TECH DAYS
Texas Instruments

SimpleLink™ SDK RTOS Solutions

Todd Mullanix

TI-RTOS Apps Manager

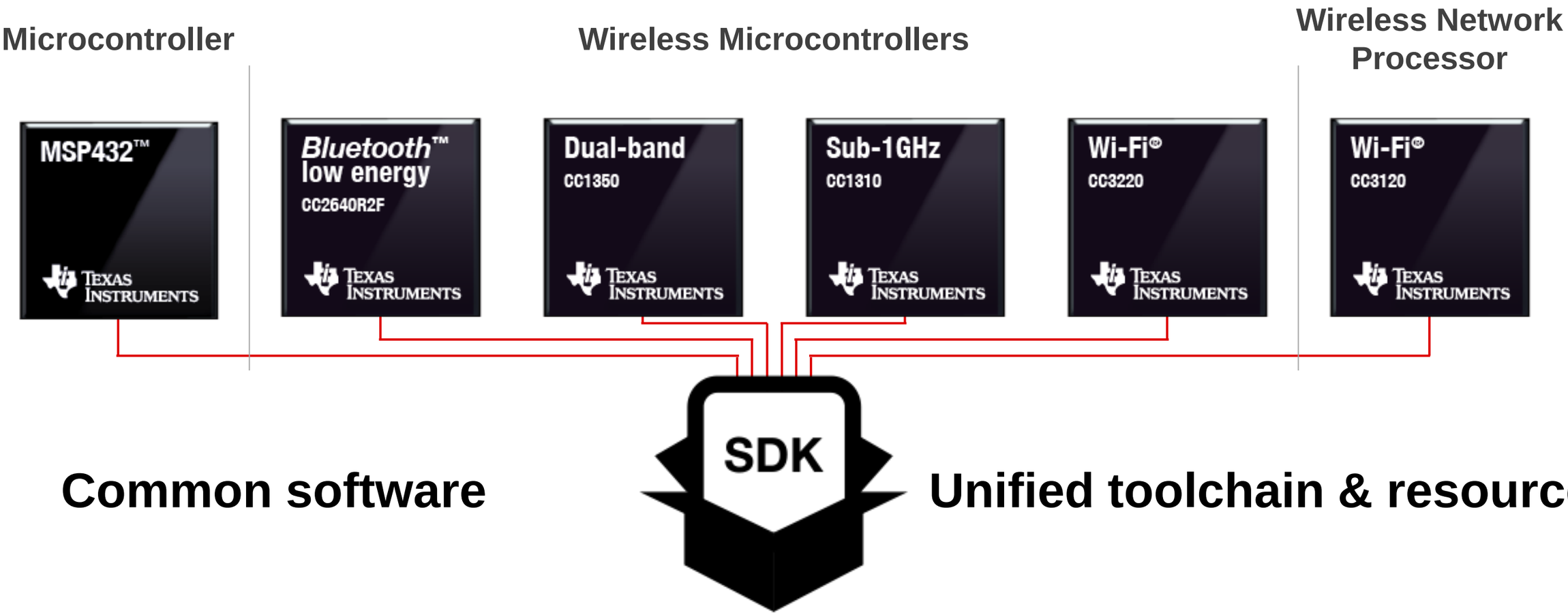
Oct. 16, 2017

Agenda

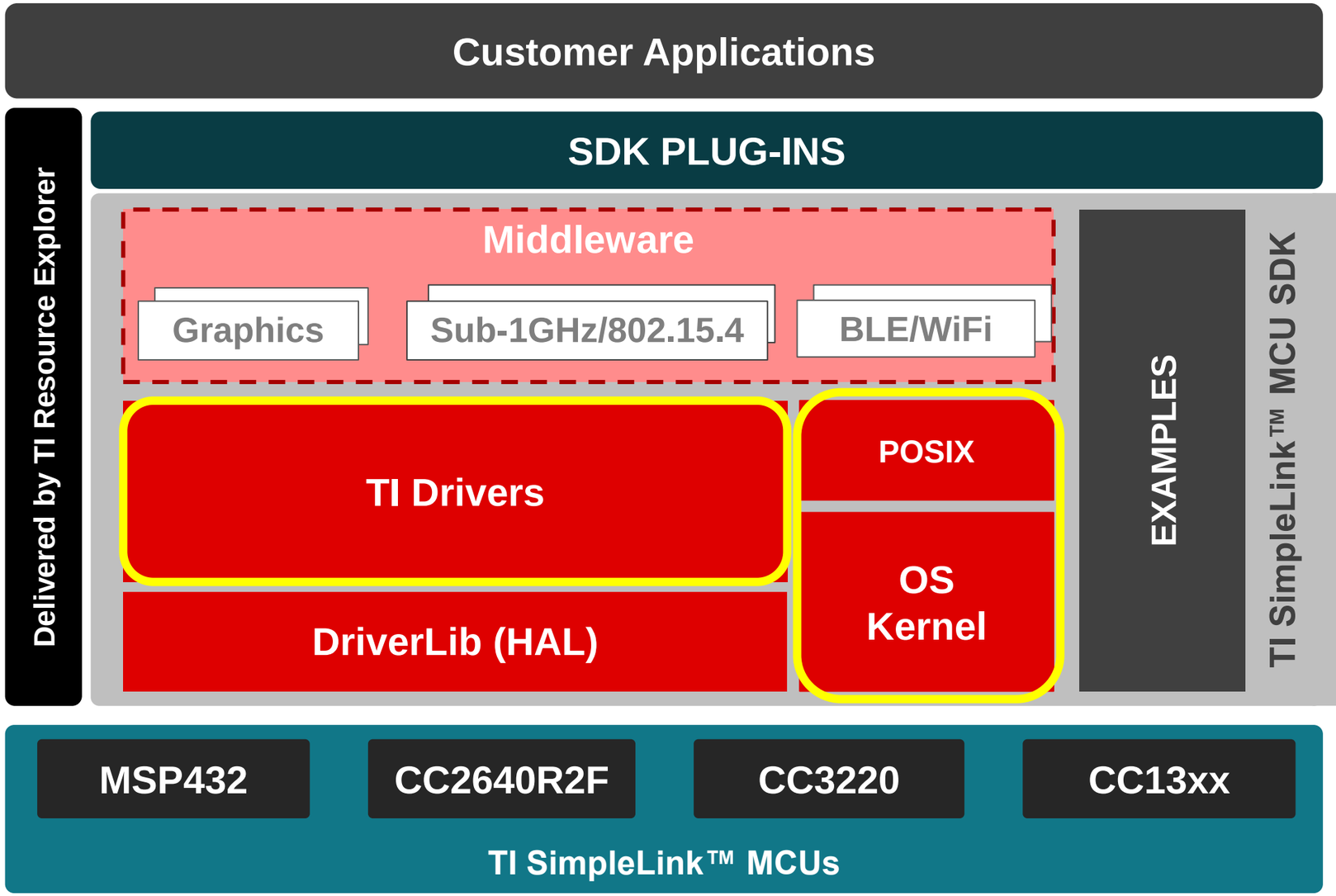
Here’s the high-level view of what we will cover in this presentation:

1. SimpleLink SDK
2. When to use an RTOS
3. RTOS Options in SimpleLink SDK
    1. TI-RTOS
    2. FreeRTOS
    3. POSIX
4. TI Drivers & RTOS
5. What is a Task and when to use one?
6. Resources

SimpleLink™ MCU platform



SimpleLink™ MCU SDK



SDK Components

Drivers

- **TI Drivers** portable, feature-rich access to peripherals
- **DriverLib** hardware abstraction layer (HAL) access for device specific optimization
- Support for TI Drivers without kernel coming soon

OS/Kernel

- Real-time Multi-tasking operating system
- POSIX compliant API enables use of a variety of RTOS kernels – validated with TI-RTOS & FreeRTOS

Middleware

- Communication stacks
- Common libraries (i.e. graphics, CapTouch, FatFs)

Examples and documentation

- Large number of examples, documentation and training make it easy to start developing applications

RTOS vs General Purpose Operating System

A real-time application has hard timing constraints. At a high level, an RTOS allows those constraints to be met, while a General Purpose Operating System (e.g. Linux) does not.

Here is a brief highlight of the differences between the two operating systems.

Traits	RTOS	GPOS
Scheduling	Efficient and Predictable	Efficient but not predictable
Features	Relatively simple	Wider range
Latency	Minimize	More focused on maximizing throughput at the expense of potential increased latency
Memory Model	Generally *Flat*. Memory appears as a single contiguous address space	Generally *Paged*. User vs kernel space
Size	Small and very scalable	Larger and less scalable
Booting	Fast (e.g. <1ms)	Not so fast

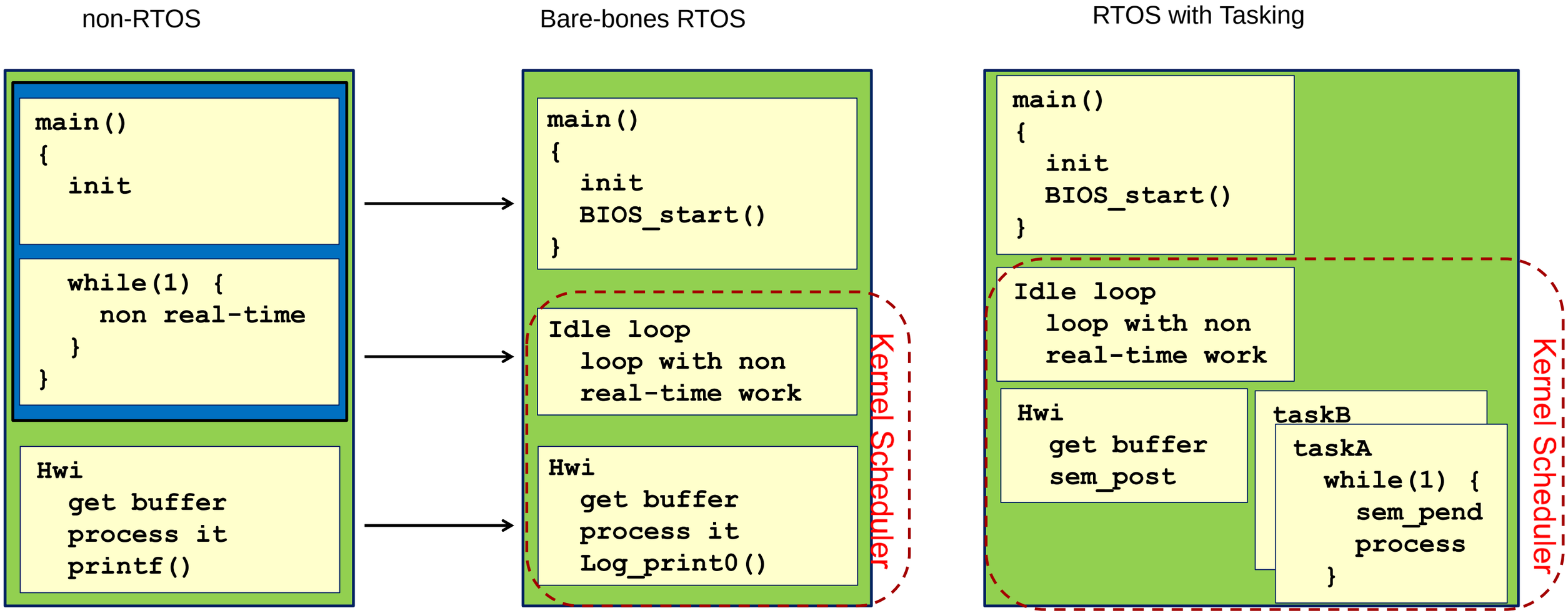
When to consider an RTOS

- The advantage of using an RTOS increase with application complexity
- Key considerations that influence how beneficial an RTOS would be include:
  - Number of interrupts that require processing too complex for an ISR
  - Number of different system functions the application must run
    - Note that communication stacks often need to create one or more threads
  - Size of the application
    - Code re-use becomes more important as the application is large, as rewriting from scratch has too great a schedule impact

	# of deferred Interrupt sources			# of system functions			Code Reuse	
	1-2	3-5	>6	1-2	3-5	>6	Unimportant	Important
Use RTOS?	No	Maybe	Yes	No	Maybe	Yes	No	Yes

Non-RTOS vs Bare-bones RTOS vs Full RTOS

Comparison between non-OS and RTOS execution structure.

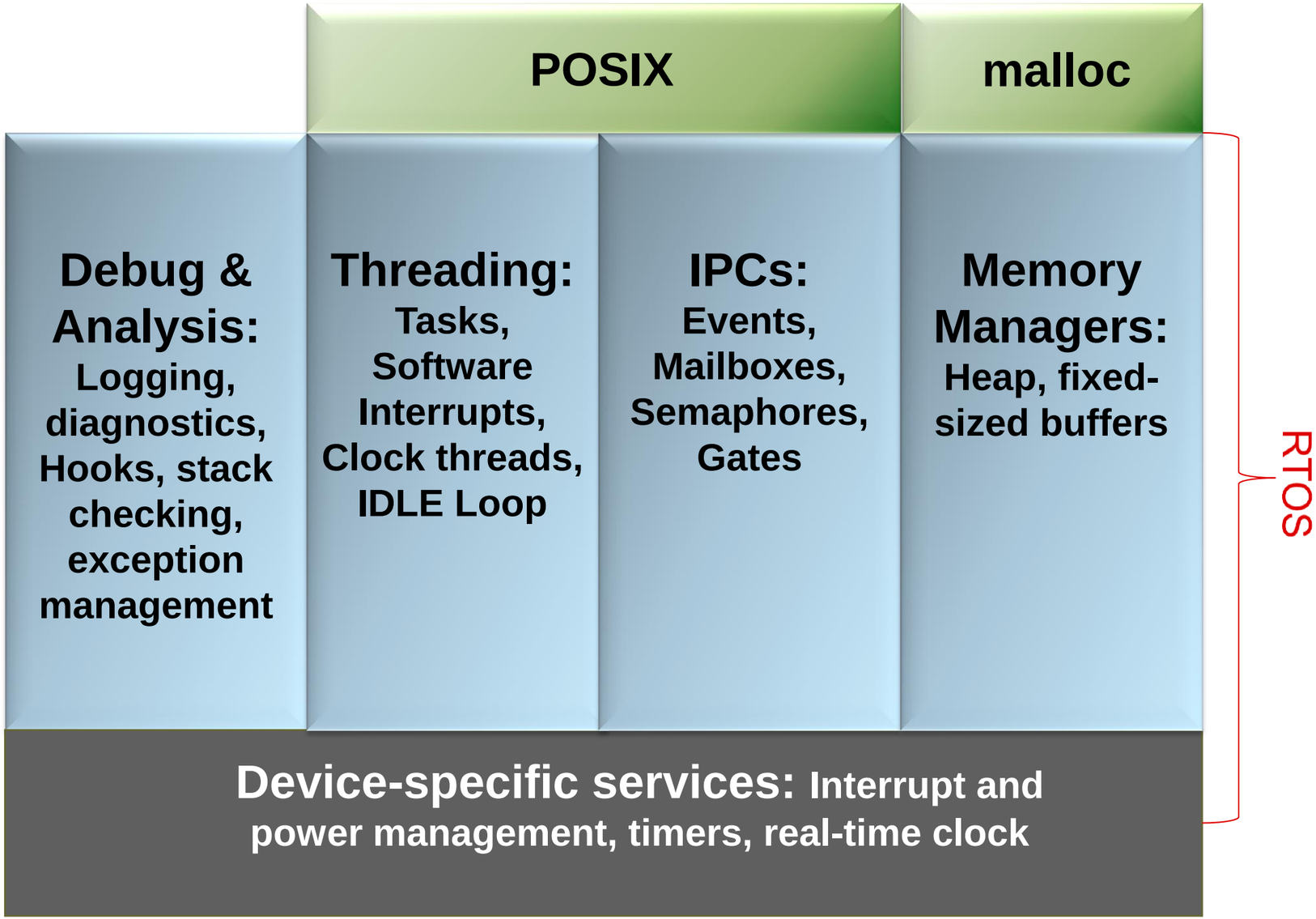


Benefits of using an RTOS

- An RTOS helps manage complexity and creates a more maintainable and reusable codebase
  - The multithreading paradigm encourages a compartmentalized design
  - The priority-based preemptive scheduling paradigm enables new system functions to be added without affected the response to the most critical real-time events
  - RTOS offerings are designed to work with many applications and naturally scale as an application evolves
    - A custom scheduler or loop will likely need on-going enhancement
  - An RTOS and its associated drivers abstract away the HW specifics and inherently encourage a more portable design
    - Note that a ‘No OS’ application using the SimpleLink SDK can gain much of this benefit by using TI drivers

POSIX vs. RTOS

- **POSIX** is an IEEE industry API standard for OS compatibility
- The SDK's POSIX APIs sit on top of TI-RTOS (or FreeRTOS)
- TI-RTOS APIs can be used along with POSIX APIs
- Not all the threading features of the RTOS are exposed with the POSIX APIs
- POSIX does not include ISR support
- Note: TI Drivers are not part of this since they are RTOS-agnostic



Both RTOS kernels (and No OS) have full SDK entitlement

SDK Feature	TI-RTOS	FreeRTOS	No RTOS
Examples	Yes	Yes	**Yes, about 75% of total. Multithreading applications not supported**
Drivers (incl. network stacks)	Yes	Yes	Yes
Board initialization	Yes	Yes	Yes
Power management	Yes	Yes	Yes
CCS support	Yes	Yes	Yes
IAR support	Yes	Yes	Yes
GCC support	Yes	Yes	Yes

RTOS support by SimpleLink MCU family

MCU	TI-RTOS	FreeRTOS	No OS
CC3220	Yes	Yes	Yes
MSP432	Yes	Yes	Yes
CC13xx	Yes, kernel in ROM	No	Yes
CC26xx	Yes, kernel in ROM	No	Yes

- TI-RTOS is included in SimpleLink SDK
- FreeRTOS kernel must be downloaded separately from: …
  - Note that FreeRTOS DPL and POSIX threads API layer is included in SimpleLink SDK

TI-RTOS, FreeRTOS business considerations

TI-RTOS

- **Cost:** OS and associated OS-aware tools are free
- **License:** Open Source (BSD)
- **Support:** Kernel and associated tools are supported directly by TI
- Above also apply to TI's 'No OS' solution

FreeRTOS

- **Cost:** OS is free. Associated OS-aware tools may require cost.
- **License:** Open Source (modified GPL v2). Exception allows customers not to ship the rest of their application source code.
- **Support:** Kernel supported by FreeRTOS.org. Associated tools may require support agreements with appropriate third-parties

RTOS configuration

- SimpleLink SDK provides ‘canned’ configurations for FreeRTOS and TI-RTOS
  - These are set up in ‘kernel’ projects that are then imported to an actual CCS or IAR project for an example or the customer’s application
- Having the ‘kernel’ project importation approach allows the kernel configuration to be managed in a single location by the selected OS expert
  - In the case of TI-RTOS, the other application developers don’t see .cfg files or .xdc tools
- These kernel project configurations will be referred to as a ‘build’ going forward

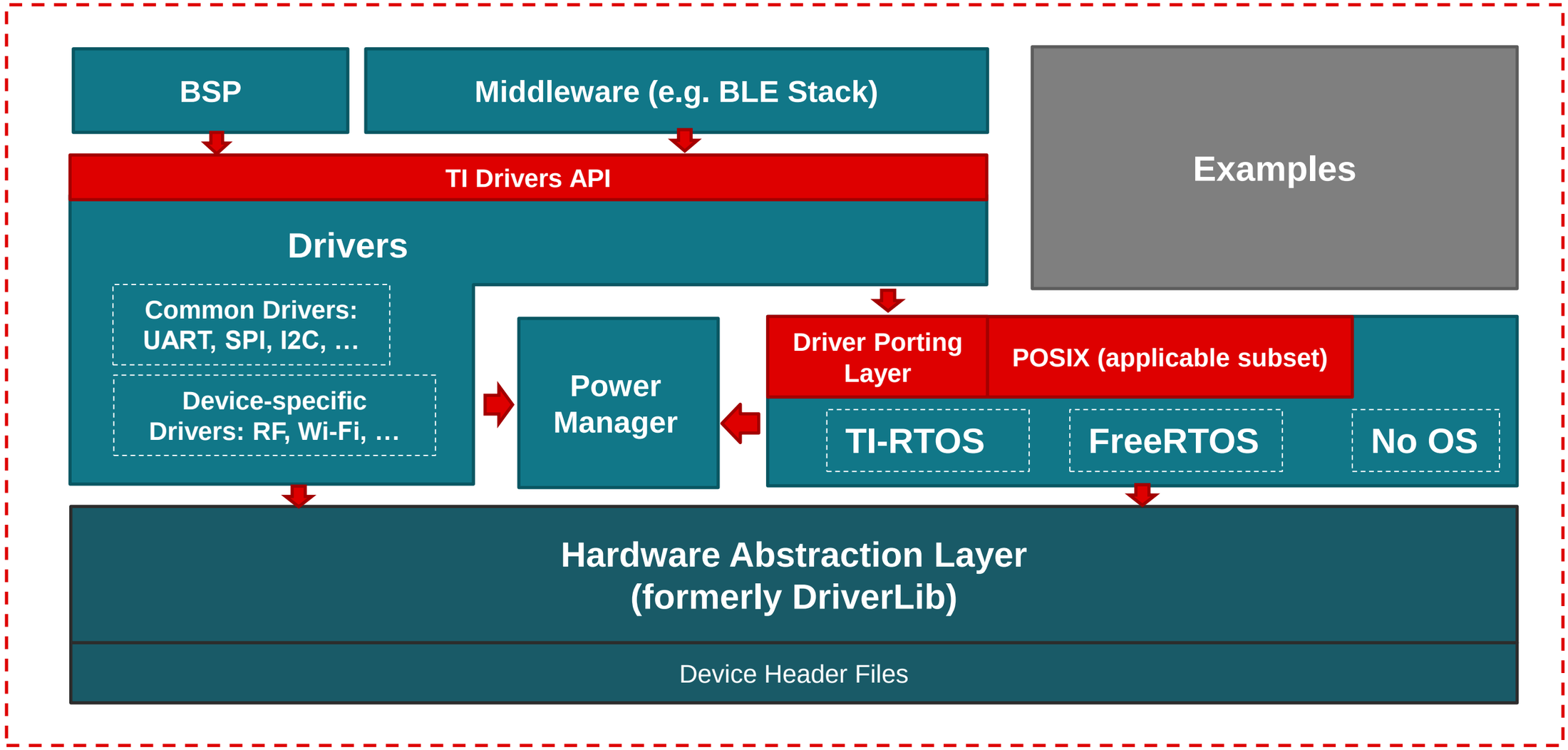
TI-RTOS

- **Configuration is done via a Javascript .cfg file**
- **Kernel then built by xdctools and user’s development tools**

FreeRTOS

- **Configuration is done via .h file**
- **Kernel is then built with user’s development tools**

RTOS Support in the SimpleLink SDK



Thread Types

Hwi
Hardware Interrupts

- Hardware event triggers Hwi to run
- BIOS handles context save/restore, nesting
- Hwi triggers follow-up processing
- Depending device, priorities maybe configurable

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*'
- Designed to run concurrently – blocks when waiting for data
- Up to 32 priority levels (16 on C28x/MSP430/55x)

Idle
Background

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

Typical Task

Here is a typical Task function. Tasks generally loop on some condition and “wait” for something to occur.

The “waiting” is generally a blocking call that does not use the CPU and allows other threads to execute.

Here are typical blocking calls that a task uses:

- Semaphore_pend
- Mailbox_pend
- Event_pend
- Driver APIs (e.g. UART_read or I2C_transfer)
- Stacks (e.g. recv, read)
- Task_sleep

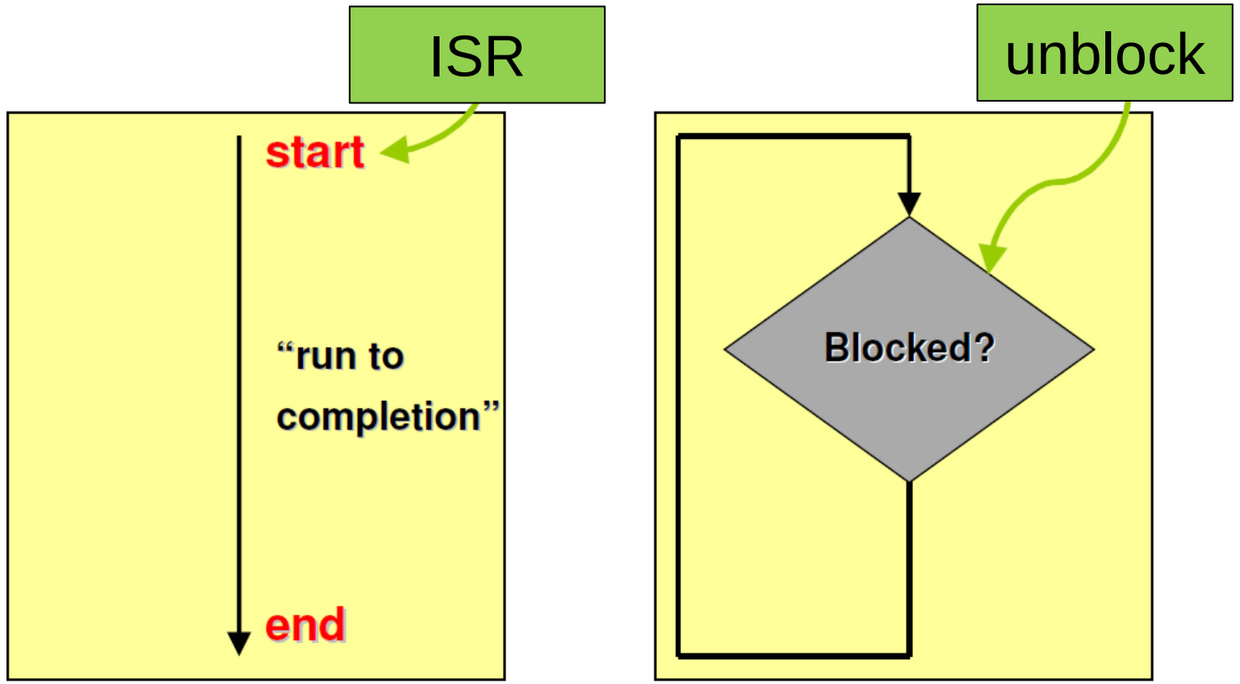
```
Void taskFxn(Arg arg0, Arg arg1)
{
    /* Prolog */

    while (‘condition’) {
        /*
         * Block waiting for
         * notification/data
         * (e.g Semaphore_pend).
         */

        /* Process */
    }

    /* Epilog */
}
```

Interrupt vs Task



A common question is whether to use a Interrupt or Task? It's really up to the application writer.

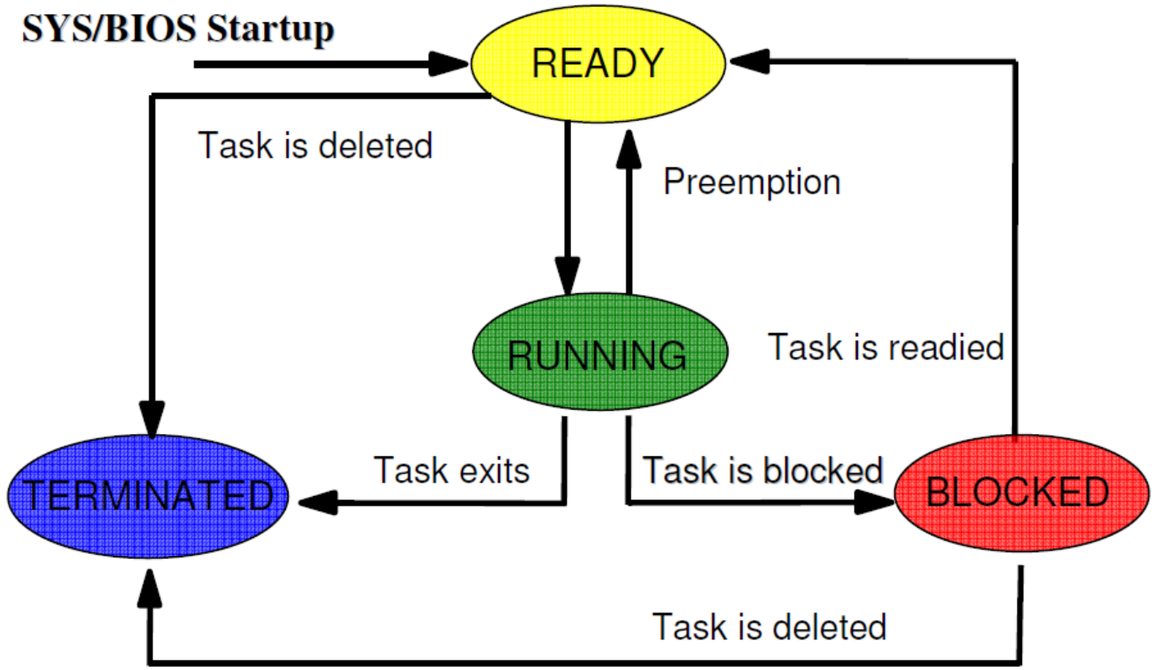
Tasks are generally better when dealing with drivers and middleware stacks (e.g. networking stack, BLE stack, etc.).

Task States

The highest priority task (call it TaskA) runs until

- A higher priority thread (Task, Swi or Hwi) becomes ready. The task (TaskA) moves to the ready/preempted state and the higher priority thread runs.
- That task becomes blocked (e.g. calls `Task_sleep()` or `Semaphore_pend()`). Then the new highest priority task that is ready will run.
- The task terminates.

Here is a pictorial view



Task Stacks

Each Task has it's **own unique stack**. This is required since they must maintain their state. Items on the stack includes **local variables, registers and return addresses**.

When the task is created, the user can specify the stack size and supply the actual stack.

```
#define TASKSTACKSIZE   512
Char taskStack[TASKSTACKSIZE];
Task_Params_init(&taskParams);
taskParams.stackSize = TASKSTACKSIZE;
taskParams.stack = &task1Stack;
Task_construct(&taskStruct, (Task_FuncPtr)myTask, &taskParams, &eb);
```

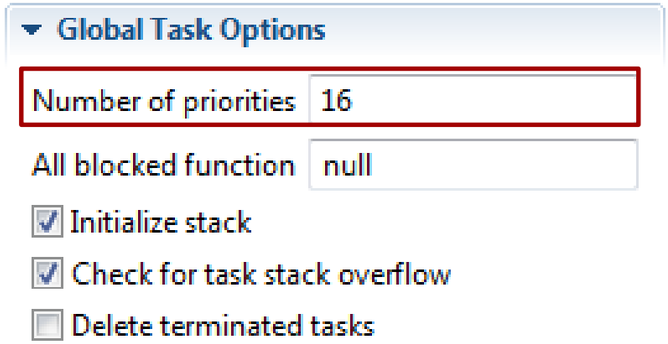
Here's a good video/presentation on how to manage your task stacks:

https://training.ti.com/debugging-common-application-issues-ti-rtos

Task Priorities

Task priorities range from -1 to `Task.numPriorities` (specified in the TI-RTOS configuration file). Where

- -1 is for inactive tasks
- 0 is reserved for the Idle Task



A **lower values** means the task is a **lower priority**.

At start-up (e.g. when `BIOS_start()` is called), the **highest priority task runs first***. If there are multiple tasks with the same priority, the order of creation dictates the order.

Once the system is running, the **highest priority task runs***. If there are multiple tasks with the same priority, the one that became ready first runs.

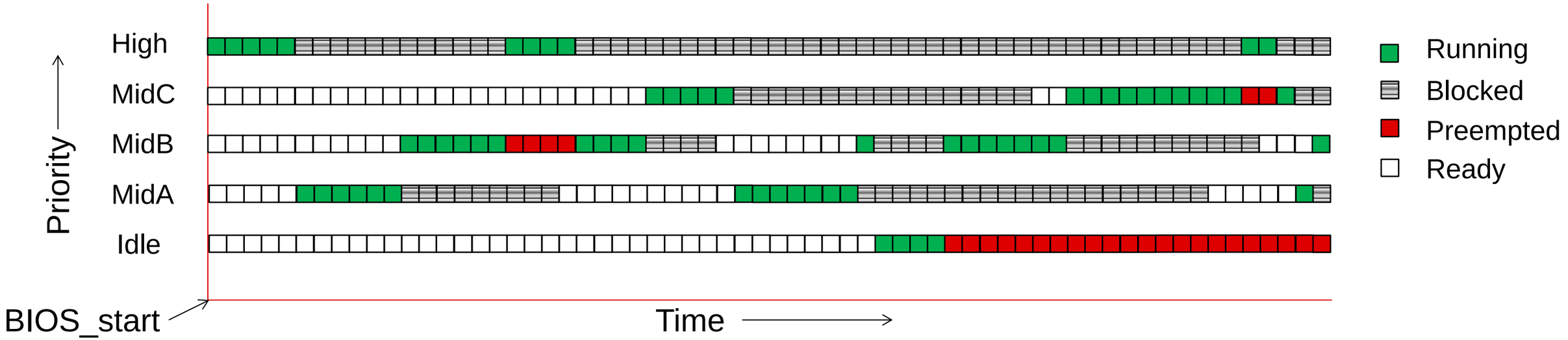
The fact that tasks of the same priority cannot be running at the same time can be used as a **simple mutual exclusion**.

* After any Hwi or Swi that is ready

Task Priorities Startup

Let’s look at an application with three tasks the same priority and one task with a higher priority. Note the start-up order of execution of the tasks with the same priority. The first one created in the .cfg (MidA) runs before the other Mid tasks. Once it blocks, MidB runs.

- MidA: created first in .cfg file priority 4
- MidB: created second in .cfg file priority 4
- MidC: created first in `main()` priority 4
- High: created second in `main()` priority 8



Task Communication: Mailboxes

A common use-case for a task is to centralize a functionality into one location. For example, a task could be used to update an LCD display. Other tasks could send a Mailbox message to the LCD Task.

```
Void taskA(Arg arg0, Arg arg1)
{
    MyMsg msg;

    while(true) {
        …
        msg.text = “Please select”
        msg.duration = 5;
        msg.color = BLUE;
        Mailbox_post(mailboxLCD, &msg,
            BIOS_WAIT_FOREVER);
    }
}
```

```
Void lcdTask(Arg arg0, Arg arg1)
{
    MyMsg msg;
    initLCD()
    while (true) {
        Mailbox_pend(mailboxLCD, &msg
                        BIOS_WAIT_FOREVER);
        updateLCD(msg);
    }
}
```

Mailbox is copied based. For large buffers, it common to pass the address of buffer instead of the content.

Task Communication: Mailboxes

Could you use Mailbox from an ISR?

Yes! You just need to use a zero timeout.

```
Void myISR(Arg arg0)
{
    MyMsg msg;

    msg.text = "Please select"
    msg.duration = 5;
    msg.color = BLUE;
    rc = Mailbox_post(mailboxLCD, &msg, 0);
    if (rc == FALSE){
        // Mailbox was full☹
        // handle accordingly…
}
```

```
Void lcdTask(Arg arg0, Arg arg1)
{
    MyMsg msg;
    initLCD()
    while (true) {
        Mailbox_pend(mailboxLCD, &msg
                     BIOS_WAIT_FOREVER);
        updateLCD(msg);
    }
}
```

Task Communication: Wait on Multiple

A task can wait on multiple things with the **Event** module. The following is a simplified copy of the “event” example in the SimpleLink SDK.

```
Void clk0Fxn(UArg arg0){
    Event_post(evtHandle, Event_Id_00);
}
```

```
Void clk1Fxn(UArg arg0){
    // this sem tied to Event_Id_01
    Semaphore_post(semHandle);
}
```

```
Void writertask(UArg arg0, UArg arg1){
    MsgObj      msg;
    while(true) {
        msg.id = i;
        msg.val = i + 'a';
        Mailbox_post(mbxHandle, &msg,
                     TIMEOUT);
    }   // mailbox tied to Event_Id_02
}
```

```
Void readertask(UArg arg0, UArg arg1) {
    MsgObj msg;
    UInt posted;
    for (;;) {
        posted = Event_pend(evtHandle,
                 Event_Id_00 + Event_Id_01,
                 Event_Id_02,
                 TIMEOUT);
        if (posted == 0) {
            continue;//expired
        }
        if ((posted & Event_Id_00) &&
            (posted & Event_Id_01)) {
            // both event and sem were posted
        }
        if (posted & Event_Id_02) {
            // Msg in Mailbox
```

RTOS footprint benchmarks – MSP432

Application	Flash (code)	RAM (data)	HWI stack peak	Task Stack Peak		Free Heap Space
				temp	console	
Portable POSIX TI-RTOS	26249	36886	456	600	576	30528
Portable POSIX FreeRTOS	26820	37424				
Portable Native TI-RTOS	22485	36782	440	568	496	30688
Portable Native FreeRTOS	23300	37392				

- Uses SimpleLink MSP432 SDK 1.30.00 CCA, release build
- RAM usage includes Task and System Stacks and 32KB heap
- The FreeRTOS release build includes asserts enabled, which TI-RTOS lacks in its release build. This inflates the FreeRTOS numbers by 1620.

Breaking down the ‘RTOS overhead’

Component	.text	.const	.cinit	.resetVecs	Total
TI-RTOS	9882	982	0	60	10924
POSIX API layer	1528	0	0	0	1528
C RTS	1764	96	0	0	1860
Drivers	8590	177	0	0	8857
‘memory holes’	274	0	7	0	281
Application	1536	676	577	0	2789
Total	23674	1931	584	60	26249

- Detailed breakdown of Portable POSIX TI-RTOS application
- Note total POSIX overhead is greater because of other dependencies
- Includes UART, I2C, and GPIO

Benchmarks

SimpleLink CC26xx SDK 1.30.00, CCS Codegen. Release kernel builds

Application	Flash (code)	RAM (data)	Hwi Stack Peak	Task Stack Peaks		Free Heap Space
				temp	console	
Portable POSIX TI-RTOS	21564	8206	448	408	540	2120
Portable Native TI-RTOS	19260	8110	392	360	444	2280

Note: RAM usage includes Task stack, System stack and 4K Heap for all cases.

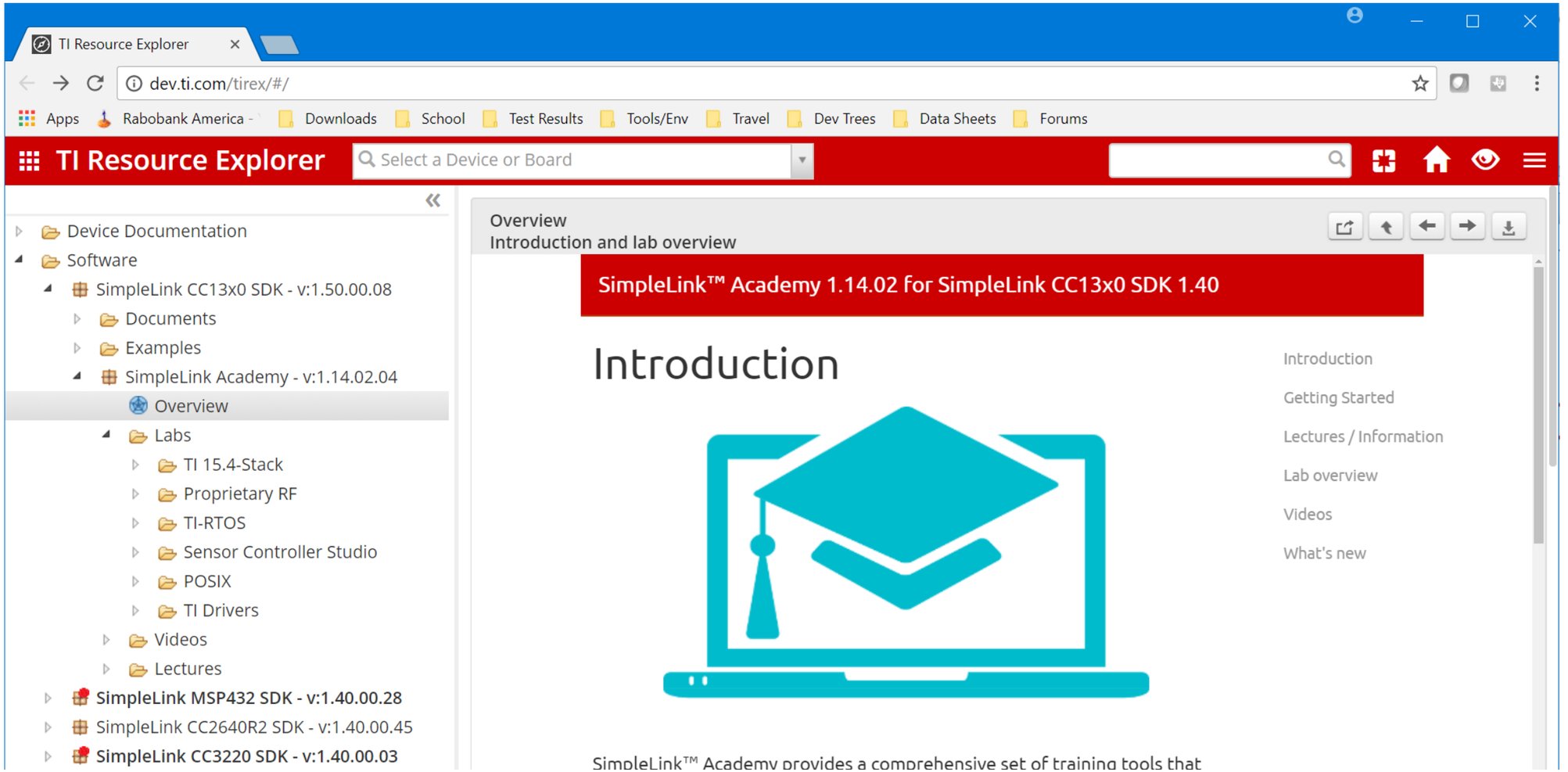
Flash Usage on CC2640R2

- Based on Portable POSIX TI-RTOS application
- TI-RTOS kernel is mostly in ROM, hence the lower .text footprint

Application	.text	.const	.cinit	.resetVecs	.ccfg	Total
TI-RTOS (includes pthreads)	3850	899	200	0	0	4949
Drivers	11990	328	0	0	0	12318
RTS	1702	0	0	0	0	1702
Holes	10	5	3	0	0	18
Application	1540	484	405	60	88	2577
Total	19092	1716	608	60	88	21564

SimpleLink Academy

SimpleLink Academy at http://dev.ti.com/tirex/#/ offers hands-on lab/quizzes/videos/etc. for many components in the SimpleLink SDKs.



Additional Resources (outside of SDK)

- TI-RTOS
  - 2-Day Workshop: https://training.ti.com/ti-rtos-workshop-series
  - Numerous topics on http://processors.wiki.ti.com/index.php/Category:TI-RTOS
- FreeRTOS
  - http://www.freertos.org/
- POSIX
  - Your favorite search engine…
- TI Online Support
  - https://e2e.ti.com/ Actively maintained by TI engineers.