in main()

- What if algos run at different rates?
  - Audio – 100kHz (10µs)
  - KeyPad – 10Hz (100ms)
- What if one starves the other or delays the response causing jitter/noise?

Could you use a TIMER to trigger each "thread" (or process) via an interrupt?

Solution #2 – Timer-based INTs

```
TimerA_ISR()
{
 read sample;
```

Audio

```
}
```

```
TimerB_ISR()
{
 read keypad;
```

KeyPad

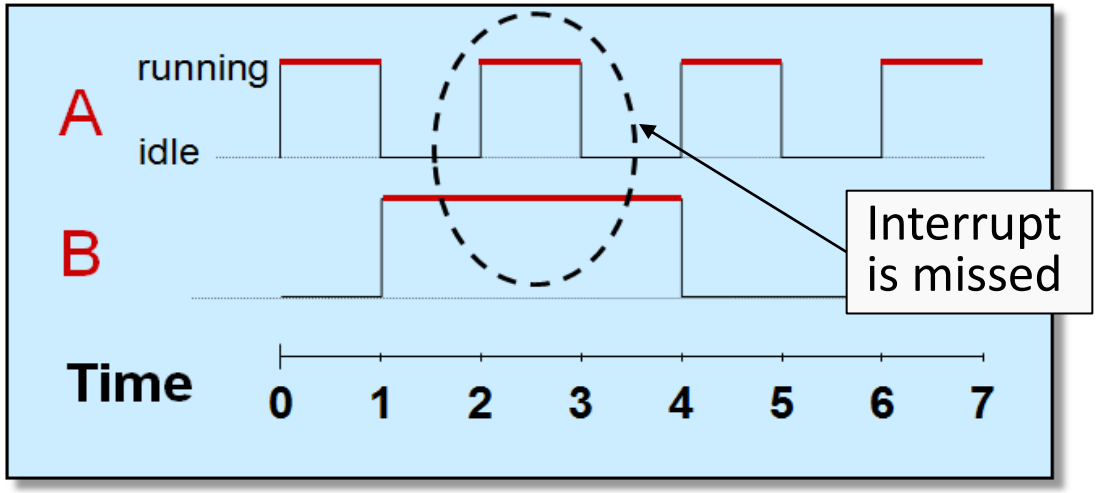
```
}
```

```
main()
{
  while(1);
}
```

Solution #2 – an *interrupt driven system* places each function in its own ISR

	Period	Compute	Usage
Audio	10µS	5µS	50%
Keypad	100ms	1ms	1%
			51%

- While CPU usage is fine, one interrupt may block the other (instantaneous):



How could we prevent this?

Solution #3 – Nested INTs

```
TimerA_ISR()
{
 read sample;
```

Audio

```
}
```

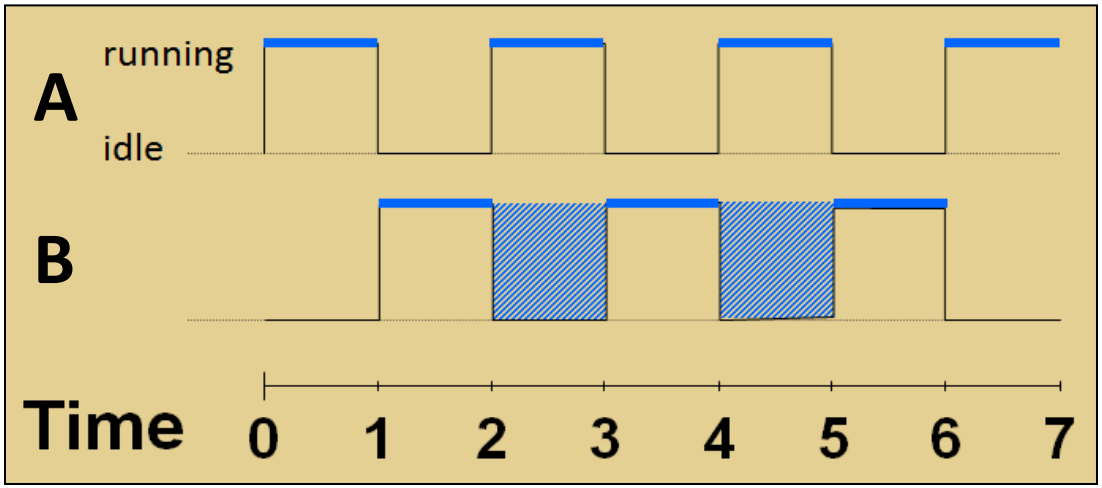
```
TimerB_ISR()
{
 read keypad;
```

KeyPad

```
}
```

```
main()
{
  while(1);
}
```

Solution #3 – *nested interrupts* allow hardware interrupts to preempt each other



- Number of priorities are tied to the number of interrupts (one fxn/ISR), h/w priorities inflexible
- Lower priority ISRs must enable higher priorities via manual code (touch one, touch all) – very messy and hard to validate

Why is nesting required in this system?

Solution #4 ? – Separate Process & ISR

```
TimerA_ISR()
{
 read sample;
```

Audio

```
}
```

```
TimerB_ISR()
{
 read keypad;
```

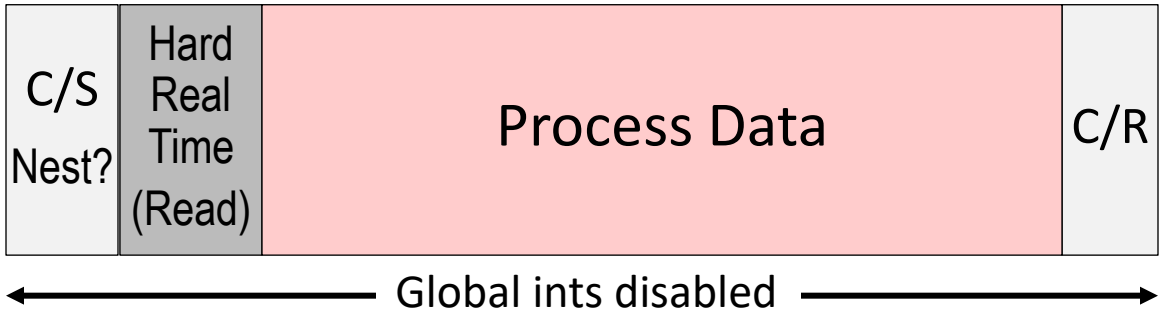
KeyPad

```
}
```

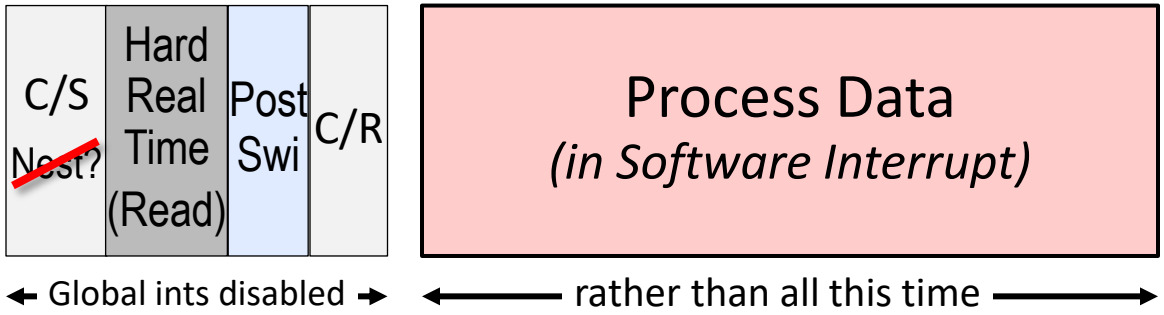
```
main()
{
  while(1);
}
```

Problem – *nested interrupts* are used because "Process" (algo) is done IN the ISR !

- When HI PRI is running, you could STILL miss interrupts:



- What if we could *separate* the PROCESS from the rest of the ISR?



This is what the BIOS Scheduler is all about…

Solution #4 – BIOS Scheduler

```
Audio_ISR()
{
 read sample;
 post_Audio;
}
```

```
Keypad_ISR()
{
 read keypad;
 post_KeyPad;
}
```

```
main()
{
  init_stuff;
  while(1);
  BIOS_start();
}
```

Hwi – Hardware INT

- *Context save/restore (done by BIOS)*
- *Hard real-time "read"*
- *Post Swi for follow-up*

Swi – Software INT

- *Posted by software*
- *PROCESS data*
- *User can select priority*

Idle – Background

- *Runs multiple fxns inside of a while(1) loop*

HI

LO

If we can DECOUPLE the processing from the TRIGGER (Hwi):

- ISRs become VERY short (no need for nesting)
- Configure PRIs of threads via software
- Add as many threads as we need (no limit)
- Touch ONE, no CODE changes to the others !

What is the difference between a THREAD and a FUNCTION ?

"Thread" vs. Function

Thread wrapper (C/S)

```
void my_fxn()
{
  int m, x, b;
  int y;

  y = m*x + b;
  serial = y;
  results += 1;
}
```

Thread wrapper (C/R)

A ***function*** is a set of program instructions that produce a given result.

- If you look at this function, can you tell what priority it is running at?

A ***thread*** is a function that runs within a specific context (PRIORITY, stack, etc.)

- `my_fxn()` could run as ANY thread type *(e.g. Swi_1 or Hwi_1)*
- User selects thread GROUP (and PRI w/in group), BIOS executes it

Hwi
- *High Priority ISR*
- *Calls my_fxn()*

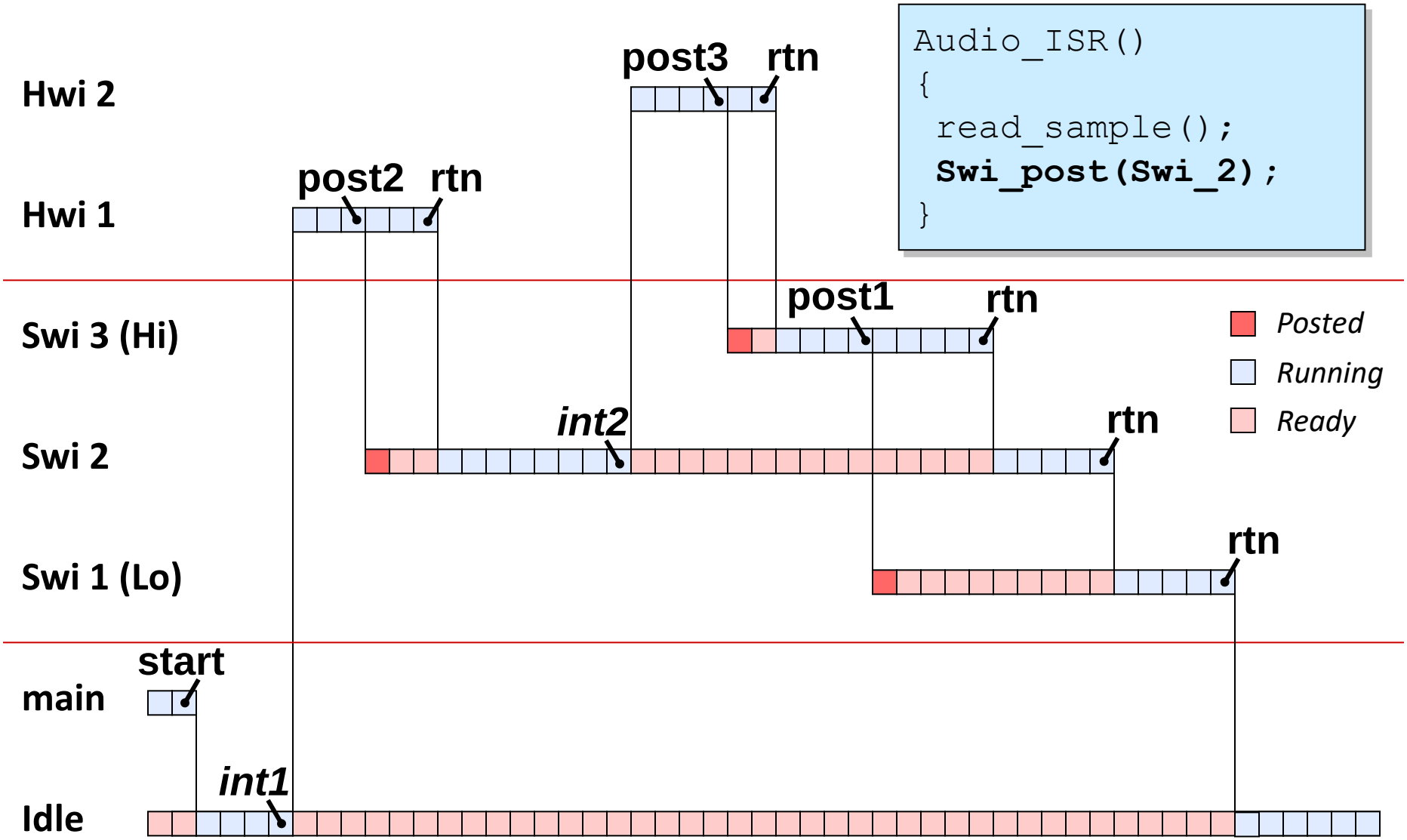
Swi
- *Follow-up to Hwi*
- *Calls my_fxn()*

Idle
- *Lowest PRI*
- *Calls my_fxn()*

← Priority ←

Let's see a scheduling example of these 3 threads…

BIOS – Priority Based Scheduling

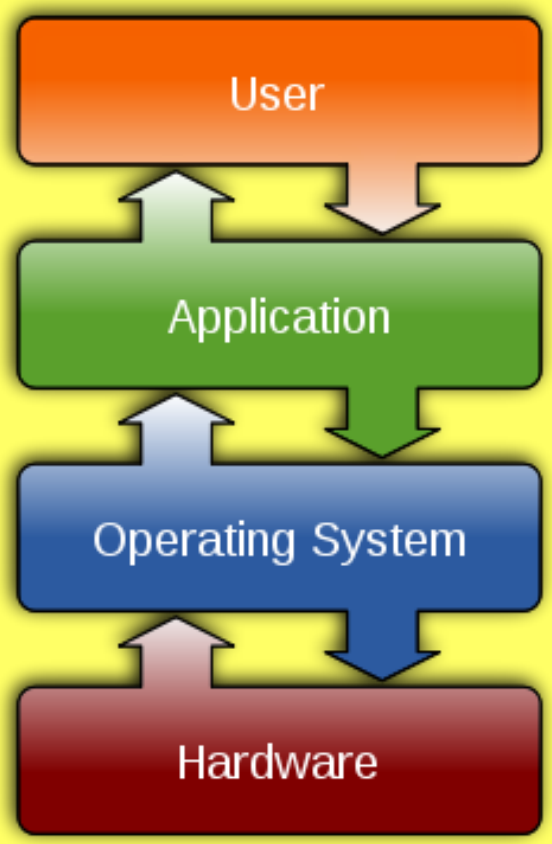



```
Audio_ISR()
{
 read_sample();
 Swi_post(Swi_2);
}
```

User SETs the priorities, BIOS executes them

Outline

- What is the TI-RTOS Kernel?
- Scheduler
- Adding Tasks…
- BIOS Environment
- BIOS Debugging Tools
- For More Info…
- Chapter Quiz



SYS/BIOS Thread Types (including Tasks)

Priority

Hwi

Hardware Interrupts

- Hardware event triggers Hwi to run
- BIOS handles context save/restore, nesting
- Hwi triggers follow-up processing
- Priorities set in silicon

Swi

Software Interrupts

- Software *posts* Swi to run
- Performs Hwi '*follow-up*' activity (process data)
- Up to 32 priority levels (16 on C28x/MSP430)

Task

Tasks

- Usually enabled to run by posting a '*semaphore*' (a task signaling mechanism)
- Designed to run concurrently – pauses when waiting for data (semaphore)
- Up to 32 priority levels (16 on C28x/MSP430)

Idle

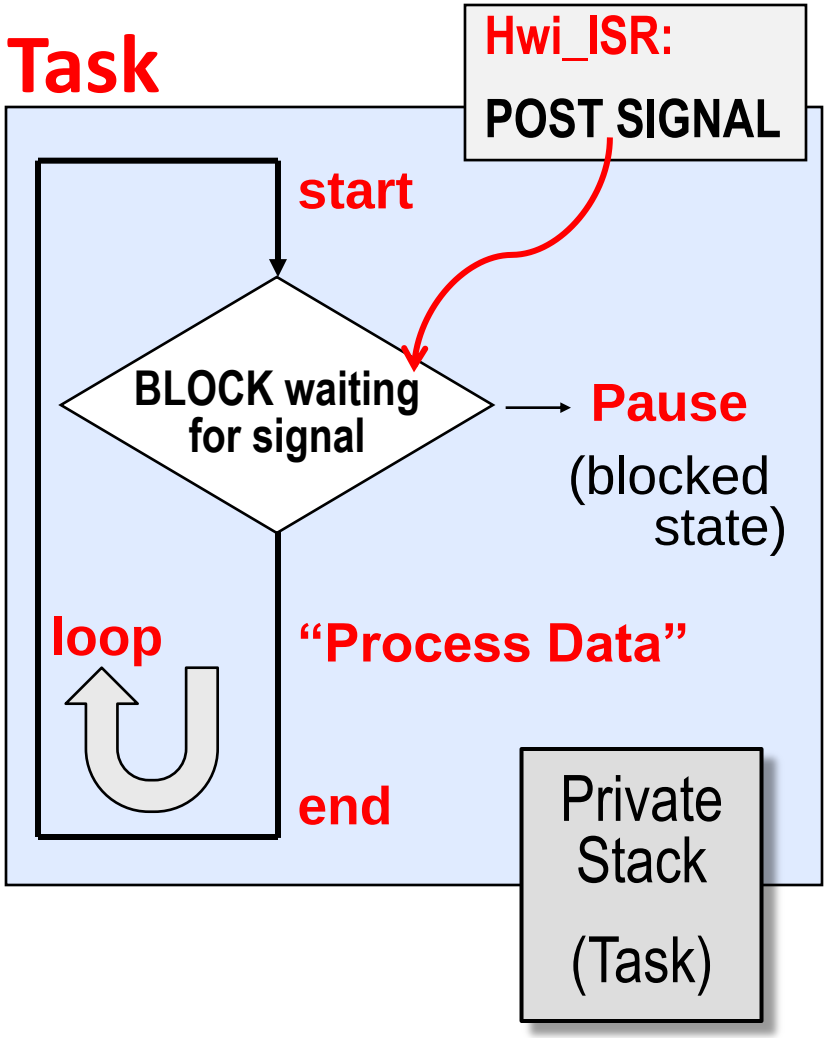
Background

- Runs as an infinite while(1) loop
- Users can assign multiple functions to Idle
- Single priority level

A little more info about Tasks...

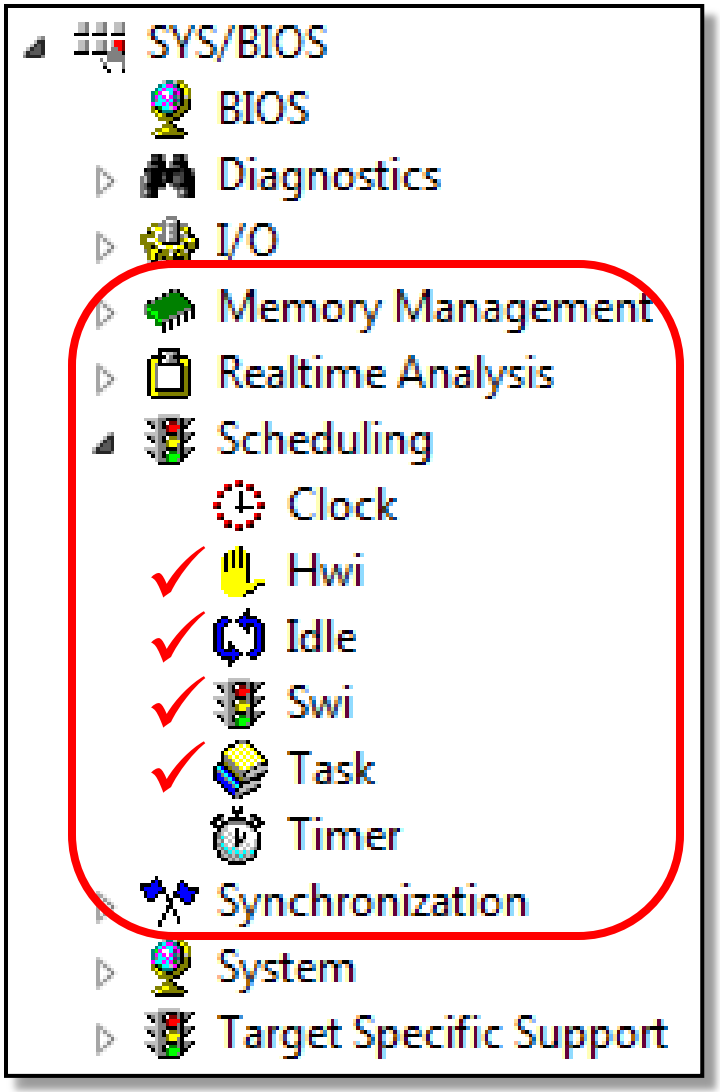
How Tasks Work…

- Tasks are just a function that run in a `while(1)` loop and contain a blocking call that waits for a signal
- When the task is BLOCKED (e.g. waiting for "*buffer ready*" signal), lower priority threads can run
- Another thread (e.g. Hwi) posts a signal to UNBLOCK the Task which triggers the execution of "Process Data"
- The code then loops back and BLOCKS again – waiting for signal
- Blocking requires each Task to have its own private stack.



Note: we have entire chapters dedicated to exploring both Swi's and Tasks

TI-RTOS Kernel Services – Summary

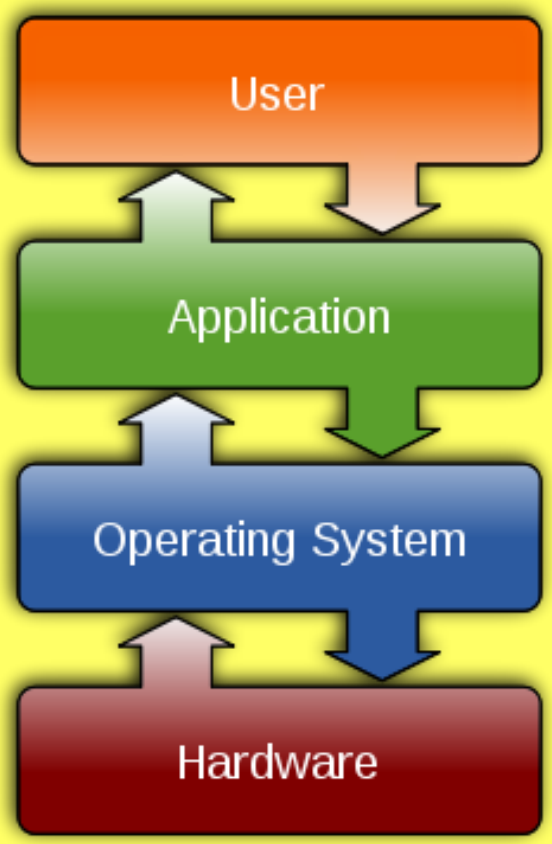


- You have now been exposed to the following thread types:
  - Idle (Chapter 4)
  - Hwi (Chapter 5)
  - Swi (Chapter 6)
  - Task (Chapter 8)
- Each of these thread types have their own chapter along with:
  - Memory Mgmt (stack, heap) – "Dyn Mem" chapter (optional)
  - Real-Time Analysis (logs, graphs, loads) – (Chapter 4)
  - *Clock/Timer* – (Chapter 7)
  - Synchronization (semaphores, events) – (Chapter 8)

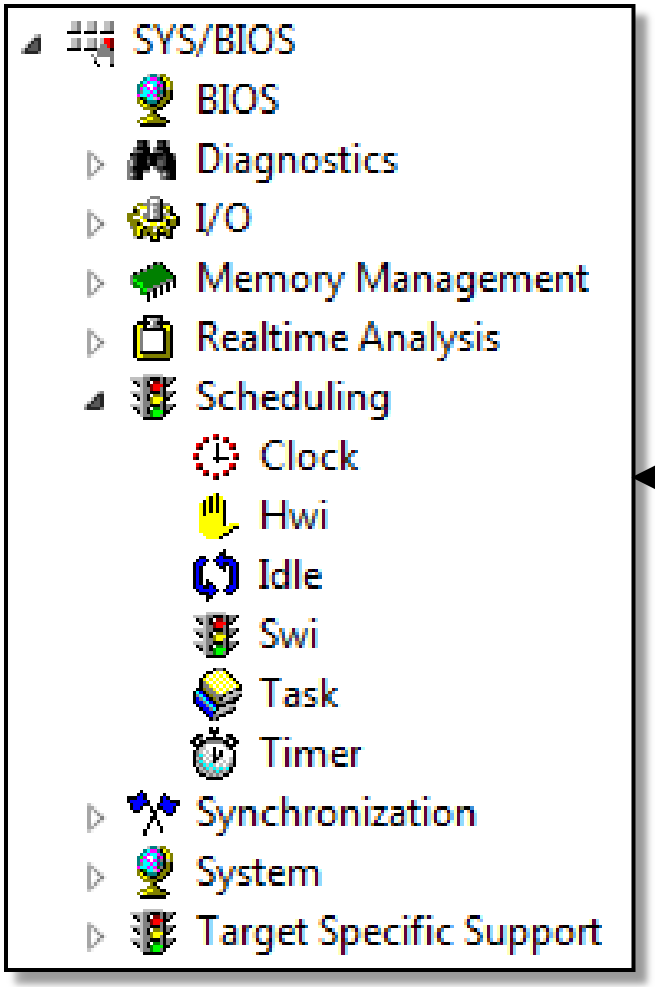
Let's take a look at the BIOS environment...

Outline

- What is the TI-RTOS Kernel?
- Scheduler
- Adding Tasks…
- **BIOS Environment**
- BIOS Debugging Tools
- For More Info…
- Chapter Quiz



Kernel APIs, Objects and Handles



BIOS is ***object oriented*** – each module is created as an *object* and objects are accessed via the API:

API

```
Audio_ISR() {
  Hard real_time access;
  Swi_post (AudioSwi);
}
```

Swi_Handle

```
typedef struct SwiObj {
   AudioSwi;    // handle
   Audio_fxn;   // fxn
   2;           // Priority
};
```

- Better encapsulation
- Change one w/o affecting others
- Multiple instances
- Easy to maintain, validate

How do you create a Swi object ?

Thread (Object) Creation in TI-RTOS

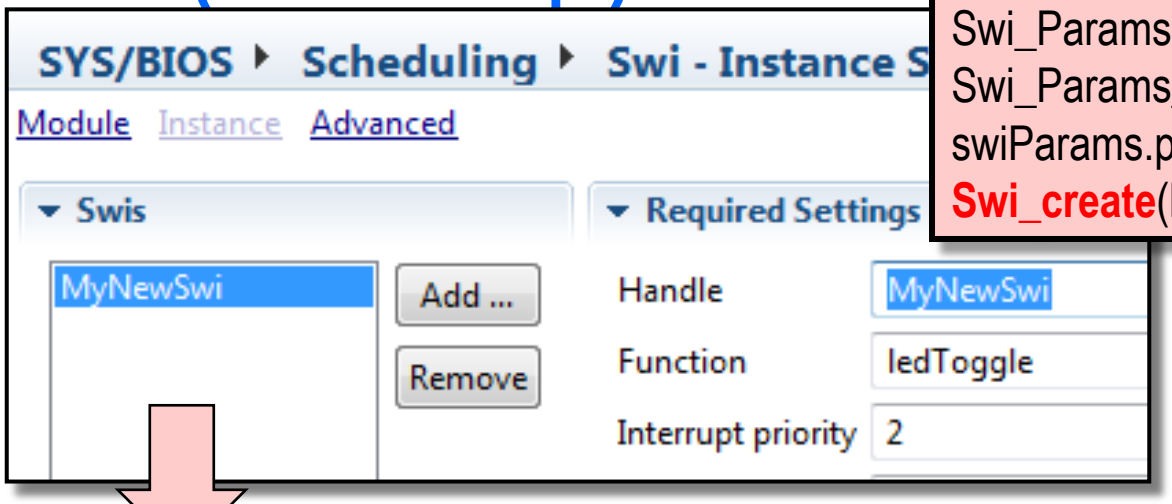
- **Users can create threads (BIOS resources or "objects"):**
  - Statically (via the GUI or .cfg script)
  - Dynamically (via C code) – *more details in the "memory mgmt" chapter*
  - BIOS doesn't care – but you might…

Dynamic (C Code)

```
#include <ti/sysbios/knl/Swi.h>
Swi_Params swiParams;
Swi_Params_init(&swiParams);
swiParams.priority = 2;
Swi_create(ledToggle, &swiParams, NULL);
```

app.c

Static (GUI or Script)



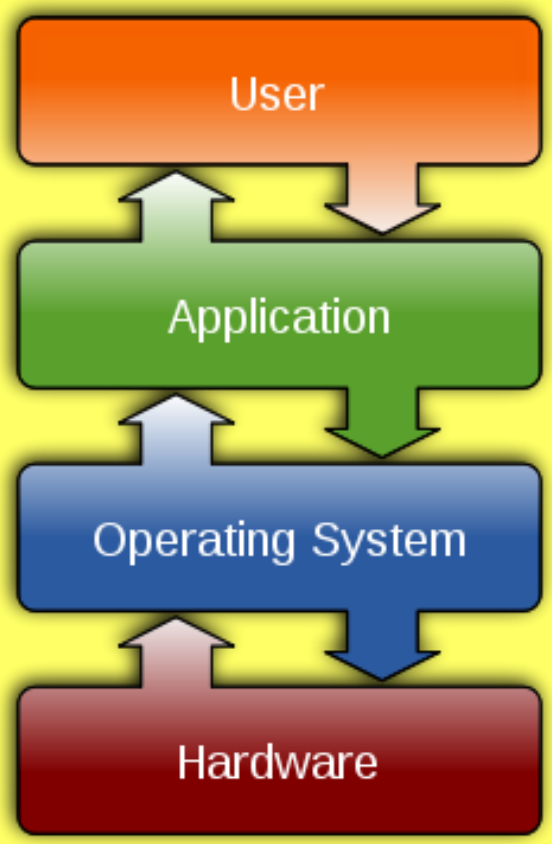

```
var Swi = xdc.useModule('ti.sysbios.knl.Swi');
var swi0Params = new Swi.Params();
swi0Params.instance.name = "MyNewSwi";
swi0Params.priority = 2;
Program.global.MyNewSwi = Swi.create("&ledToggle", swi0Params);
```

app.cfg

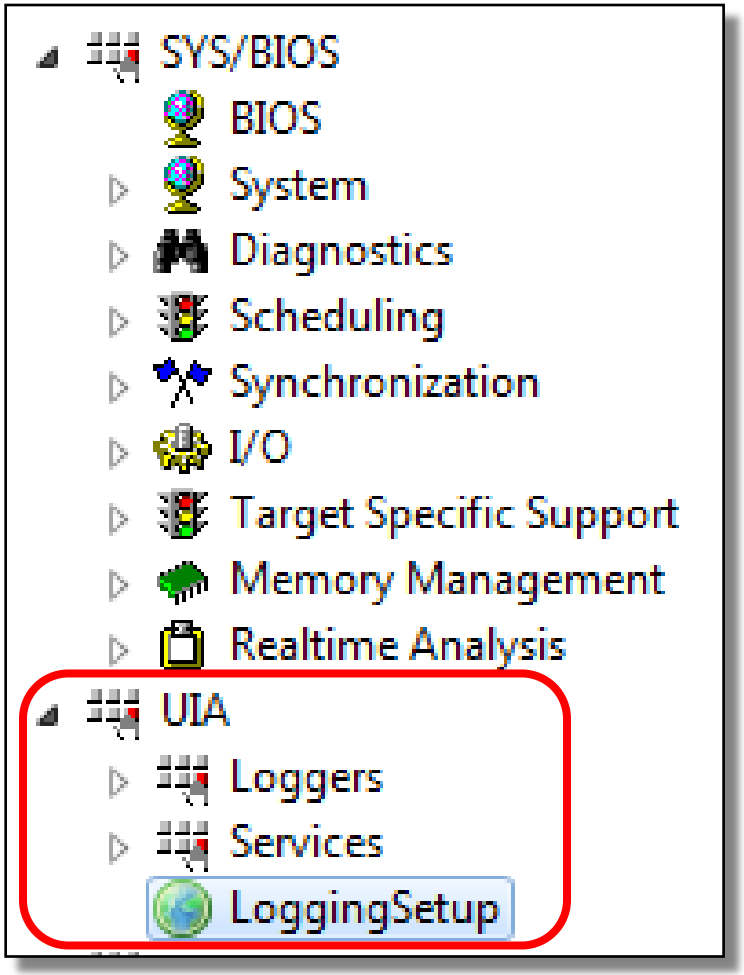
Note: more details on BIOS config in the next chapter…

Outline

- What is the TI-RTOS Kernel?
- Scheduler
- Adding Tasks…
- BIOS Environment
- BIOS Debugging Tools
- For More Info…
- Chapter Quiz



Built-in Debug Tools (UIA & ROV) – Intro



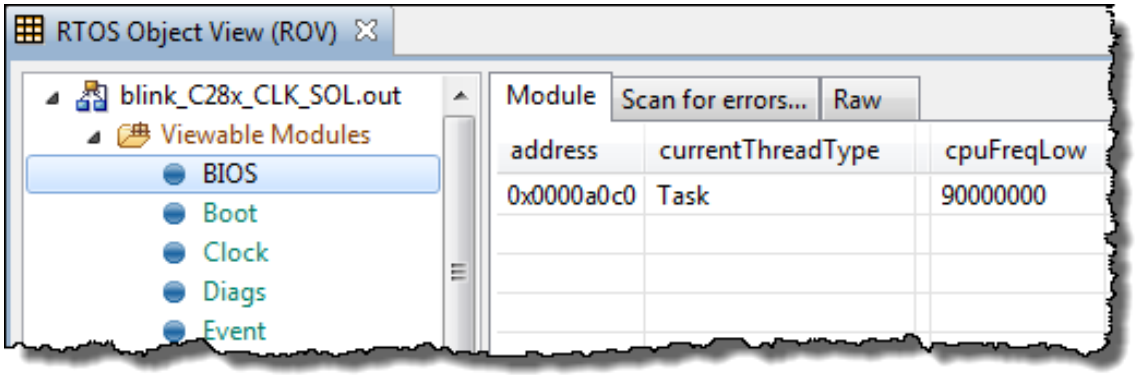
- **UIA -** Unified Instrumentation Architecture tools provide *visibility* into what is going on in your system:
  - Logging – "printf()-lite"
  - Execution Graph – software "logic analyzer"
  - Load – CPU/Thread loading
  - UIA replaces the older RTA tools – requires "LoggingSetup"
- **ROV** – RTOS Object Viewer – see status of BIOS objects in your system (when halted)

Let's look at what these tools look like in CCS...

ROV and UIA – Visibility/Debug Tools

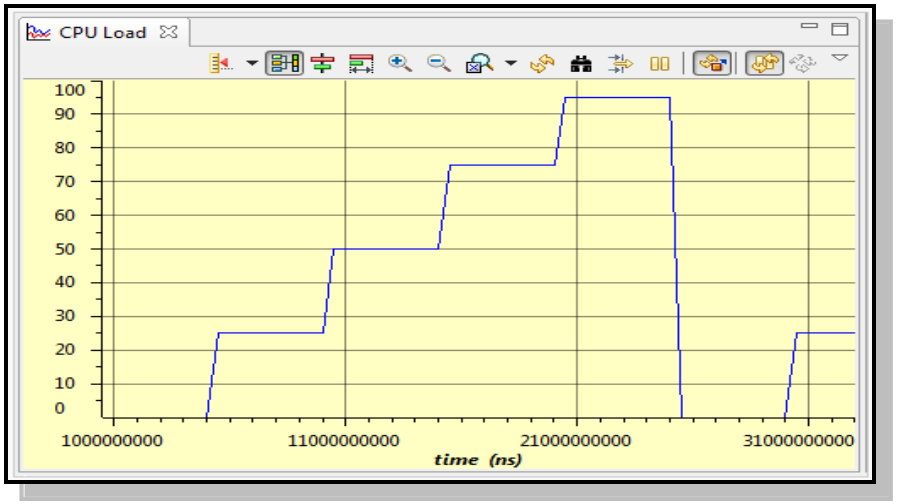
Real-time is…

- Gather data on target (30-40 CPU cycles)
- Format data on host (1000s of host PC cycles)
- Data gathering does NOT stop target CPU
- Halt CPU to see results (stop-time debug)



RTOS Obj Viewer (ROV)

- Halt to see results
- Displays stats about all threads in system



CPU/Thread Load Graph

- Analyze time NOT spent in Idle

ROV and UIA – Visibility/Debug Tools

Logs

- Send DBG Msgs to PC
- Data displayed during stop-time
- Deterministic, low CPU cycle count
- WAY more efficient than traditional `printf()`

*Live Session | Execution Graph

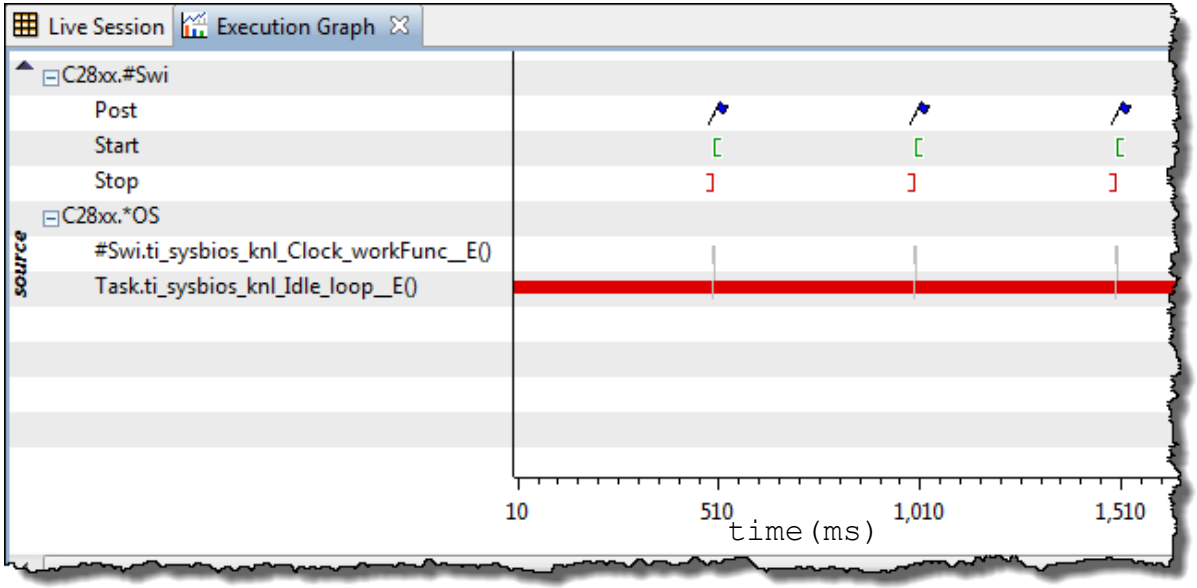
	Type	Time	Master	Message
18	i	4500296077	C28xx	[../main.c:119] LED BENCHMARK = [464] CYCLES
19	i	5000253544	C28xx	[../main.c:117] LED TOGGLED [10] TIMES
20	i	5000296033	C28xx	[../main.c:119] LED BENCHMARK = [464] CYCLES
21	i	5500253533	C28xx	[../main.c:117] LED TOGGLED [11] TIMES
22	i	5500296022	C28xx	[../main.c:119] LED BENCHMARK = [464] CYCLES
23	i	6000253522	C28xx	[../main.c:117] LED TOGGLED [12] TIMES

```
Log_info1("LED TOGGLED [%u] times", count);
```

Execution Graph

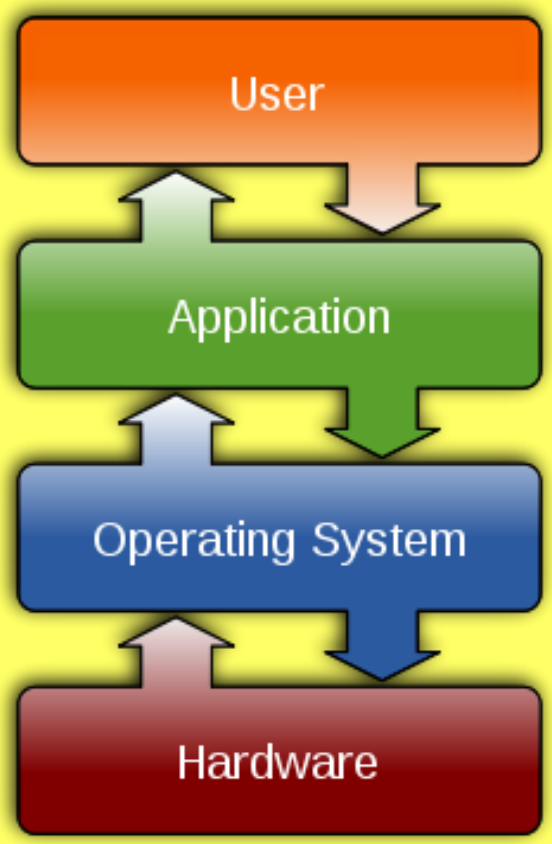
- View system events down to the CPU cycle…
- Calculate benchmarks

Note: more details on how to configure these tools in the next chapter…



Outline

- What is the TI-RTOS Kernel?
- Scheduler
- Adding Tasks…
- BIOS Environment
- BIOS Debugging Tools
- For More Info…
- Chapter Quiz



For More Information

- **TI-RTOS Kernel Product Page (www.ti.com/sysbios)**

Part Number	Buy from Texas Instruments or Third Party	Status	Current Version	Version Date
TI-RTOS-KERNEL: (Formerly known as SYS/BIOS)	Free Get Software	ACTIVE	v6.40.01.15	23-Apr-2014

Description

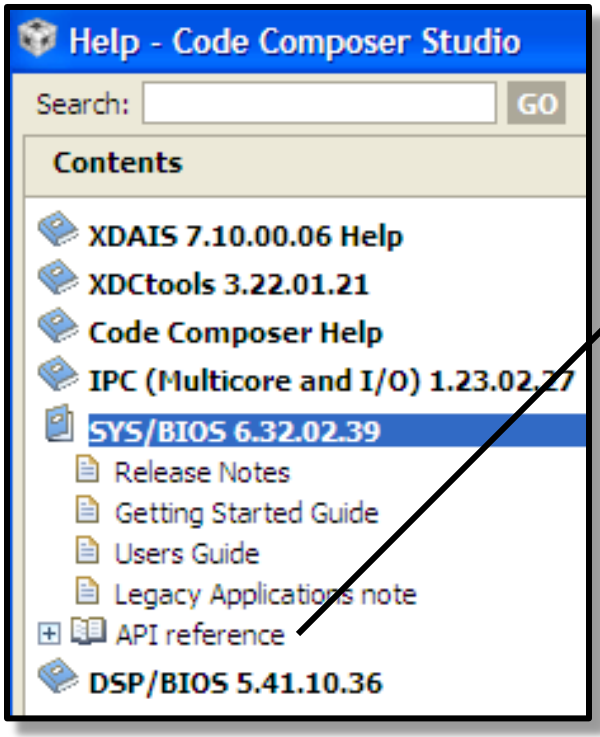
Advanced Real-Time Kernel Solution

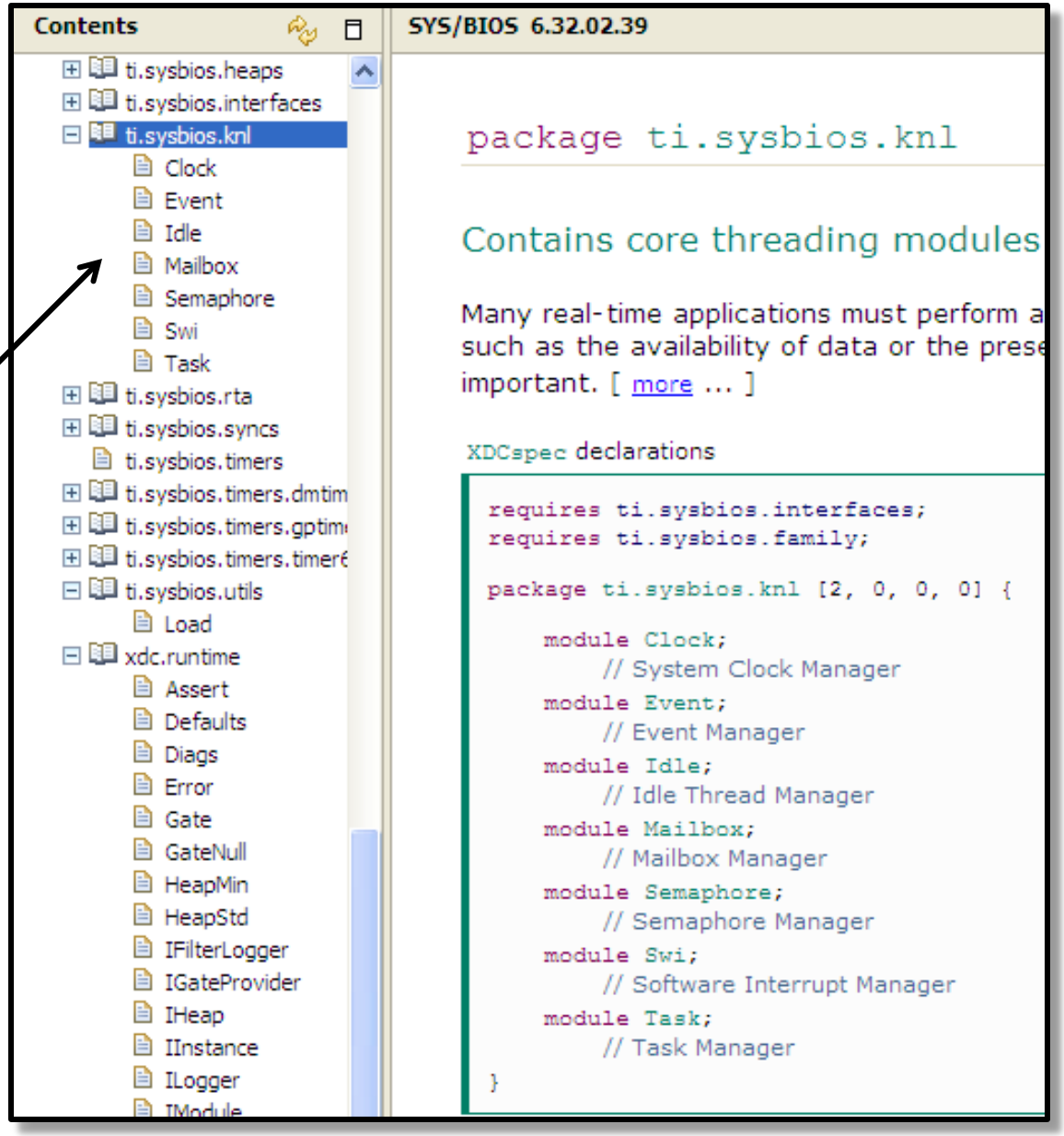
TI-RTOS Kernel (formerly known as SYS/BIOS™) is an advanced, real-time kernel for use in a wide range of DSPs, ARMs, and microcontrollers. It represents the successor product to the well-known DSP/BIOS real-time kernel, which has been used in thousands of DSP applications. It provides preemptive multitasking, hardware abstraction, and memory management. TI-RTOS Kernel is at the core of TI-RTOS, a full-featured real-time operating system including drivers drivers, networking and USB stacks. TI-RTOS is available for select TI devices.

TI-RTOS Kernel is currently available for TI C64x+ core based devices (including the OMAP35x™ and DaVinci™ digital video processors), TMS320C674x™ devices (including OMAP-L13x), TMS320C66x™ multicore processors, Sitara™ ARM9® Cortex A8® microprocessors, as well as TMS320C28x™, Tiva™ Cortex M4™, and MSP430™ microcontrollers.

For More Information (2)

- **CCS Help Contents**

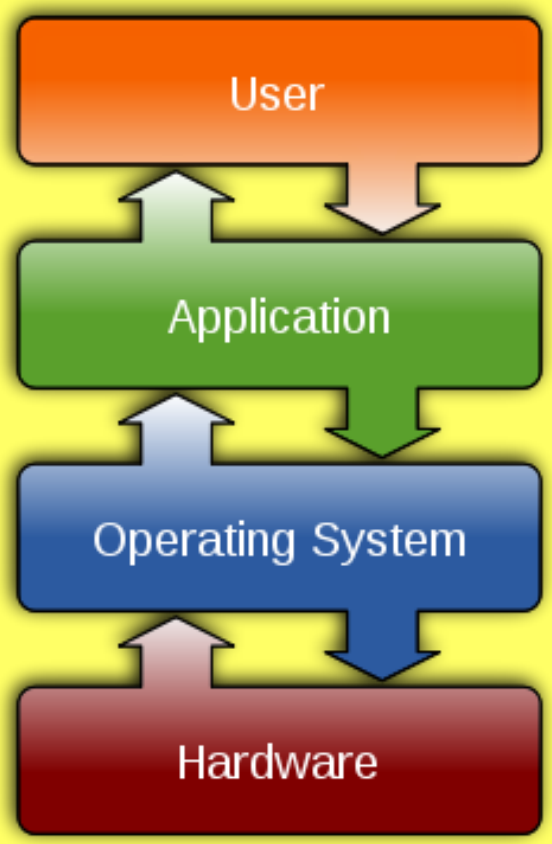




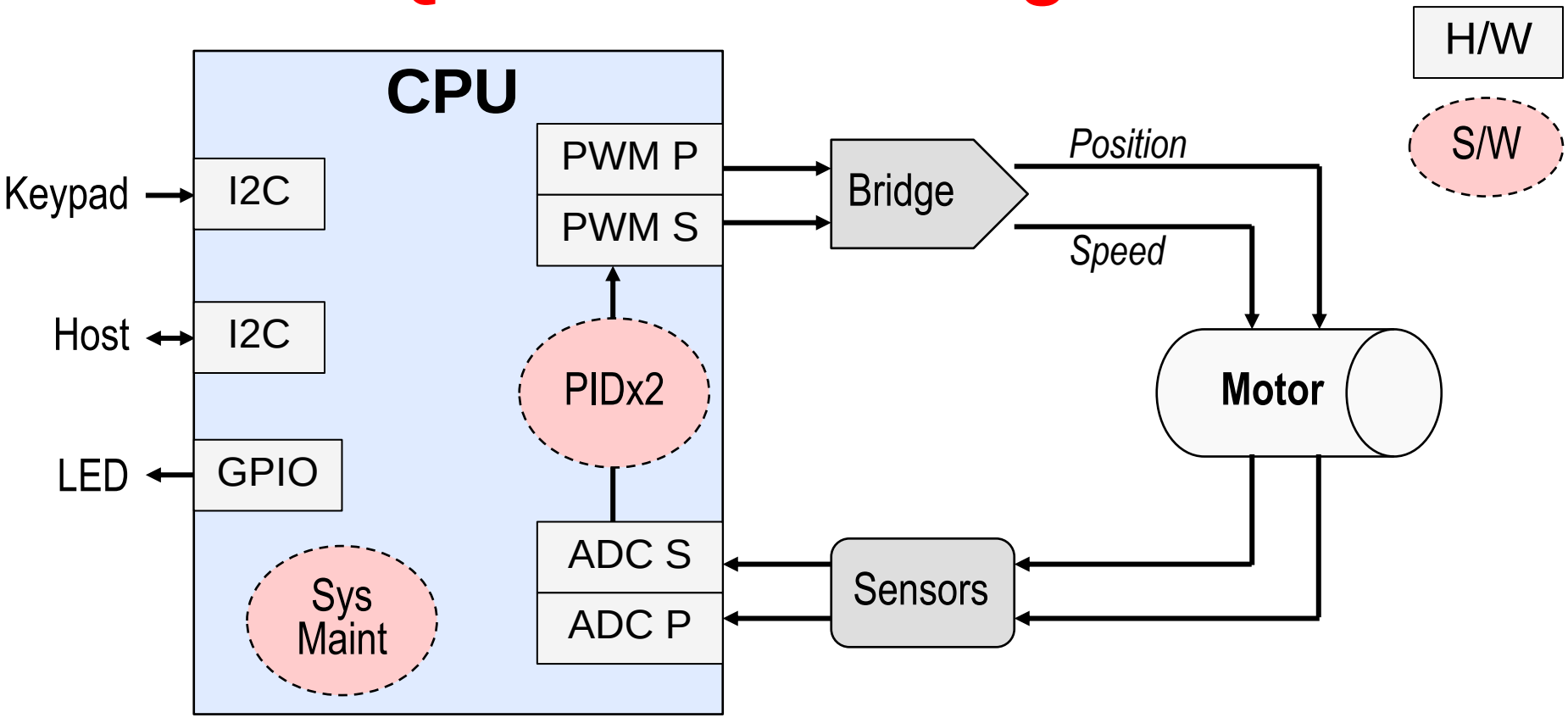
- User Guides
- API Reference (knl)

Outline

- What is the TI-RTOS Kernel?
- Scheduler
- Adding Tasks…
- BIOS Environment
- BIOS Debugging Tools
- For More Info…
- Chapter Quiz



Quiz – Block Diagram



Basic Motor Control System

- Goal – Control motor speed/position via PID algo based on ADC info and output control to PWM
- Other services include: *Keypad, LED, Host, System Maintenance*

Quiz

- Fill in the missing info in the tables (next page) regarding HOW to schedule the threads in the system
- Think about **Priority** and the **TYPE** of BIOS thread you would assign to each

Quiz – Fill in the missing pieces…

System Threads

Hwi's	S/W function	BIOS Thread Type	S/W Priority?
ADC_P_ISR	PID_Position		High -
ADC_S_ISR	PID_Speed		
Host_ISR	Host_Cmd_Proc		Med -
Keypad_ISR	Keypad_Read		Low -
LED_blink_ISR	LED_toggle		
	Sys_maint		Lowest -

- **Hwi's**: triggered by interrupt, ISR called via BIOS Hwi.
- **S/W function**: called by "BIOS Thread Type" (*e.g. Swi 5 calls PID_Position*)
- **BIOS Thread Type**: choices are – Hwi, Swi, Task, Idle
- **S/W Priority**: Swi (0-15/31), Task (0-15/31), Idle (0)

Bonus Question

If you had ONE timer and needed to run 5 different threads based off that timer, how would you accomplish this?

Click for ALL answers…

Quiz – One "Solution"

System Threads

Hwi's	S/W function	BIOS Thread Type	S/W Priority?
ADC_P_ISR	PID_Position	Swi	High – Swi 5
ADC_S_ISR	PID_Speed	Swi	Swi 3
Host_ISR	Host_Cmd_Proc	Task	Med – Task 5
Keypad_ISR	Keypad_Read	Task	Low – Task 3
LED_blink_ISR	LED_toggle	Task	Task 2
	Sys_maint	Idle	Lowest – Idle

- **Hwi's**: triggered by interrupt, ISR called via BIOS Hwi.
- **S/W function**: called by "BIOS Thread Type" (*e.g. Swi 5 calls PID_Position*)
- **BIOS Thread Type**: choices are – Hwi, Swi, Task, Idle
- **S/W Priority**: Swi (0-15/31), Task (0-15/31), Idle (0)

Bonus Question

If you had ONE timer and needed to run 5 different threads based off that timer, how would you accomplish this? Use BIOS Clock Functions.

Texas Instruments