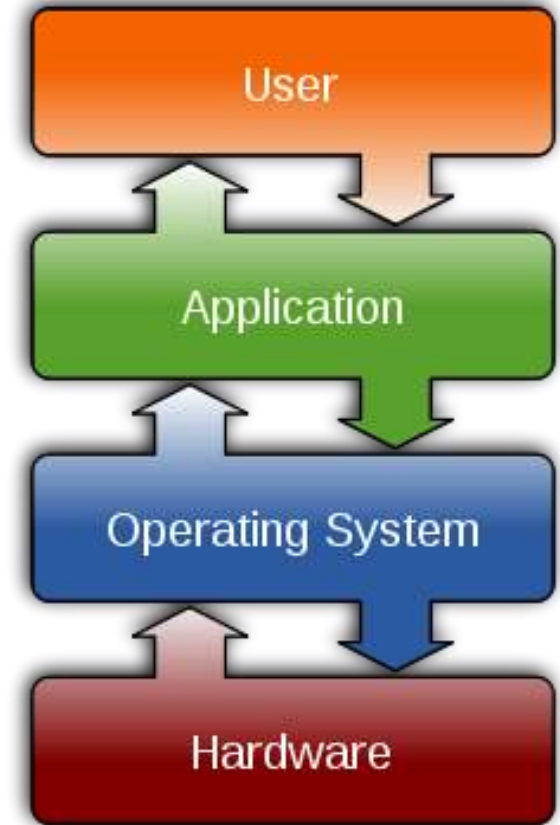


Introduction to SYS/BIOS

Outline

◆ Intro to SYS/BIOS

- ◆ Overview
- ◆ Threads and Scheduling
- ◆ Creating a BIOS Thread
- ◆ System Timeline
- ◆ Real-Time Analysis Tools
- ◆ Create A New Project
- ◆ BIOS Configuration (.CFG)
- ◆ Platforms
- ◆ For More Info.....
- ◆ BIOS Threads



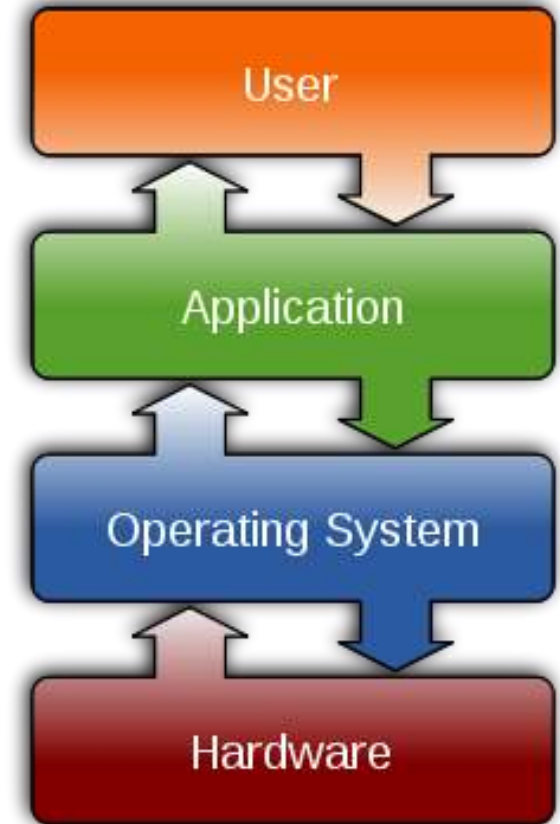
Outline

◆ Intro to SYS/BIOS

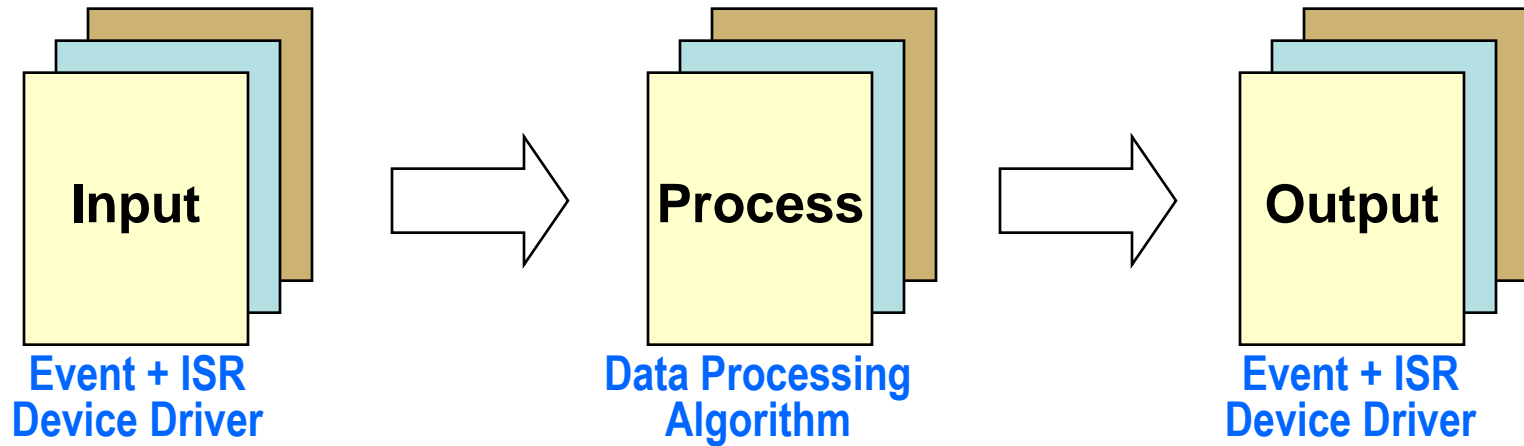
◆ Overview

- ◆ Threads and Scheduling
- ◆ Creating a BIOS Thread
- ◆ System Timeline
- ◆ Real-Time Analysis Tools
- ◆ Create A New Project
- ◆ BIOS Configuration (.CFG)
- ◆ Platforms
- ◆ For More Info.....

◆ BIOS Threads

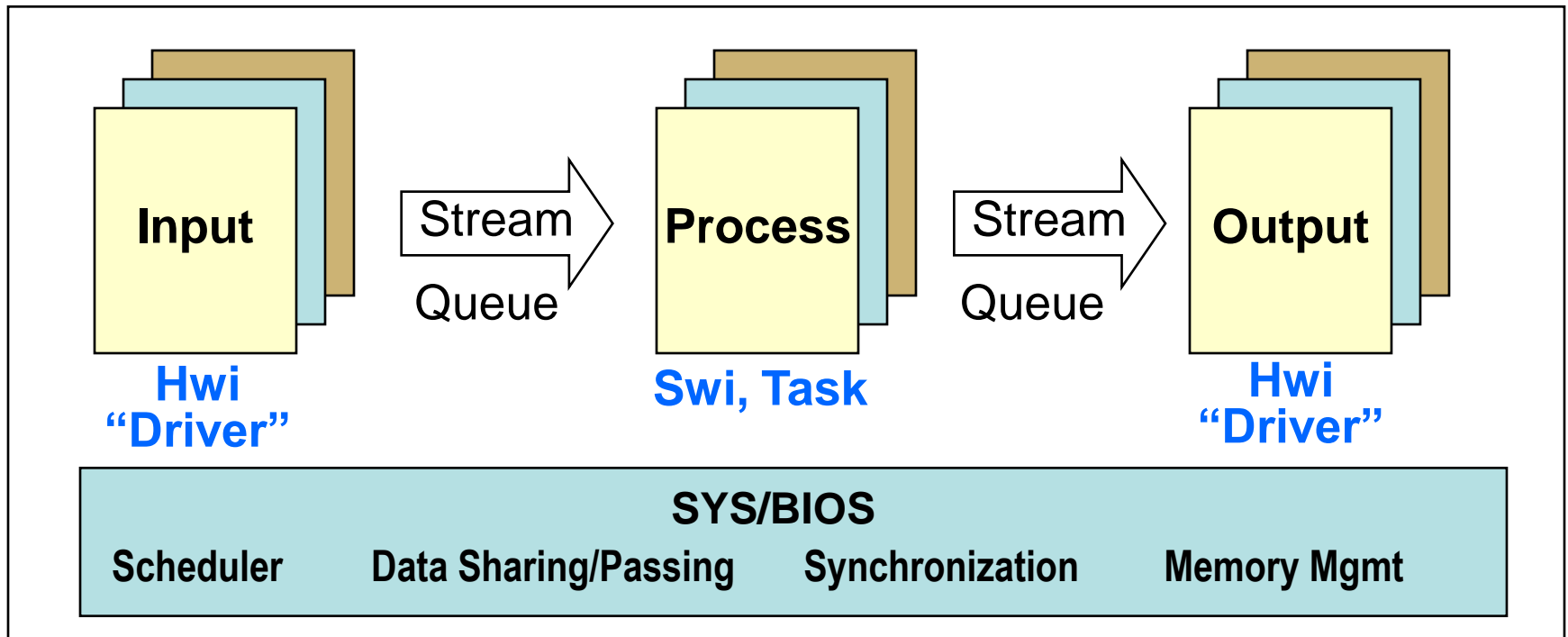


Need for an Operating System



- Simple system: single I-P-O is easy to manage
- As system complexity increases (multiple threads):
 - Can they all meet real time ?
 - Synchronization of events?
 - Priorities of threads/algos ?
 - Data sharing/passing ?
- 2 options: “home-grown” or use existing (SYS/BIOS)
(either option requires overhead)
- If you choose an existing O/S, what should you consider ?
 - Is it modular ?
 - Is it reliable?
 - Is it easy to use ?
 - Data sharing/passing ?
 - How much does it cost ?
 - What code overhead exists?

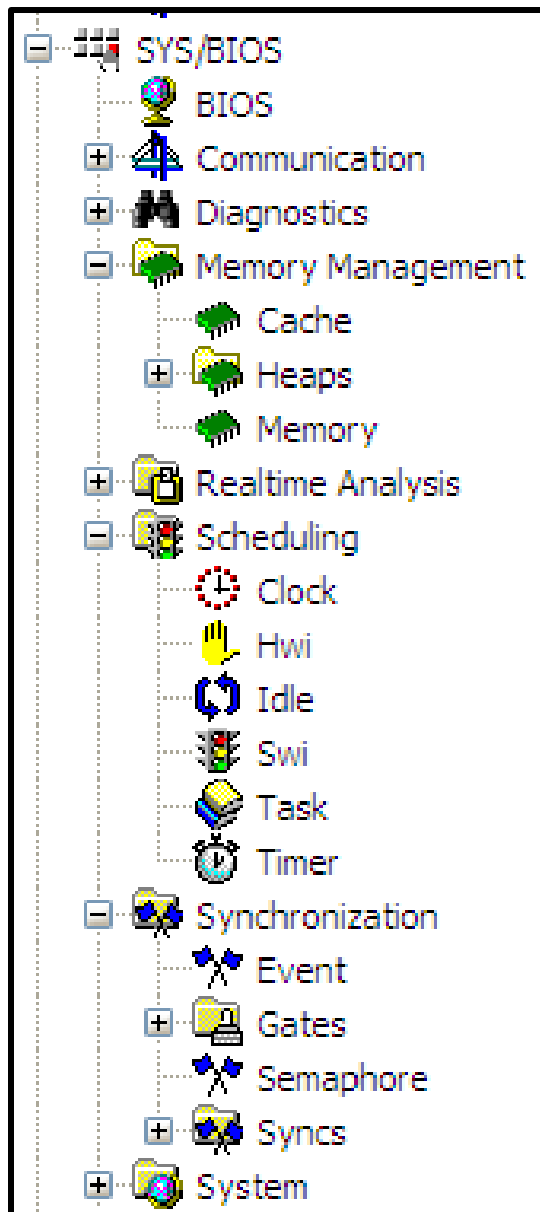
SYS/BIOS Overview



SYS/BIOS is a scalable, real-time kernel used in 1000s of systems today:

- Pre-emptive Scheduler to design system to meet real-time (including sync/priorities)
- Modular – pre-defined interface for inter-thread communications
- Reliable – 1000s of applications have used it for more than 10 years
- Footprint – deterministic, small code size, can choose which modules you desire
- Cost – free of charge

SYS/BIOS Modules & Services

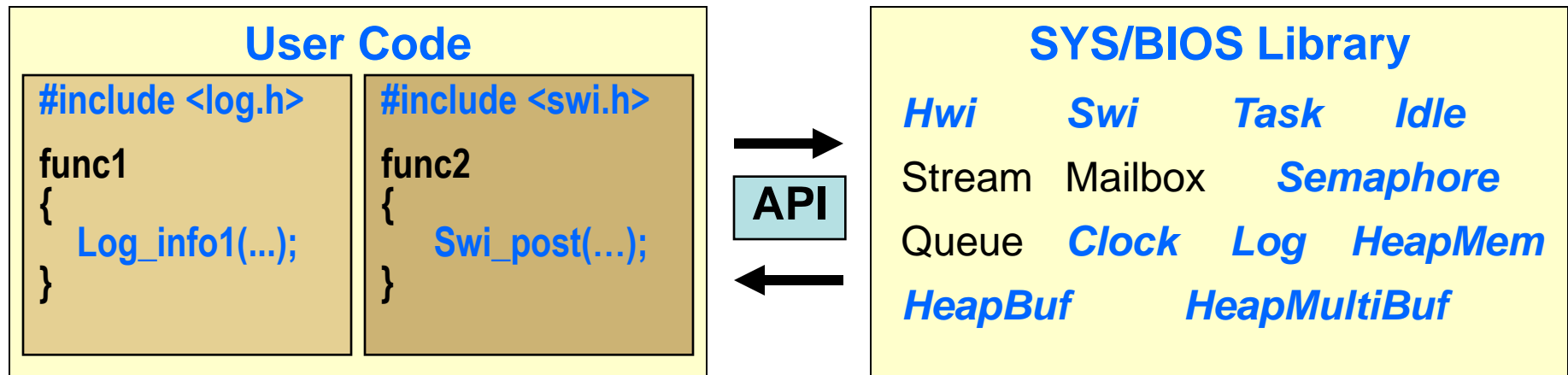


BIOS Configuration

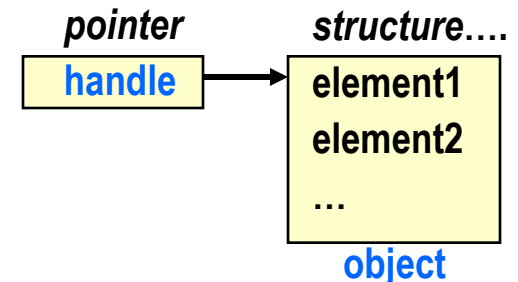
- ◆ **Memory Mgmt**
 - Cache & Heaps
- ◆ **Realtime Analysis**
 - Logs, Loads, Execution Graph
- ◆ **Scheduling**
 - All thread types
- ◆ **Synchronization**
 - Semaphores, Events, Gates

How do you interact with the SYS/BIOS services?

SYS/BIOS Environment



- ◆ SYS/BIOS is a library that contains modules with a particular interface and data structures
- ◆ **Application Program Interfaces** (API) define the interactions (methods) with a module and data structures (objects)
- ◆ **Objects** - are structures that define the state of a component
 - ◆ Pointers to objects are called **handles**
 - ◆ Object based programming offers:
 - ◆ *Better encapsulation and abstraction*
 - ◆ *Multiple instance ability*



Definitions / Vocabulary

- ◆ In this workshop, we'll be using these terms often:

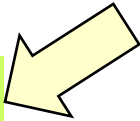
Real-time System

- *Where processing must keep up with the rate of I/O*

Function

- *Sequence of program instructions that produce a given result*

Thread



- *Function that executes within a specific context (regs, stack, PRIORITY)*

API

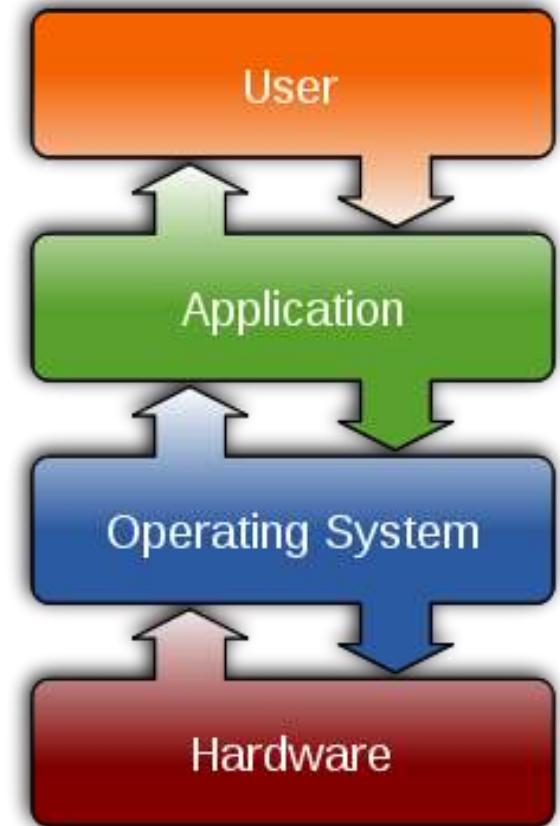
- *Application Programming Interface – “methods” for interacting with library routines and data objects*

RTOS vs GP/OS

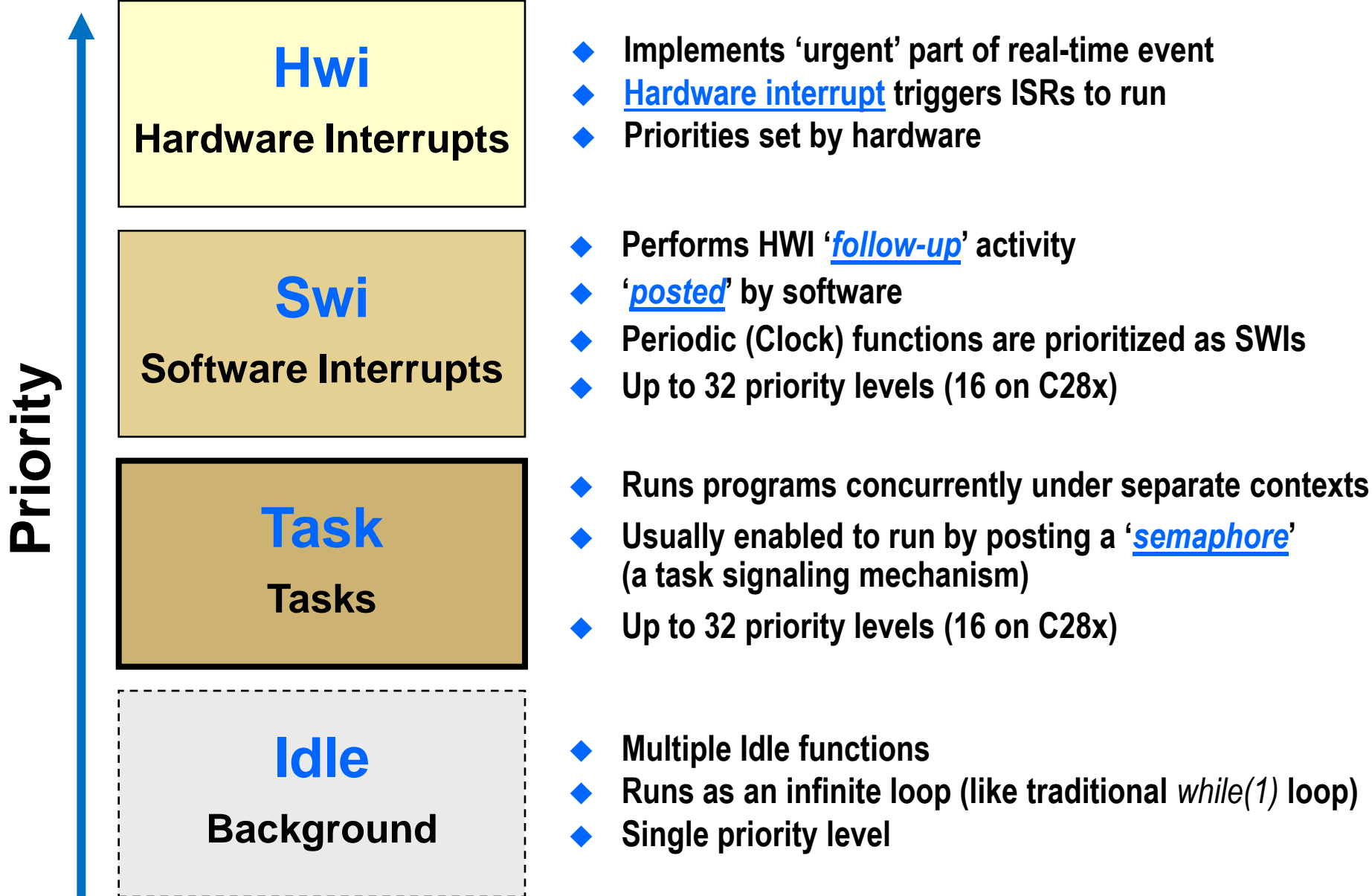
	GP/OS (e.g. Linux)	RTOS (e.g. SYS/BIOS)
Scope	General	Specific
Size	Large: 5M-50M	Small: 5K-50K
Event response	1ms to .1ms	100 – 10 ns
File management	FAT, etc	FatFS
Dynamic Memory	Yes	Yes
Threads	Processes, pThreads, Ints	Hwi, Swi, Task, Idle
Scheduler	Time Slicing	Preemption
Host Processor	ARM, x86, Power PC	ARM, MSP430, M3, C28x, DSP

Outline

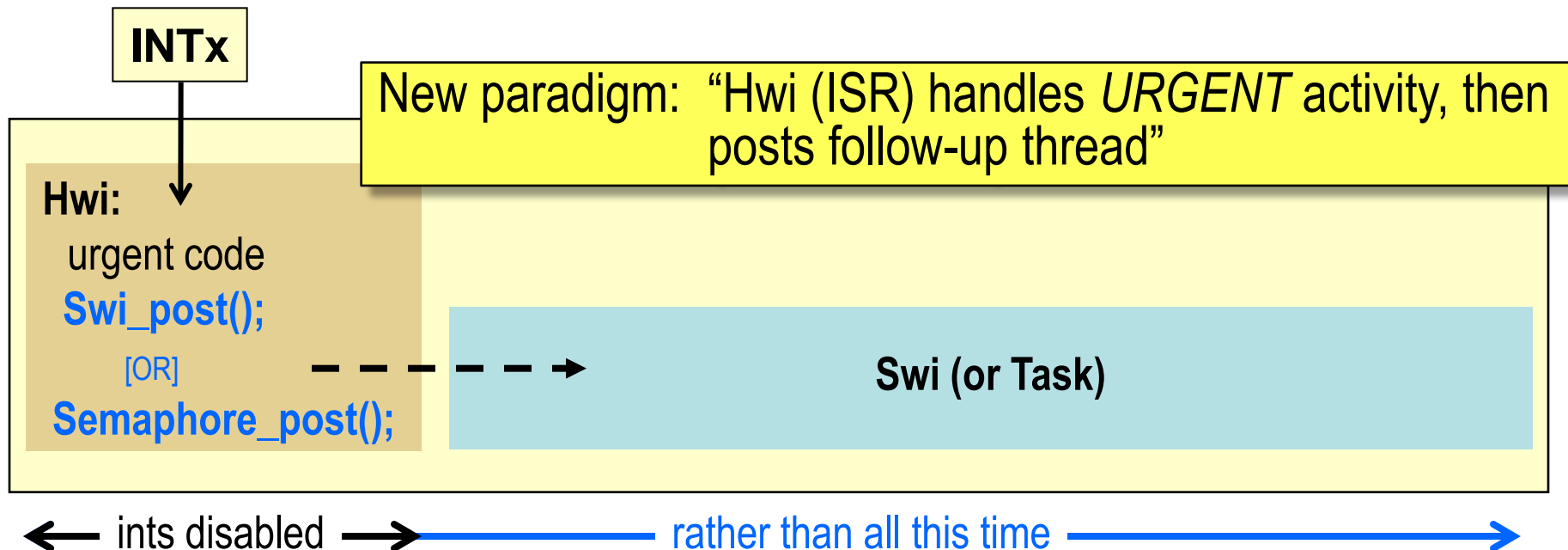
- ◆ Intro to SYS/BIOS
 - ◆ Overview
 - ◆ Threads and Scheduling
 - ◆ Creating a BIOS Thread
 - ◆ System Timeline
 - ◆ Real-Time Analysis Tools
 - ◆ Create A New Project
 - ◆ BIOS Configuration (.CFG)
 - ◆ Platforms
 - ◆ For More Info.....
- ◆ BIOS Threads



SYS/BIOS Thread Types



Hwi's Signaling Swi/Task



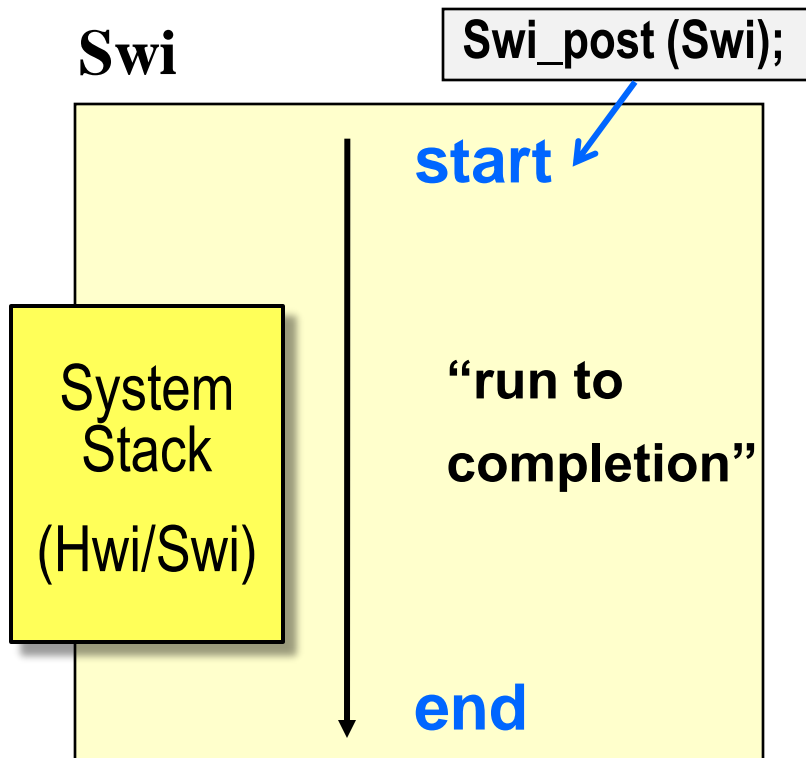
Hwi

- ♦ Fast response to interrupts
- ♦ Minimal context switching
- ♦ High priority only
- ♦ Can post Swi
- ♦ Use for urgent code only – then post follow up activity

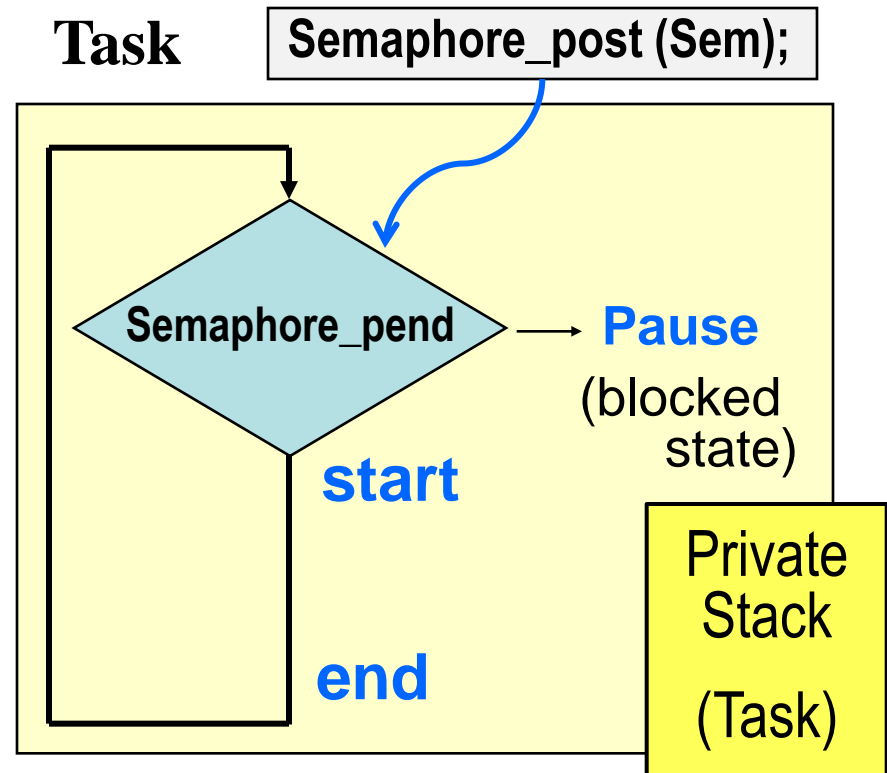
Swi

- ♦ Latency in response time
- ♦ Context switch performed
- ♦ Selectable priority levels
- ♦ Can post another Swi
- ♦ Execution managed by scheduler

Swi's and Tasks



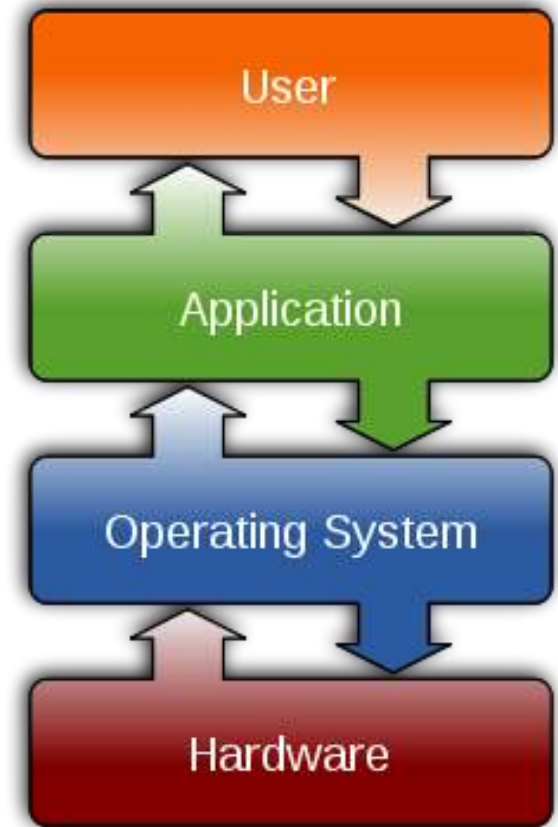
- ◆ Similar to hardware interrupt, but triggered when posted
- ◆ All Swi's share system software stack with Hwi's



- ◆ Unblocking triggers execution (also could be mailbox, events, etc.)
- ◆ Each Task has its own stack, which allows them to pause (i.e. block)
- ◆ Topology: prologue, loop, epilogue...

Outline

- ◆ Intro to SYS/BIOS
 - ◆ Overview
 - ◆ Threads and Scheduling
 - ◆ **Creating a BIOS Thread**
 - ◆ System Timeline
 - ◆ Real-Time Analysis Tools
 - ◆ Create A New Project
 - ◆ BIOS Configuration (.CFG)
 - ◆ Platforms
 - ◆ For More Info.....
- ◆ BIOS Threads



Thread (Object) Creation in BIOS

Users can create threads (BIOS resources or “objects”):

- Statically (via the GUI or .cfg script)
- Dynamically (via C code) – *more details in the “dynamic” chapter*
- BIOS doesn't care – but you might...

Dynamic (C Code)

```
#include <ti/sysbios/hal/Hwi.h>
Hwi_Params hwiParams;
Hwi_Params_init(&hwiParams);
hwiParams.eventId = 61;
Hwi_create(5, isrAudio, &hwiParams, NULL);
```

app.c

Static (GUI or Script)

Generic Hardware Interrupt Instance

Basic Advanced

Basic Settings

Name: HWI_INT5

ISR function: isrAudio

Interrupt Number: 5

Interrupt Scheduling Options

Interrupts to mask: MaskingOption_SELF

Priority: 5

Event Id: 61

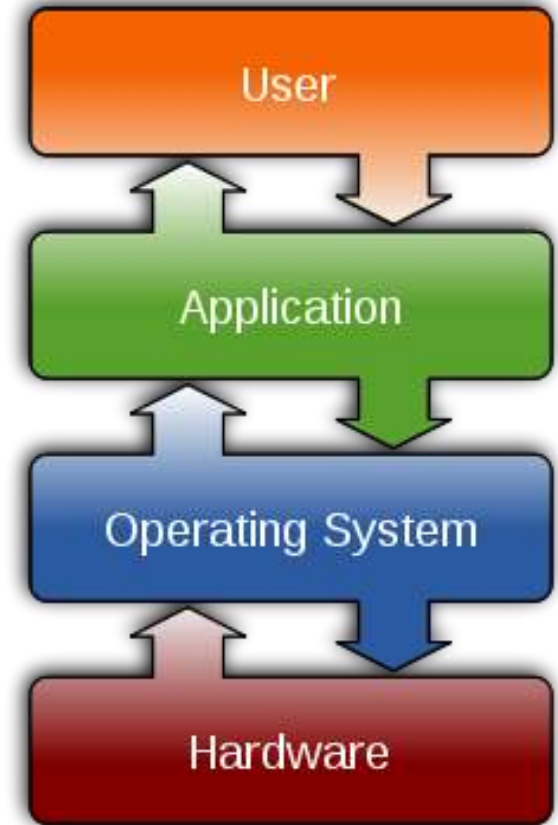
☒ Enabled at startup

```
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
var hwiParams = new Hwi.Params();
hwiParams.eventId = 61;
Hwi.create(5, "&isrAudio", hwiParams);
```

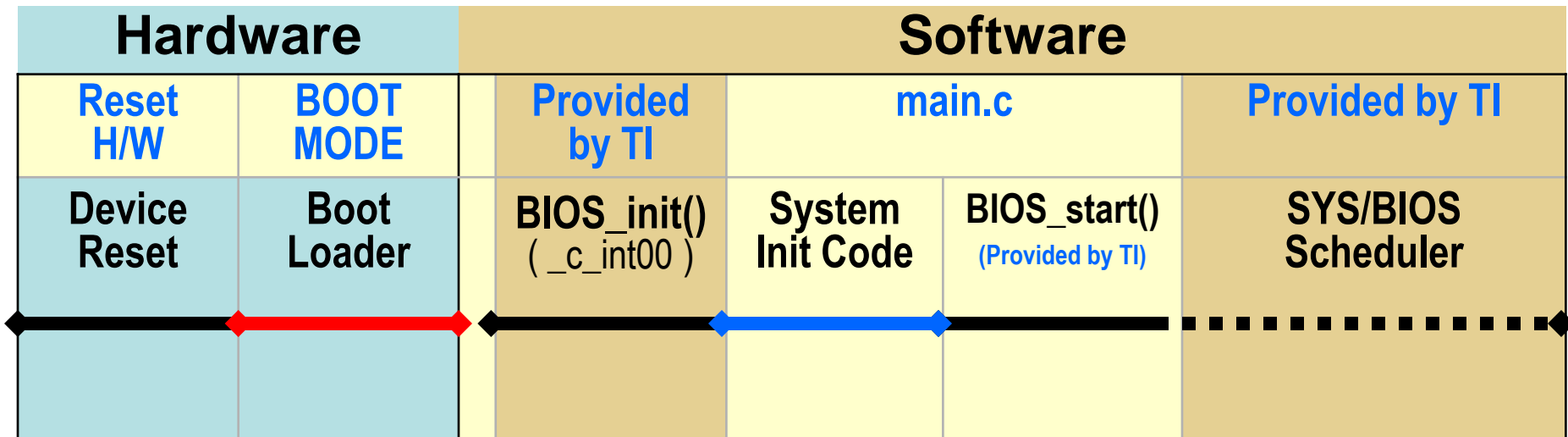
app.cfg

Outline

- ◆ Intro to SYS/BIOS
 - ◆ Overview
 - ◆ Threads and Scheduling
 - ◆ Creating a BIOS Thread
 - ◆ System Timeline
 - ◆ Real-Time Analysis Tools
 - ◆ Create A New Project
 - ◆ BIOS Configuration (.CFG)
 - ◆ Platforms
 - ◆ For More Info.....
- ◆ BIOS Threads



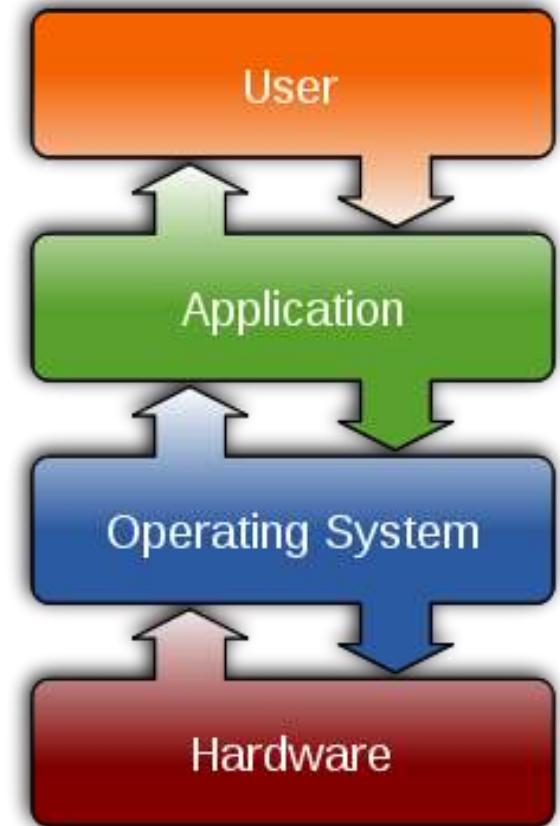
System Timeline



- ◆ **RESET** – Device is reset, then jumps to bootloader or code entry point (`_c_int00`)
- ◆ **BOOT MODE** – runs bootloader (if applicable)
- ◆ **BIOS_init()** – configs static BIOS objects, jumps to `_c_int00` to init Stack Pointer (SP), globals/statics, then calls `main()`
- ◆ **main()**
 - ◆ User initialization
 - ◆ Must execute `BIOS_start()` to enable BIOS Scheduler & INTs

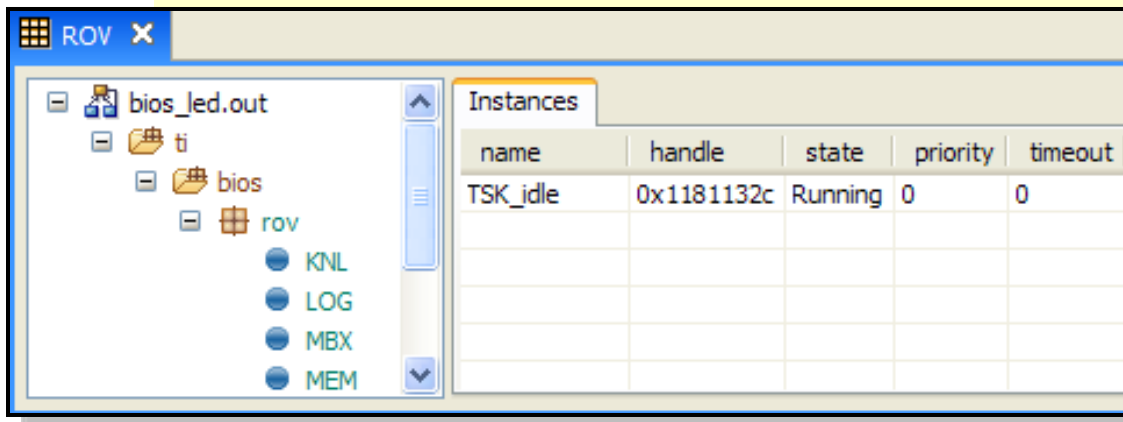
Outline

- ◆ Intro to SYS/BIOS
 - ◆ Overview
 - ◆ Threads and Scheduling
 - ◆ Creating a BIOS Thread
 - ◆ System Timeline
 - ◆ Real-Time Analysis Tools
 - ◆ Create A New Project
 - ◆ BIOS Configuration (.CFG)
 - ◆ Platforms
 - ◆ For More Info.....
- ◆ BIOS Threads



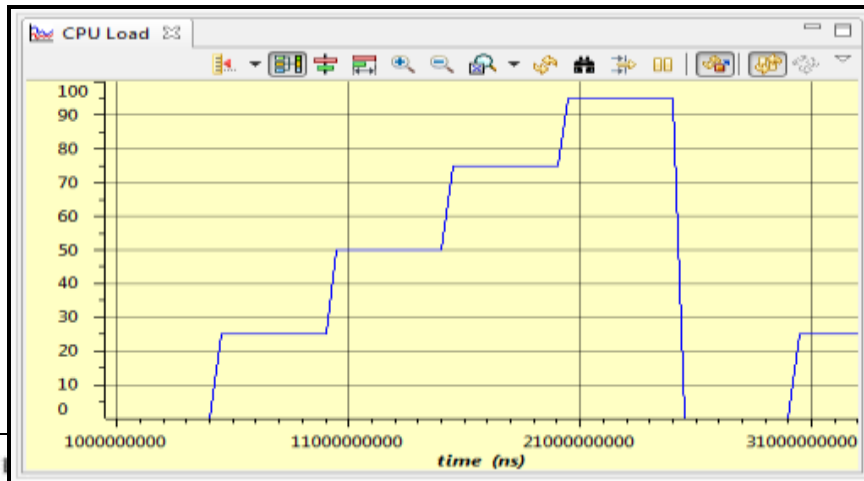
Built-in Real-Time Analysis Tools

- ◆ 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)



RunTime Obj View (ROV)

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



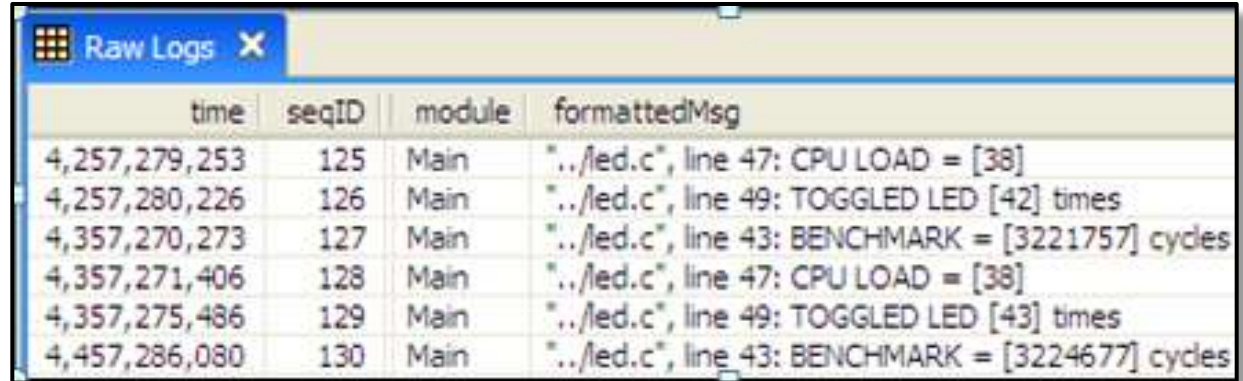
CPU/Thread Load Graph

- ◆ Analyze time NOT spent in Idle

Built-in Real-Time Analysis Tools

Logs

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

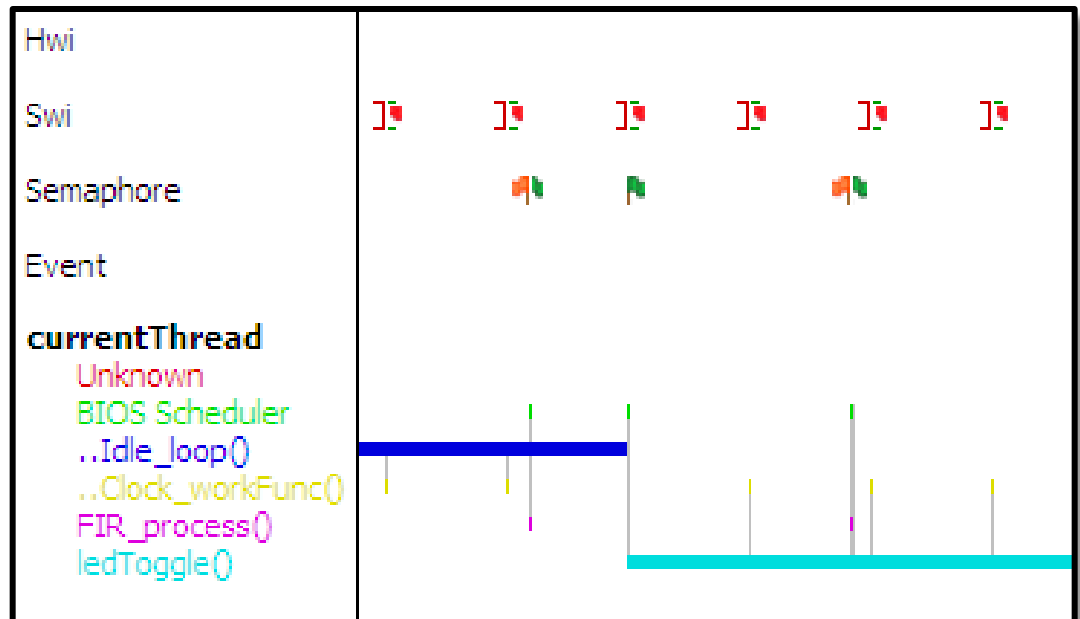


time	seqID	module	formattedMsg
4,257,279,253	125	Main	"../led.c", line 47: CPU LOAD = [38]
4,257,280,226	126	Main	"../led.c", line 49: TOGGLED LED [42] times
4,357,270,273	127	Main	"../led.c", line 43: BENCHMARK = [3221757] cycles
4,357,271,406	128	Main	"../led.c", line 47: CPU LOAD = [38]
4,357,275,486	129	Main	"../led.c", line 49: TOGGLED LED [43] times
4,457,286,080	130	Main	"../led.c", line 43: BENCHMARK = [3224677] cycles

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

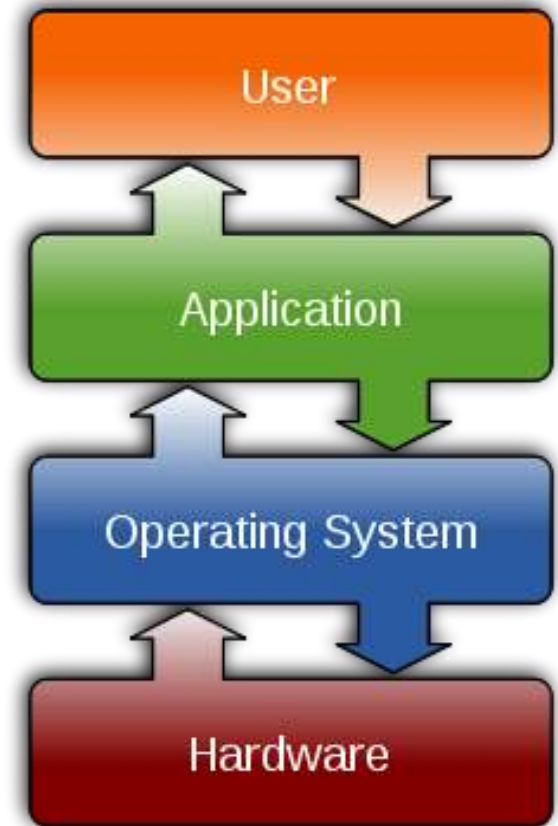
Execution Graph

- ◆ View system events down to the CPU cycle...
- ◆ Calculate benchmarks



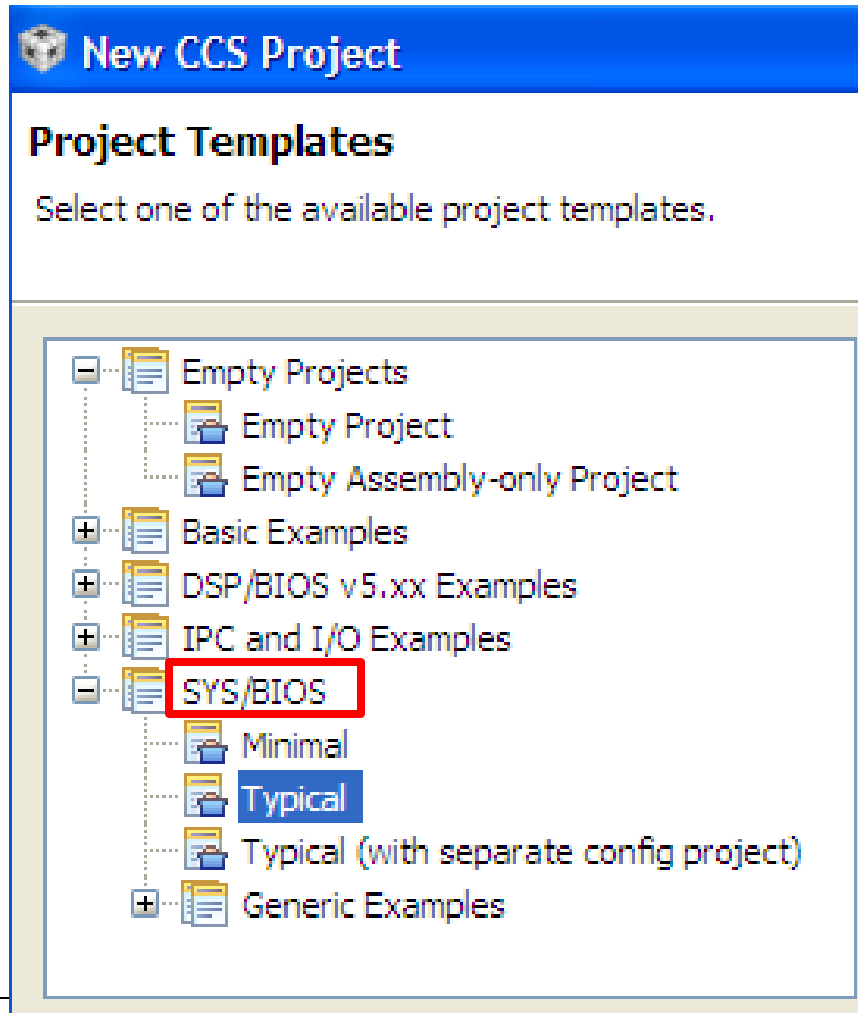
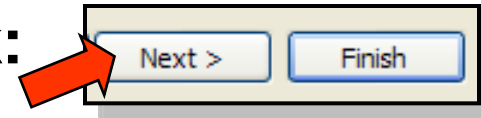
Outline

- ◆ Intro to SYS/BIOS
 - ◆ Overview
 - ◆ Threads and Scheduling
 - ◆ Creating a BIOS Thread
 - ◆ System Timeline
 - ◆ Real-Time Analysis Tools
 - ◆ **Create A New Project**
 - ◆ BIOS Configuration (.CFG)
 - ◆ Platforms
 - ◆ For More Info.....
- ◆ BIOS Threads



Building a NEW SYS/BIOS Project

- ◆ Create CCS Project (as normal), then click:
- ◆ Select a SYS/BIOS Example:

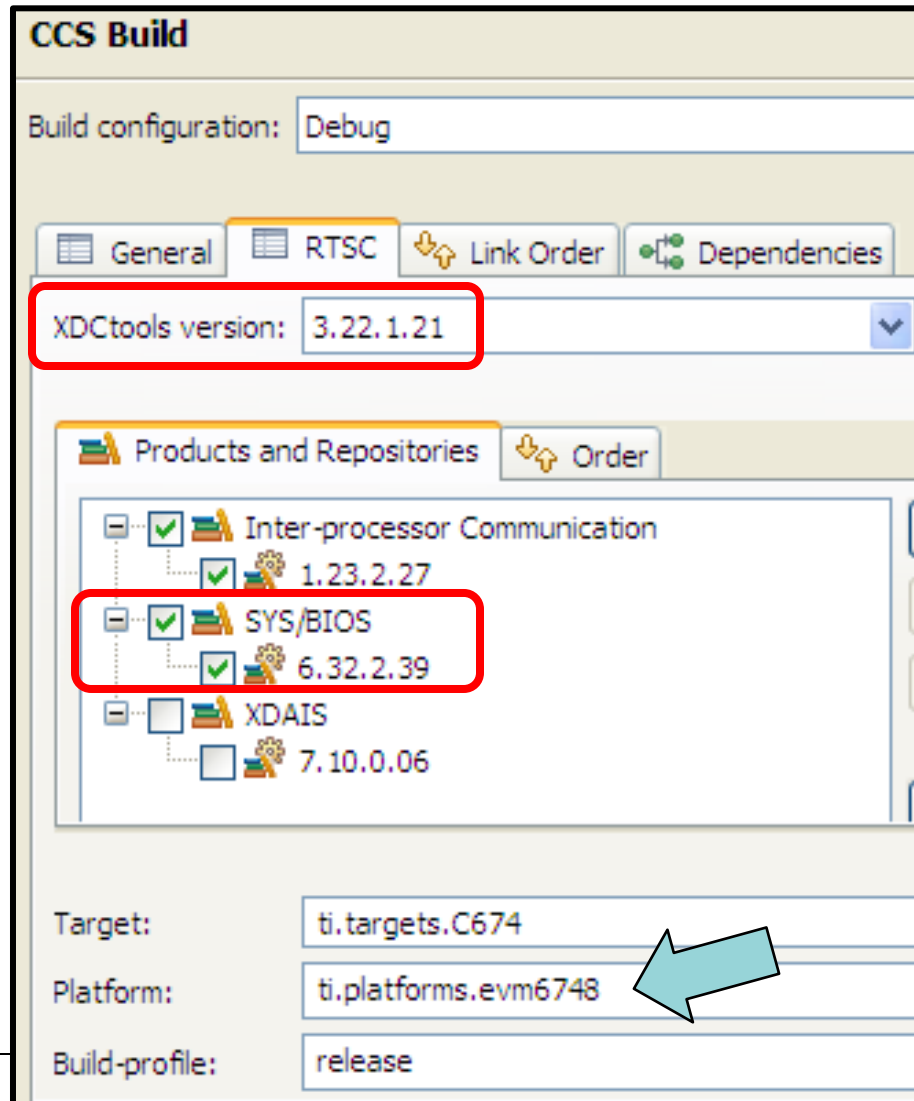


What's in the project created by "Typical"?

- Paths to SYS/BIOS tools
- .CFG file (app.cfg) that contains "typical" configuration for static objects (e.g. Swi, Task...)
- Source files (main.c) that contains appropriate #includes of header files

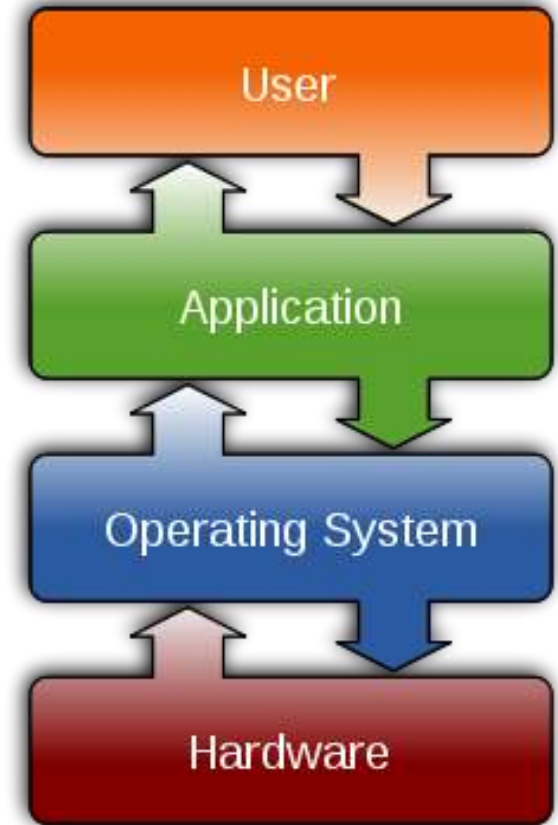
SYS/BIOS Project Settings

- ◆ Select versions for XDC, IPC, SYS/BIOS, xDAIS
- ◆ Select “Platform” file (similar to the .tcf seed file for memory)



Outline

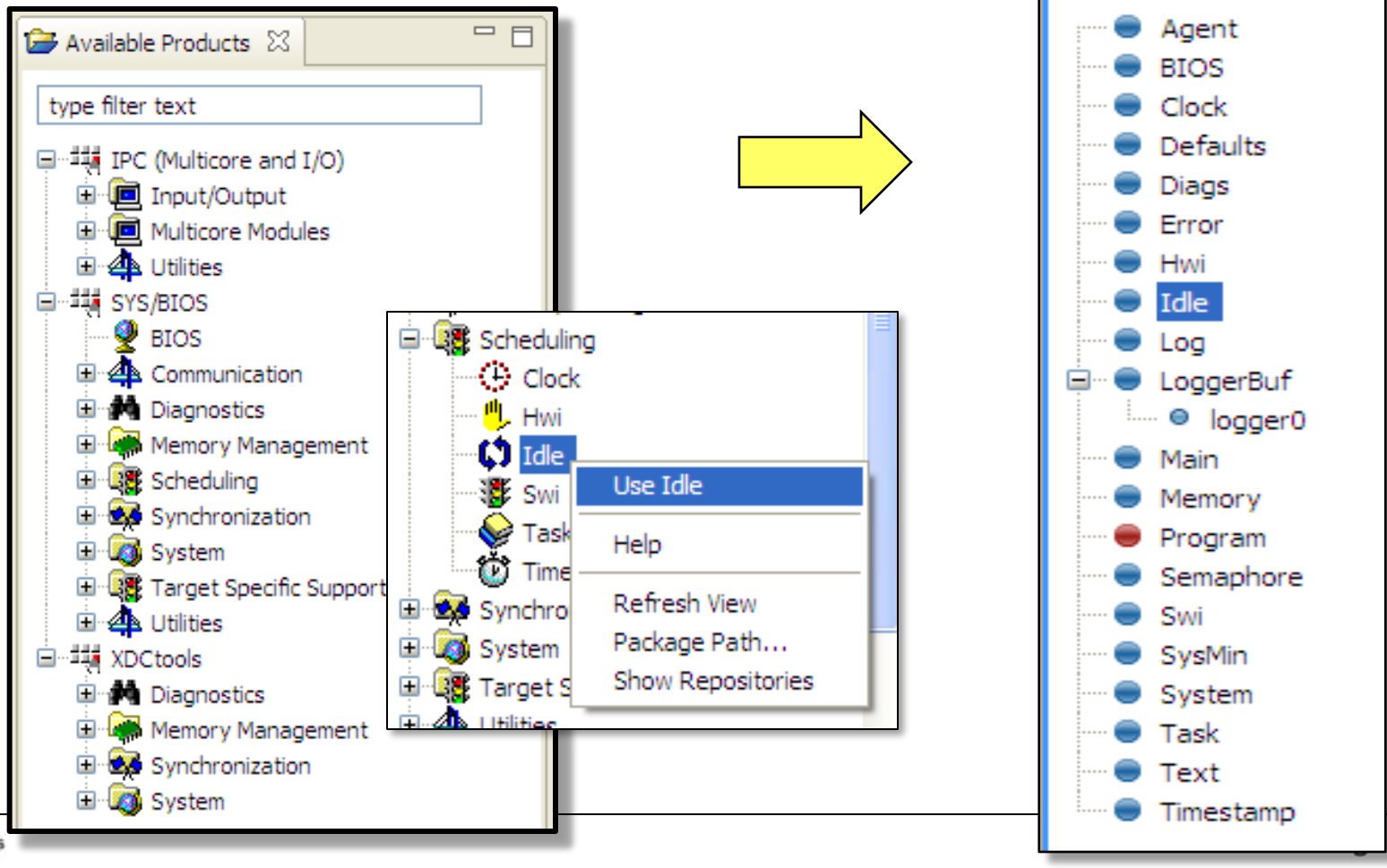
- ◆ Intro to SYS/BIOS
 - ◆ Overview
 - ◆ Threads and Scheduling
 - ◆ Creating a BIOS Thread
 - ◆ System Timeline
 - ◆ Real-Time Analysis Tools
 - ◆ Create A New Project
 - ◆ BIOS Configuration (.CFG)
 - ◆ Platforms
 - ◆ For More Info.....
- ◆ BIOS Threads



Static BIOS Configuration

◆ Users interact with the CFG file via the GUI – XGCONF:

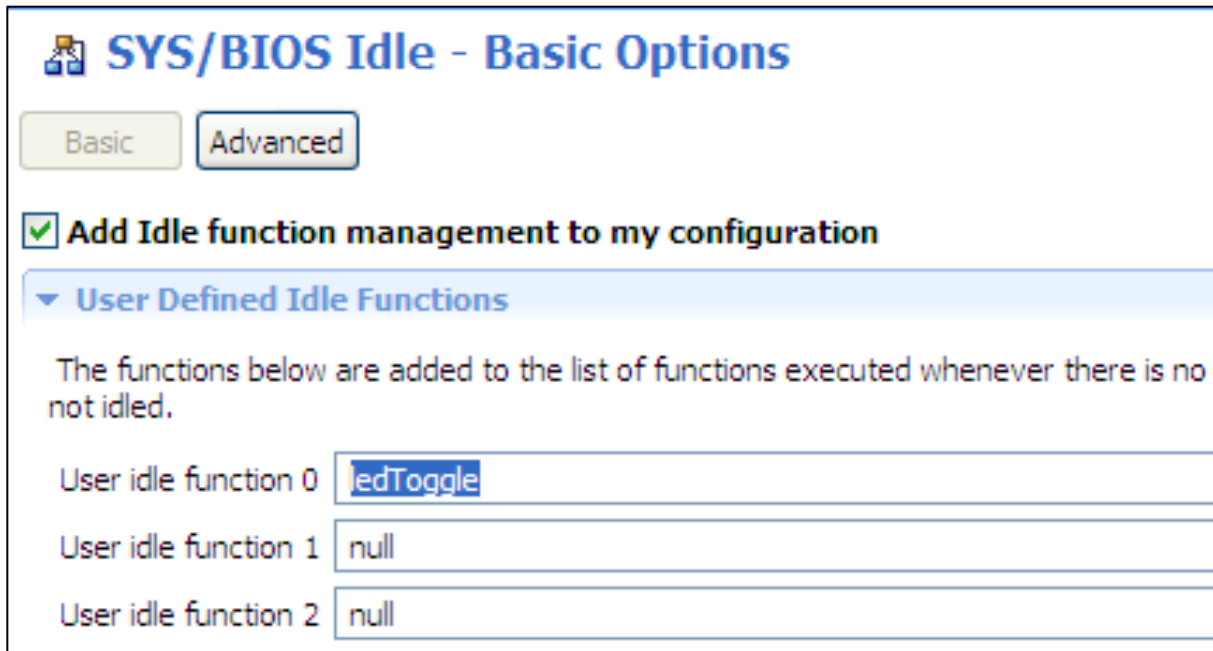
- XGCONF shows “Available Products” – Right-click and “Use Mod”
- “Mod” shows up in Outline view – Right-click and “Add New”
- All graphical changes in GUI displayed in [.cfg](#) source code



Static Config – .CFG Files

◆ Users interact with the CFG file via the GUI – XGCONF:

- When you “Add New”, you get a dialogue box to set up parameters
- Two views: “Basic” and “Advanced”



SYS/BIOS Idle - Basic Options

Basic Advanced

☒ Add Idle function management to my configuration

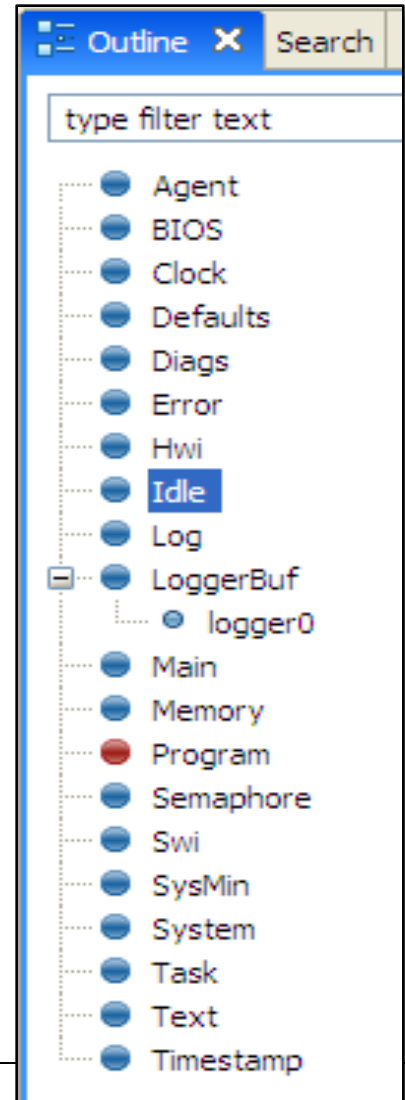
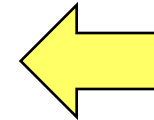
▼ User Defined Idle Functions

The functions below are added to the list of functions executed whenever there is no other function not idled.

User idle function 0

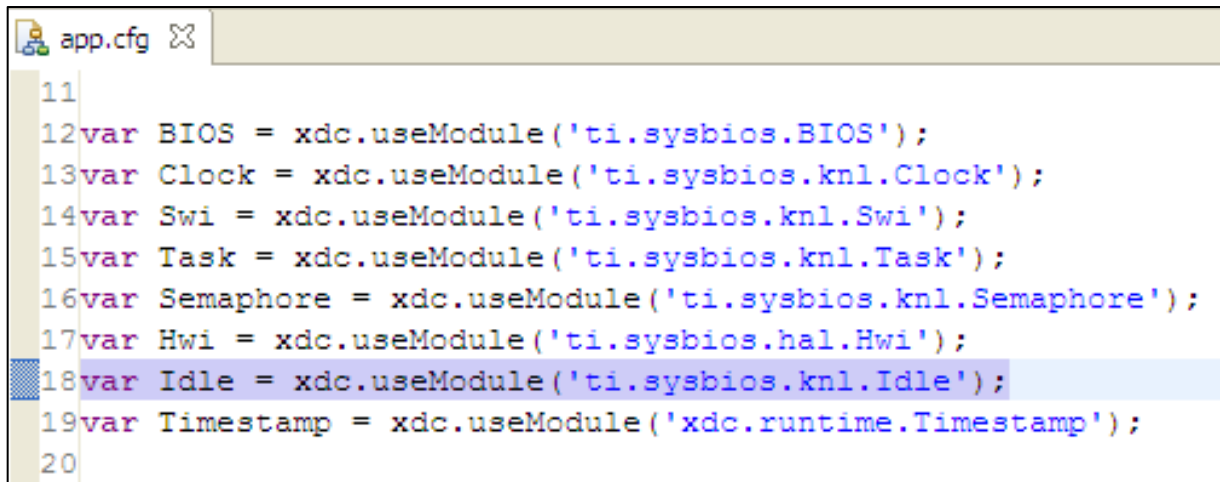
User idle function 1

User idle function 2

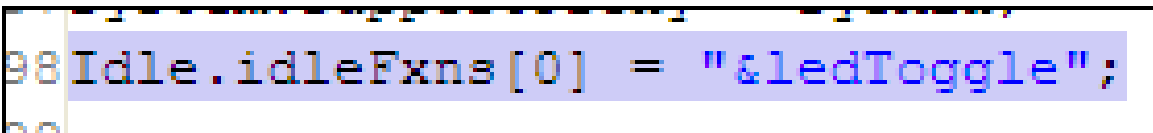


.CFG Files (XDC script)

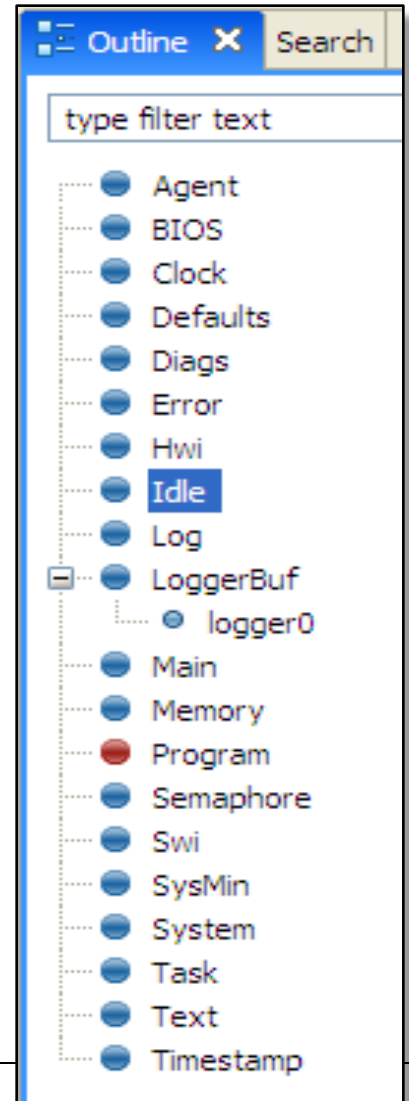
- ◆ All changes made to the GUI are reflected with java script in the .CFG file
- ◆ Click on a module on the right, see the corresponding script in app.cfg



```
11
12 var BIOS = xdc.useModule('ti.sysbios.BIOS');
13 var Clock = xdc.useModule('ti.sysbios.knl.Clock');
14 var Swi = xdc.useModule('ti.sysbios.knl.Swi');
15 var Task = xdc.useModule('ti.sysbios.knl.Task');
16 var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');
17 var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
18 var Idle = xdc.useModule('ti.sysbios.knl.Idle');
19 var Timestamp = xdc.useModule('xdc.runtime.Timestamp');
20
```

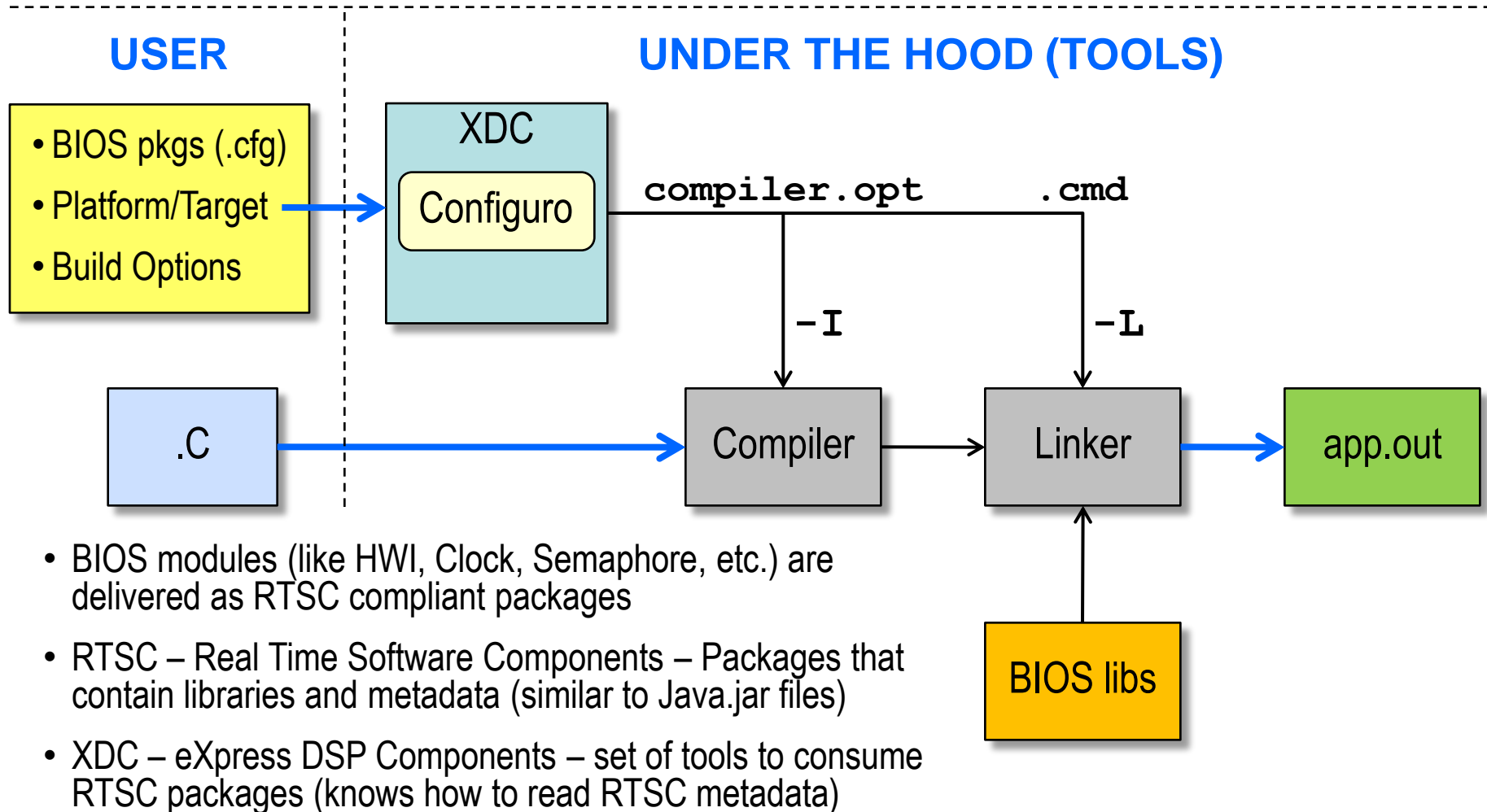


```
98 Idle.idleFxn[0] = "&ledToggle";
```



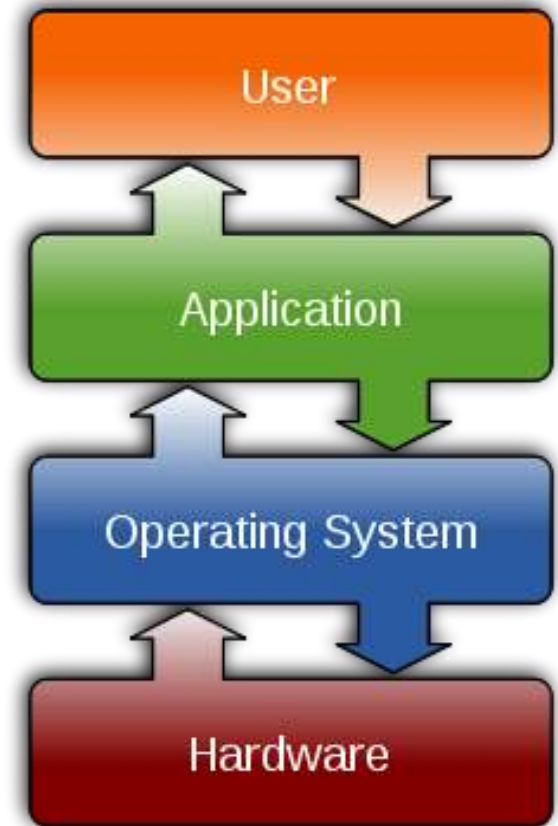
Configuration Build Flow (CFG)

- SYS/BIOS – user configures system with CFG file
- The rest is “under the hood”



Outline

- ◆ **Intro to SYS/BIOS**
 - ◆ Overview
 - ◆ Threads and Scheduling
 - ◆ Creating a BIOS Thread
 - ◆ System Timeline
 - ◆ Real-Time Analysis Tools
 - ◆ Create A New Project
 - ◆ BIOS Configuration (.CFG)
 - ◆ **Platforms**
 - ◆ For More Info.....
- ◆ **BIOS Threads**



Platform (Memory Config)

Memory Config

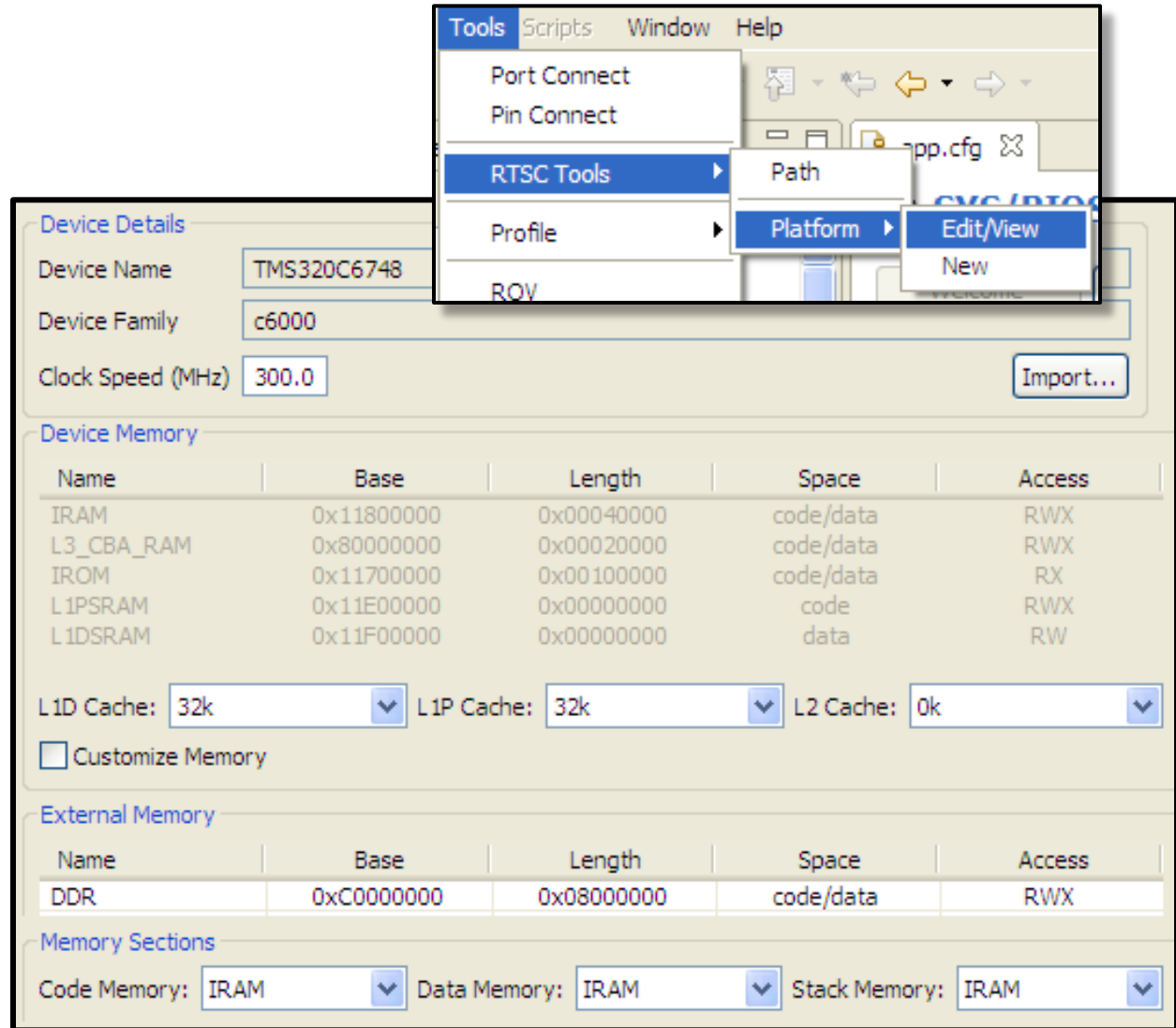
- ◆ Create Internal Memory Segments (e.g. IRAM)
- ◆ Configure cache
- ◆ Define External Memory Segments

Section Placement

- ◆ Can link code, data and stack to any defined mem segment

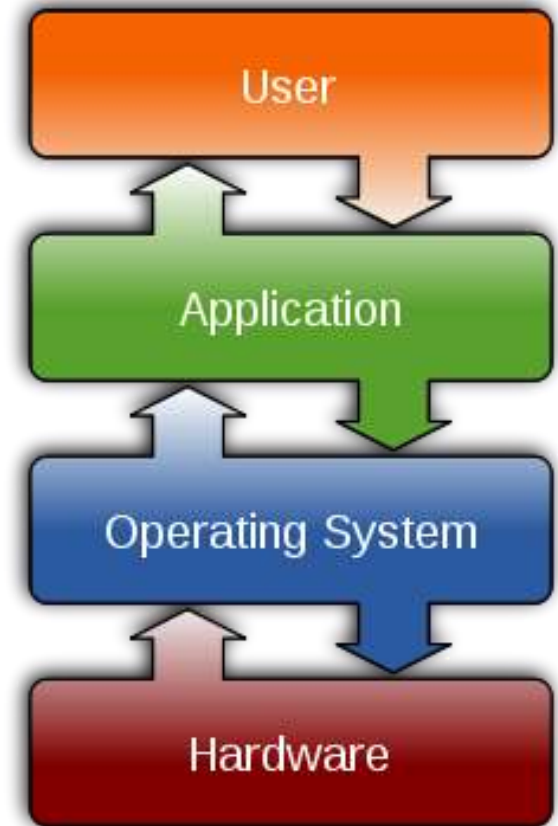
Custom Platform

- ◆ Use “Import” button to copy “seed” platform and then customize



Outline

- ◆ **Intro to SYS/BIOS**
 - ◆ Overview
 - ◆ Threads and Scheduling
 - ◆ Creating a BIOS Thread
 - ◆ System Timeline
 - ◆ Real-Time Analysis Tools
 - ◆ Create A New Project
 - ◆ BIOS Configuration (.CFG)
 - ◆ Platforms
 - ◆ For More Info.....
- ◆ **BIOS Threads**



For More Information (1)

- ◆ **SYS/BIOS Product Page** (www.ti.com/sysbios) .

SYS/BIOS Real-Time Operating System (RTOS) Status

: ACTIVE
SYSBIOS

Description/Features

Technical Documents

Support & Community

Order Now

Part Number	Texas Instruments	Status	F
SYSBIOS6: SYS/BIOS 6.x Real-Time Operating System (previously DSP/BIOS v6)	Get Software	ACTIVE	F

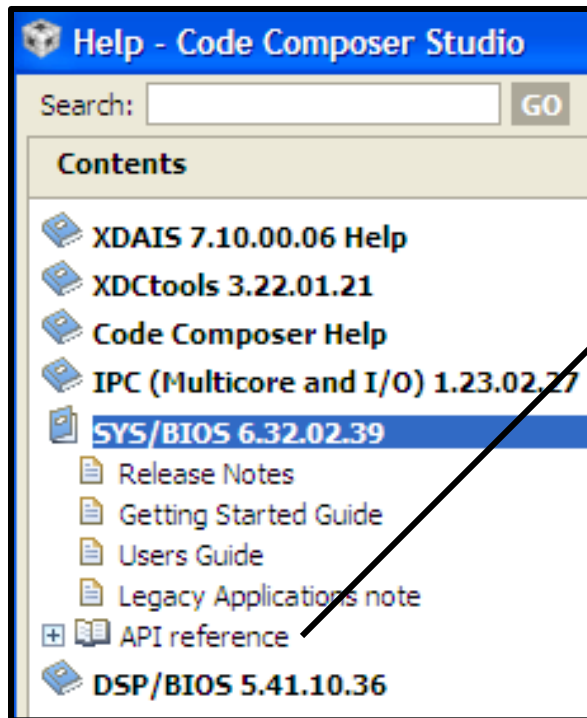
Description

Advanced RTOS Solution

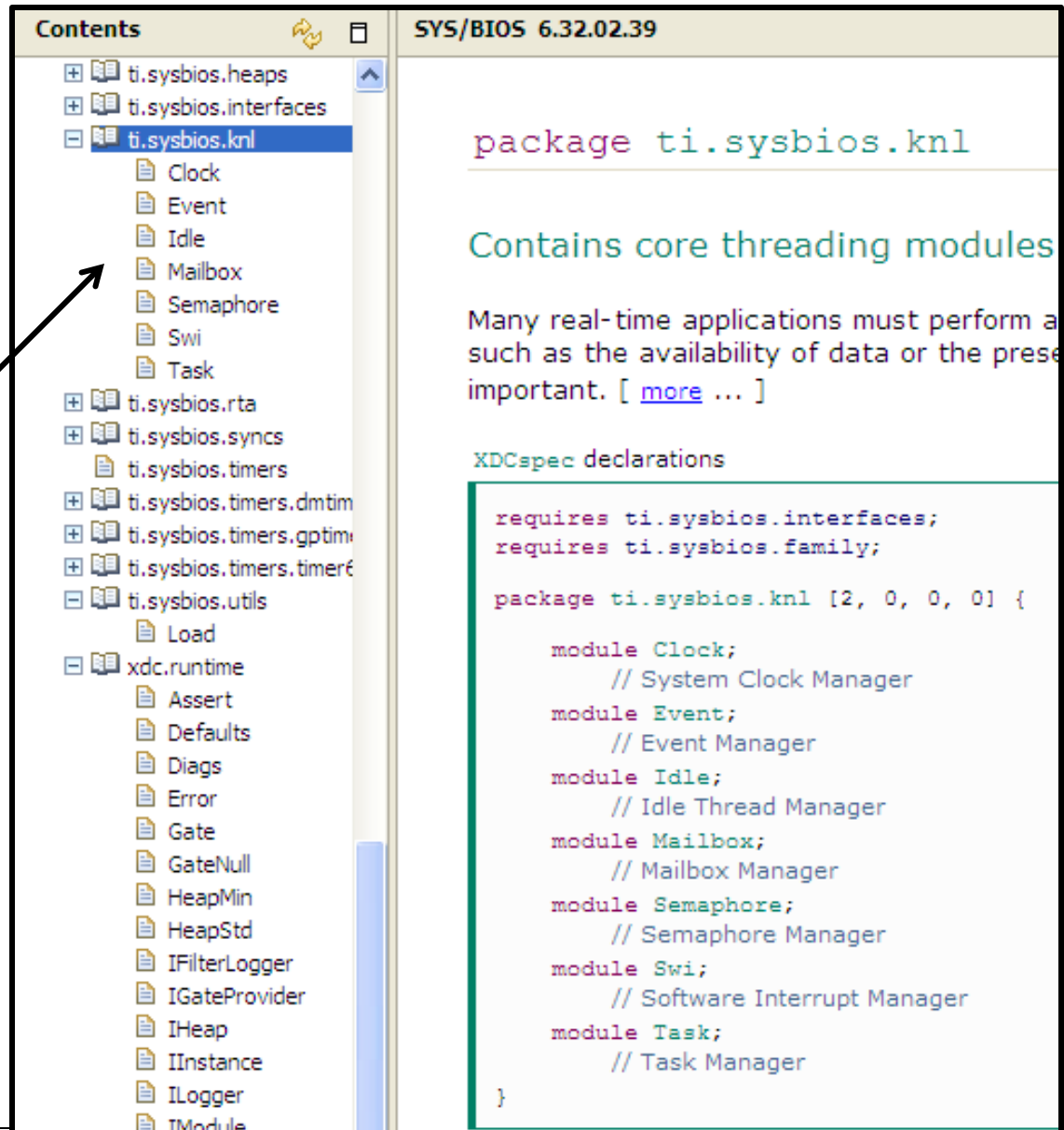
SYS/BIOS™ 6.x is an advanced, real-time operating system for use in a wide range of DSPs, ARMs, and microcontrollers. It is designed for use in embedded applications that need real-time scheduling, synchronization, and instrumentation. It provides preemptive multitasking, hardware abstraction, and memory management. Compared to its predecessor, DSP/BIOS™ 5.x, it has numerous enhancements in functionality and performance.

For More Information (2)

◆ CCS Help Contents



- User Guides
- API Reference (knl)



Download Latest Tools

◆ Download Target Content

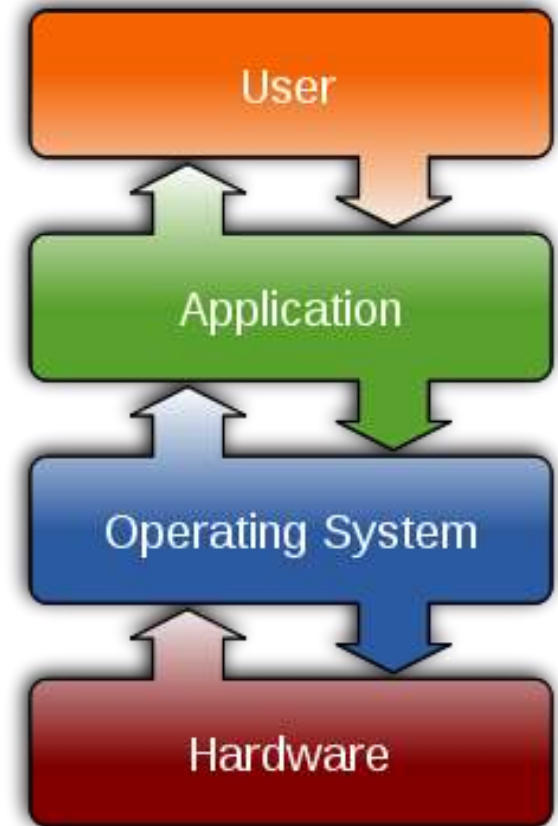
http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/

Target Content Infrastructure Product Downloads
BIOS Platform Support Packages
DSP/BIOS and SYS/BIOS
DSP/BIOS BIOSUSB Product
DSP/BIOS Utilities
Digital Video Software Development Kits (DVSDK)
DSP Link and SysLink <ul style="list-style-type: none">• SysLink (BIOS 6)• DSP Link (BIOS 5)
Graphics SDK
EDMA3 Low-level Driver
Interprocessor Communication (IPC)

- ◆ DSP/BIOS
- ◆ SYS/BIOS
- ◆ Utilities
- ◆ SysLink
- ◆ DSP Link
- ◆ IPC
- ◆ Etc.

Outline

- ◆ Intro to SYS/BIOS
- ◆ BIOS Threads
 - ◆ Hardware Interrupts (HWI)
 - ◆ Software Interrupts (SWI)
 - ◆ Tasks (TSK)
 - ◆ Semaphores (SEM)



Hwi Scheduling

*Hard
R/T*

Hwi (hi)
Hardware Interrupts

- ◆ Hwi priorities set by hardware
- ◆ Fixed number, preemption optional

Swi
Software Interrupts

- ◆ Up to 32 priority levels (16 on C28x)
- ◆ Any number possible, all preemptive

Task
Tasks

- ◆ Up to 32 priority levels (16 on C28x)
- ◆ Any number possible, all preemptive

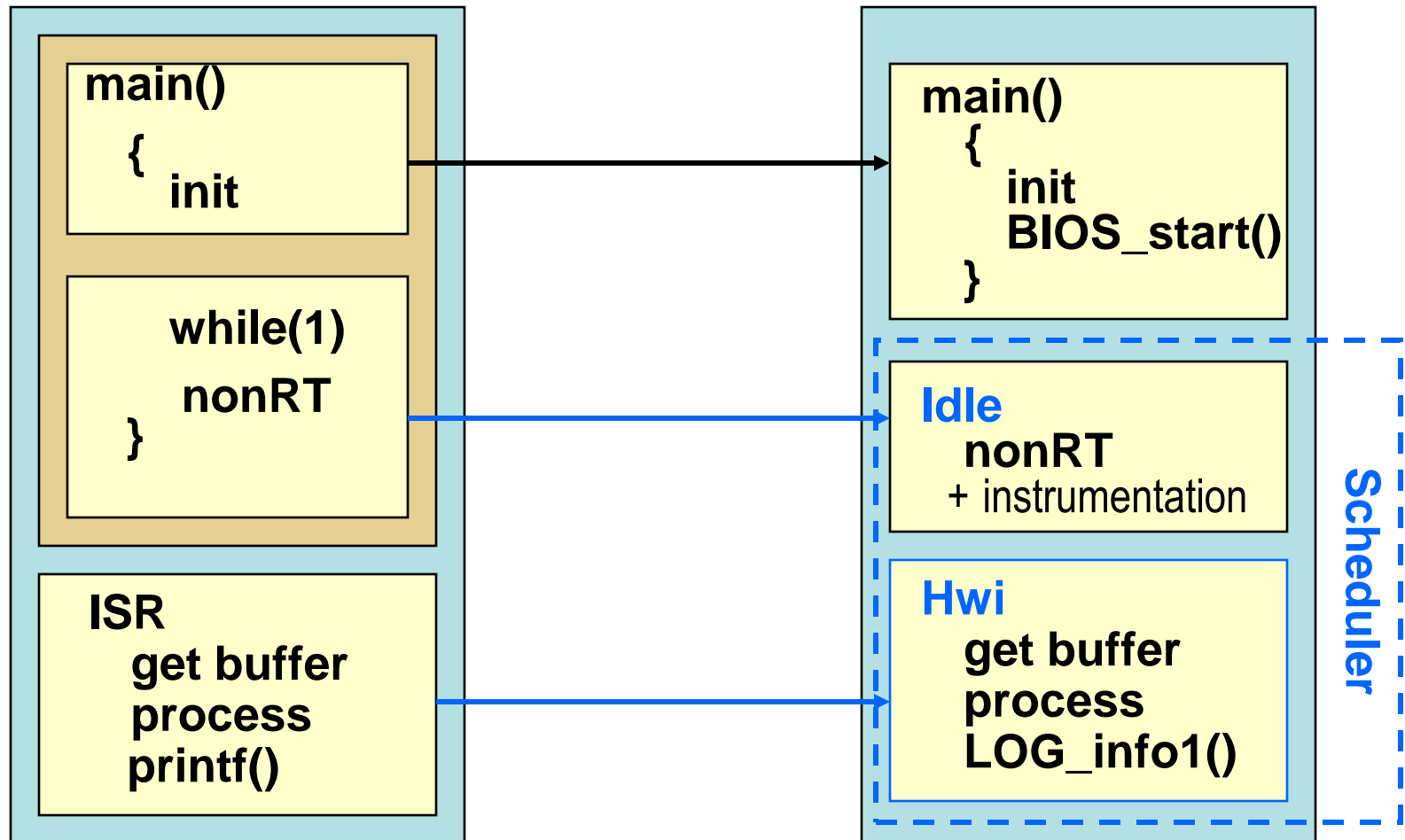
*Soft
R/T*

Idle (lo)
Background

- ◆ Continuous loop
- ◆ Non-realtime in nature

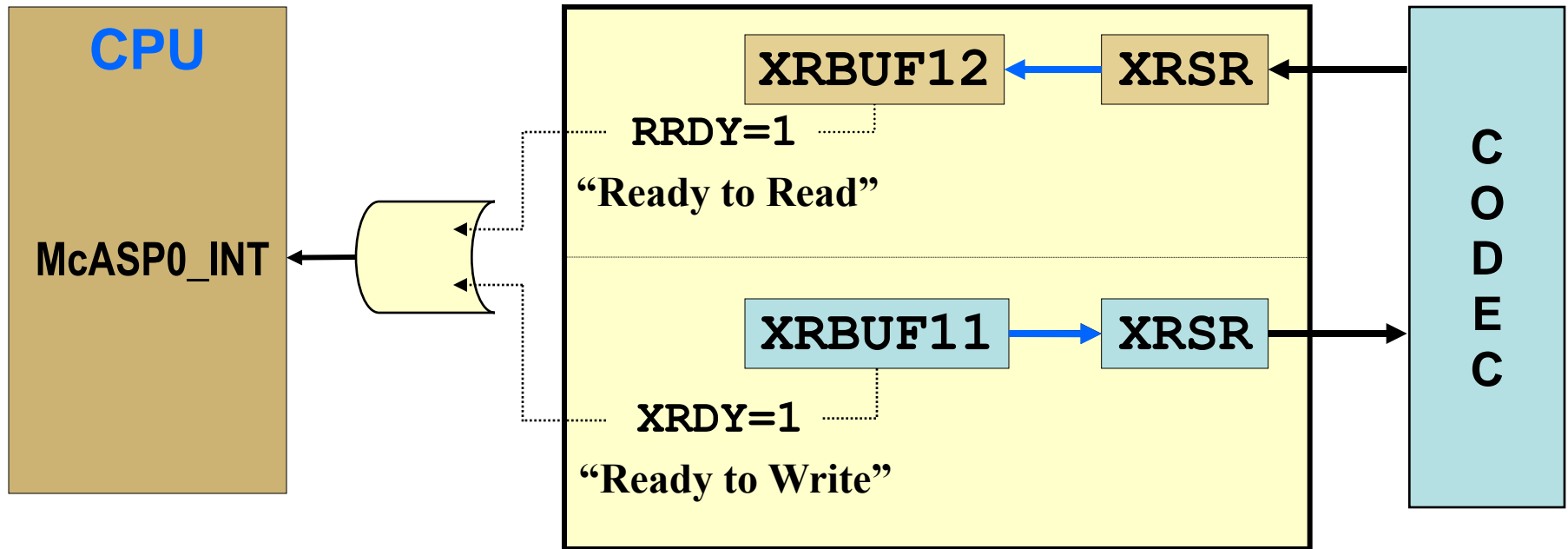
- ◆ **Idle** events run in sequence when no **Hwis** are posted
- ◆ **Hwi** is ISR with automatic vector table generation + context save/restore
- ◆ Any **Hwi** preempts **Idle**, **Hwi** may preempt other **Hwi** if desired

Foreground / Background Scheduling



- ◆ **Idle** events run in sequence when no **Hwis** are posted
- ◆ **Hwi** is ISR with automatic vector table generation + context save/restore
- ◆ Any **Hwi** preempts **Idle**, **Hwi** may preempt other **Hwi** if desired

CPU Interrupts from Peripheral (Ex: McASP)



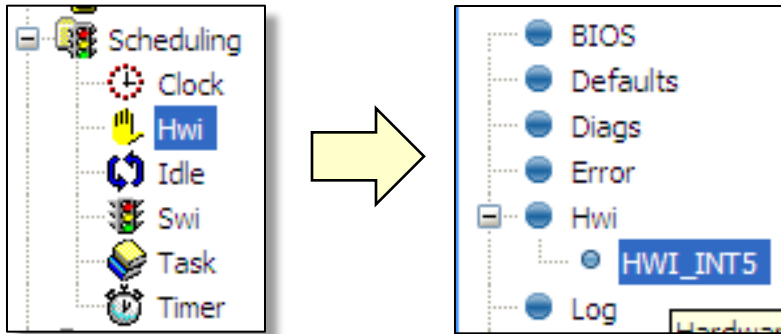
- ◆ Peripheral (e.g. McASP on C6748) causes an interrupt to the CPU to indicate “service required”.
- ◆ This “event” will have an ID (datasheet) and can be tied to a specific CPU interrupt (target specific)

How do we configure SYS/BIOS to respond to this interrupt and call the appropriate ISR?

Configuring an Hwi – Statically via GUI

Example: Tie McASP0_INT to the CPU's HWI₅

- 1 Use Hwi module (Available Products) , insert new Hwi (Outline View)



Remember, BIOS objects can be created via the GUI, script code or C code (dynamic)

- 2 Configure Hwi – Event ID, CPU Int #, ISR vector:

The screenshot shows the 'Generic Hardware Interrupt Instance' configuration window. The 'Basic' tab is selected. Under 'Basic Settings', the 'Name' is 'HWI_INT5', the 'ISR function' is 'isrAudio', and the 'Interrupt Number' is '5'. Under 'Interrupt Scheduling Options', the 'Interrupts to mask' is 'MaskingOption_SELF', the 'Priority' is '5', the 'Event Id' is '61', and the 'Enabled at startup' checkbox is checked.

To enable INT at startup, check the box

Where do you find the Event Id #?

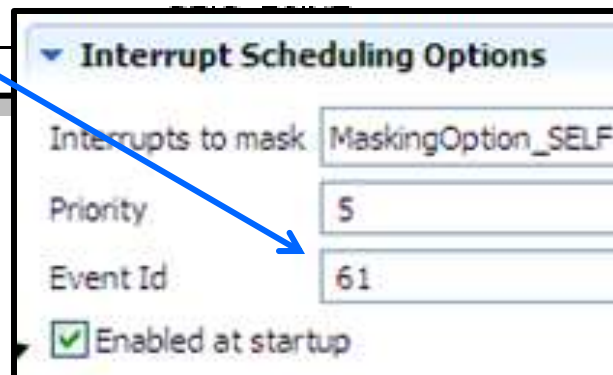
Hardware Event IDs

- ◆ So, how do you know the names of the interrupt events and their corresponding event numbers?

Look it up (in the datasheet), of course...

Ref: TMS320C6748 datasheet (exerpt):

59	GPIO_B5INT	GPIO Bank 5 Interrupt
60	DDR2_MEMERR	DDR2 Memory Error Interrupt
61	MCASP0_INT	McASP0 Combined RX/TX Interrupts
62		GPIO Bank 6 Interrupt
63		RTC Combined



▼ **Interrupt Scheduling Options**

Interrupts to mask: MaskingOption_SELF

Priority: 5

Event Id: 61

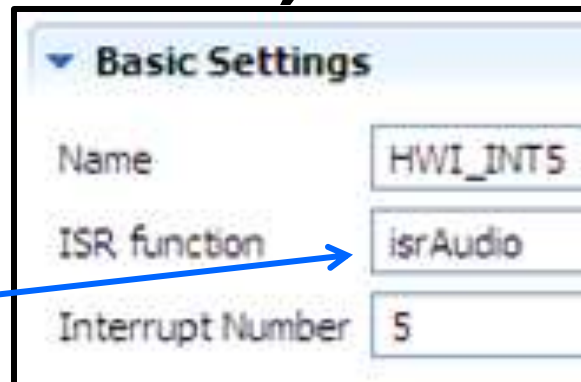
☒ Enabled at startup

- ◆ This example is target-specific for the C6748 DSP. Simply refer to your target's datasheet for similar info.

What happens in the ISR ?

Example ISR (McASP)

Example ISR for MCASP0_INT interrupt



Basic Settings	
Name	HWI_INT5
ISR function	isrAudio
Interrupt Number	5

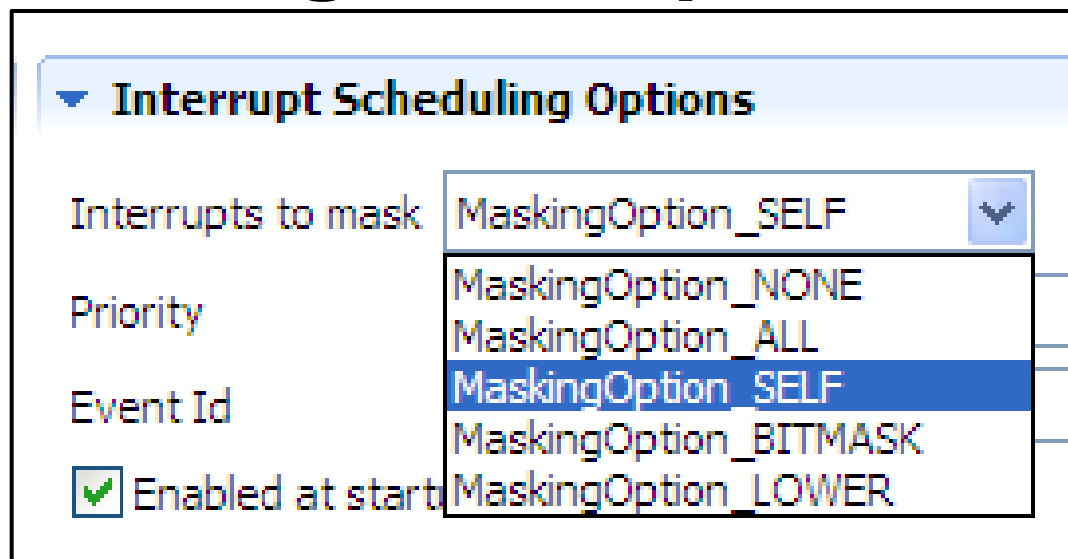
isrAudio:

```
pInBuf[blkCnt] = MCASP1->RCV;    // READ audio sample from McASP
MCASP->XMT = pOutBuf[blkCnt]    // WRITE audio sample to McASP
blkCnt+=1;                      // increment blk counter

if( blkCnt >= BUFFSIZE )
{
    memcpy(pOut, pIn, Len);      // Copy pIn to pOut (Algo)
    blkCnt = 0;                  // reset blkCnt for new buf's
    pingPong ^= 1;               // PING/PONG buffer boolean
}
```

Can one interrupt preempt another?

Enabling Preemption of Hwi



- ◆ **Default** mask is “SELF” – which means all other Hwi’s can pre-empt except for itself
- ◆ Can choose other masking options as required:

ALL:	Best choice if ISR is short & fast
NONE:	Dangerous – make sure ISR code is re-entrant
BITMASK:	Allows custom mask
LOWER:	Masks any interrupt(s) with lower priority (ARM)

SYS/BIOS Hwi APIs

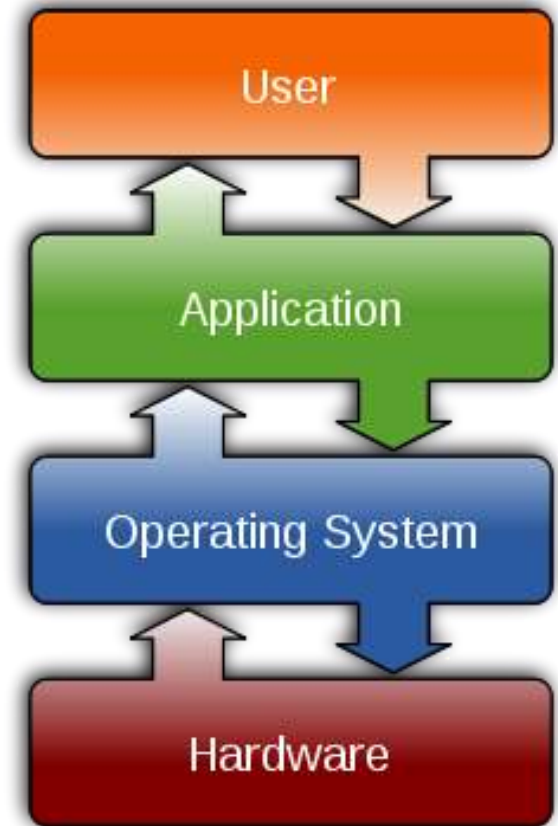
Other useful Hwi APIs:

<code>Hwi_disableInterrupt()</code> <code>Hwi_enableInterrupt()</code> <code>Hwi_clearInterrupt()</code>	Set enable bit = 0 Set enable bit = 1 Clear INT flag bit = 0
<code>Hwi_post()</code> New in SYS/BIOS	Post INT # (in code)
<code>Hwi_disable()</code> <code>Hwi_enable()</code> <code>Hwi_restore()</code>	Global INTs disable Global INTs enable Global INTs restore

Let's move on to SWIs...

Outline

- ◆ Intro to SYS/BIOS
- ◆ BIOS Threads
 - ◆ Hardware Interrupts (HWI)
 - ◆ Software Interrupts (SWI)
 - ◆ Tasks (TSK)
 - ◆ Semaphores (SEM)



Swi Scheduling

Hard
R/T

Hwi (hi)

Hardware Interrupts

- ◆ Hwi priorities set by hardware
- ◆ Fixed number, preemption optional

Swi

Software Interrupts

- ◆ Up to 32 priority levels (16 on C28x)
- ◆ Any number possible, all preemptive

Task

Tasks

- ◆ Up to 32 priority levels (16 on C28x)
- ◆ Any number possible, all preemptive

Soft
R/T

Idle (lo)

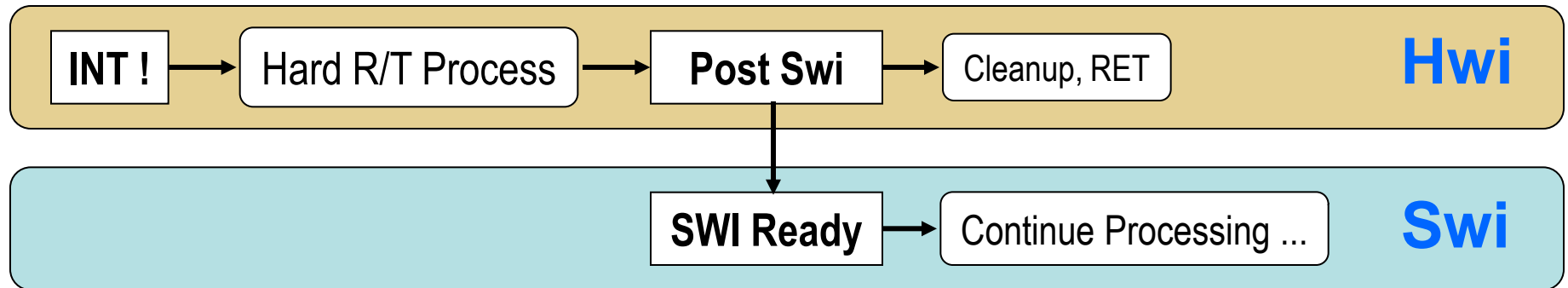
Background

- ◆ Continuous loop
- ◆ Non-realtime in nature

- ◆ SYS/BIOS provides for Hwi and Swi management
- ◆ SYS/BIOS allows the Hwi to post a Swi to the ready queue

Hardware and Software Interrupt System

Execution flow for flexible real-time systems:



Hwi

- ◆ Fast response to INTs
- ◆ Min context switching
- ◆ High priority for CPU
- ◆ Limited # of Hwi possible

isrAudio:

```
*buf++ = *XBUF;  
cnt++;  
if (cnt >= BLKSZ) {  
    Swi_post(swiFir);  
    count = 0;  
    pingPong ^= 1;  
}
```

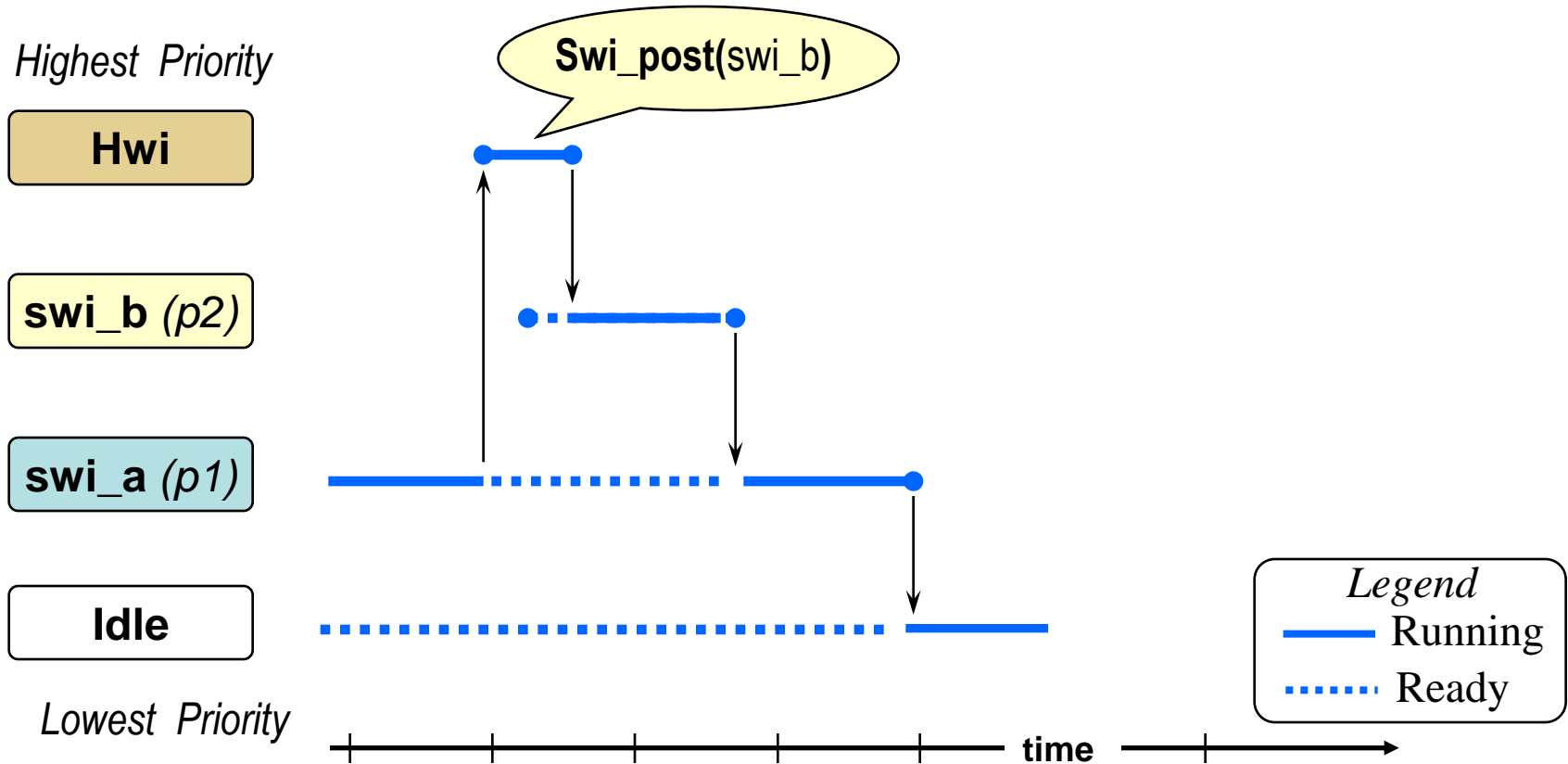
Swi

- ◆ Latency in response time
- ◆ Context switch
- ◆ Selectable priority levels
- ◆ Scheduler manages execution

- ◆ SYS/BIOS provides for Hwi and Swi management
- ◆ SYS/BIOS allows the Hwi to post a Swi to the ready queue

[Scheduling SWIs...](#)

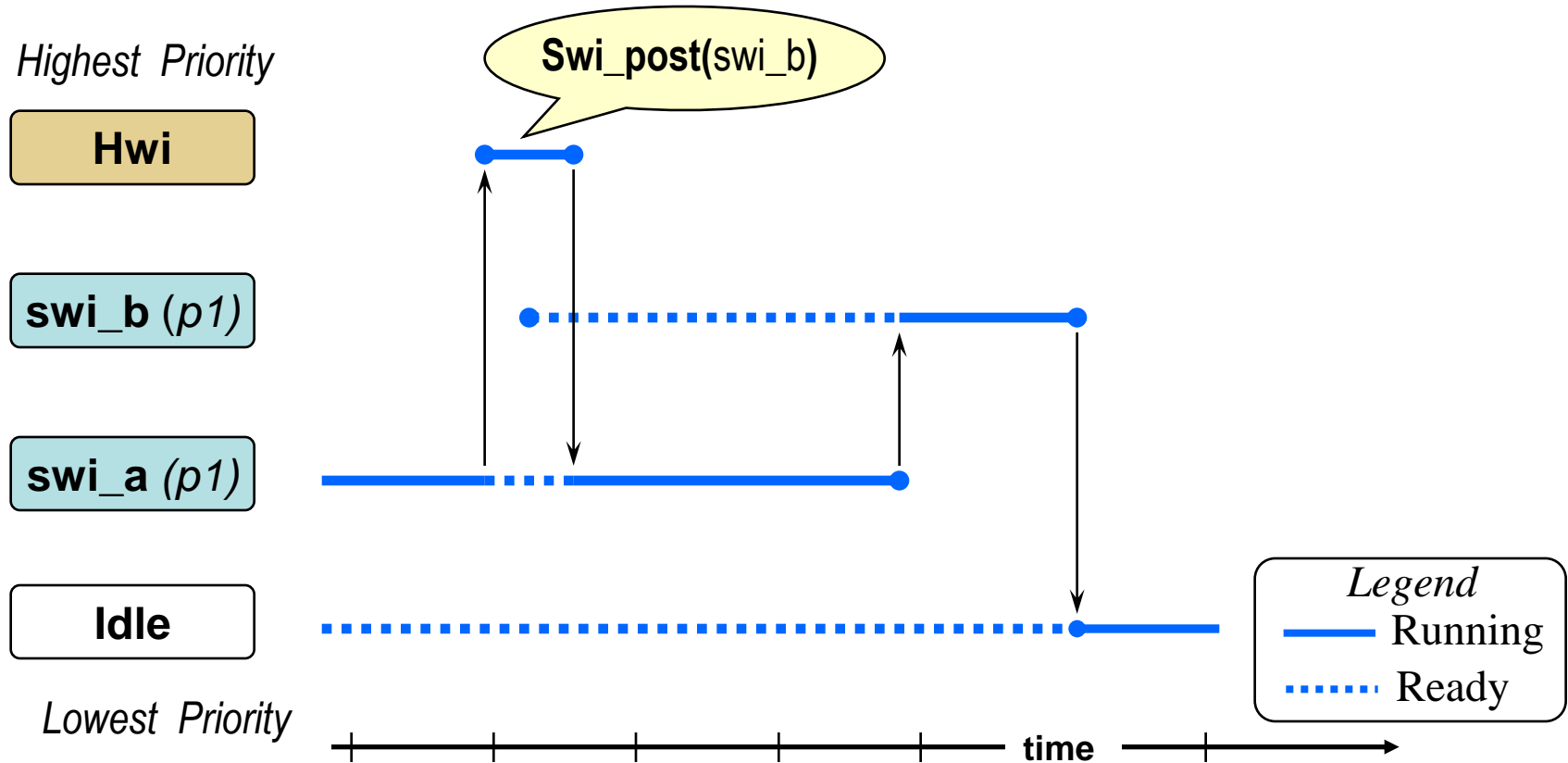
Scheduling Rules



- ◆ **Swi_post(mySwi)** : Unconditionally post a software interrupt (in the ready state)
- ◆ If a higher priority thread becomes ready, the running thread is preempted
- ◆ **Swi** priorities from 1 to 32 (C28x has 16)
- ◆ Automatic context switch (uses system stack)

What if the SWIs are at the same priority?

Scheduling Rules



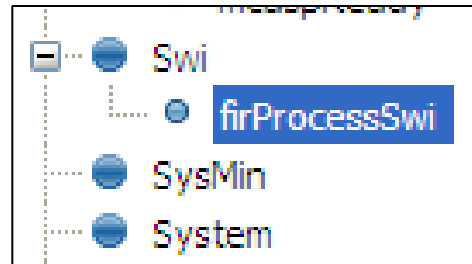
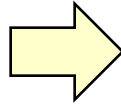
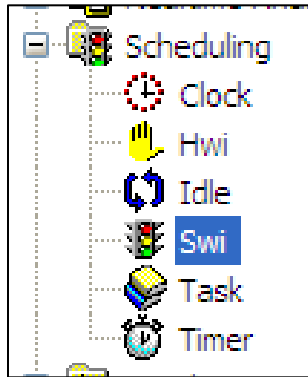
- ◆ Processes of same priority are scheduled first-in first-out (FIFO)
- ◆ Having threads at the SAME priority offers certain advantages – such as resource sharing (without conflicts)

[How do you configure a SWI?](#)

Configuring a Swi – Statically via GUI

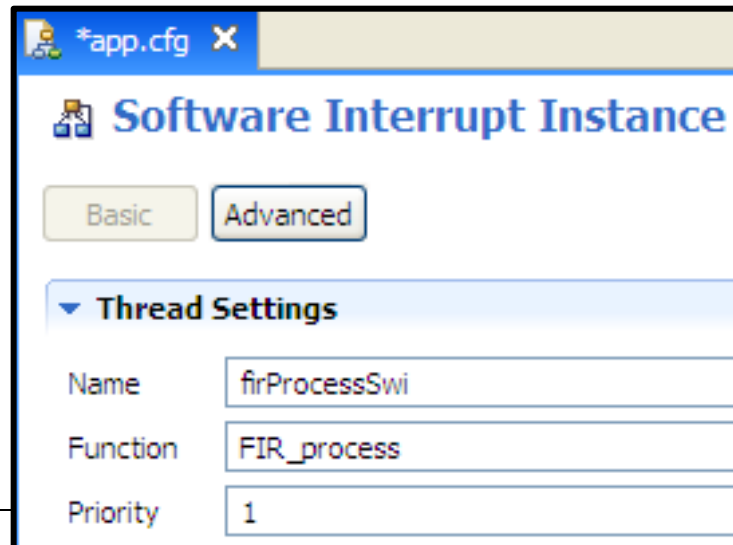
Example: Tie isrAudio() fxn to Swi, use priority 1

- 1 Use Swi module (Available Products) , insert new Hwi (Outline View)



Remember, BIOS objects can be created via the GUI, script code or C code (dynamic)

- 2 Configure Swi – Object name, function, priority:



Let's move on to Tasks...

SYS/BIOS Swi APIs

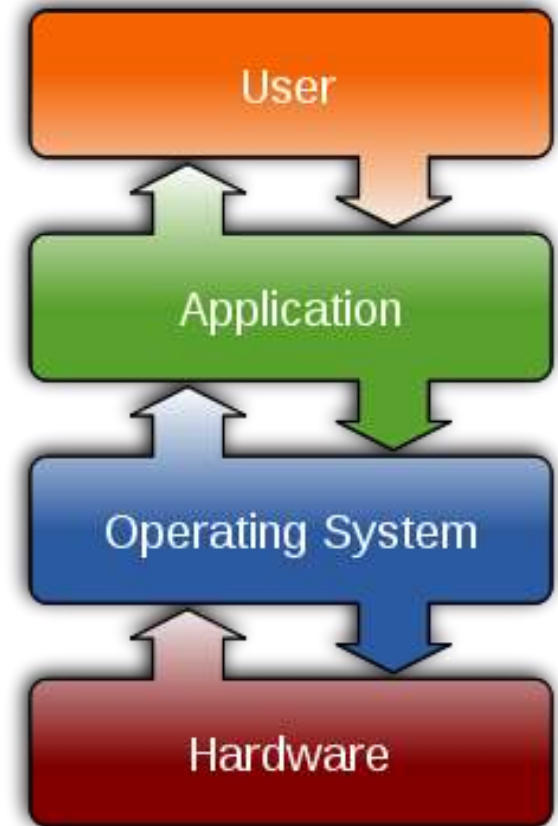
Other useful Swi APIs:

<code>Swi_inc()</code>	Post, increment count
<code>Swi_dec()</code>	Decrement count, post if 0
<code>Swi_or()</code>	Post, OR bit (signature)
<code>Swi_andn()</code>	ANDn bit, post if all posted
<code>Swi_getPri()</code>	Get any Swi Priority
<code>Swi_enable</code>	Global Swi enable
<code>Swi_disable()</code>	Global Swi disable
<code>Swi_restore()</code>	Global Swi restore

Let's move on to Tasks...

Outline

- ◆ Intro to SYS/BIOS
- ◆ BIOS Threads
 - ◆ Hardware Interrupts (HWI)
 - ◆ Software Interrupts (SWI)
 - ◆ Tasks (TSK)
 - ◆ Semaphores (SEM)



Task Scheduling

Hard
R/T

Hwi (hi)

Hardware Interrupts

- ◆ Hwi priorities set by hardware
- ◆ Fixed number, preemption optional

Swi

Software Interrupts

- ◆ Up to 32 priority levels (16 on C28x)
- ◆ Any number possible, all preemptive

Task

Tasks

- ◆ Up to 32 priority levels (16 on C28x)
- ◆ Any number possible, all preemptive

Soft
R/T

Idle (lo)

Background

- ◆ Continuous loop
- ◆ Non-realtime in nature

- ◆ All Tasks are preempted by all Swi and Hwi
- ◆ All Swi are preempted by all Hwi
- ◆ Preemption amongst Hwi is determined by user
- ◆ In absence of Hwi, Swi, and Task, Idle functions run in loop

Task Code Topology – Pending



```
Void taskFunction(...)
```

```
{
```

```
/* Prolog */
```

```
while ('condition'){
```

```
    Semaphore_pend()
```

```
    /* Process */
```

```
}
```

```
/* Epilog */
```

```
}
```

- ◆ **Initialization** (runs once only)
- ◆ **Processing** loop – (optional: *cond*)
- ◆ Wait for resources to be available
- ◆ Perform desired algo work...
- ◆ **Shutdown** (runs once - at most)

- ◆ Task can encompass *three* phases of activity
- ◆ Semaphore can be used to signal resource availability to Task
- ◆ Semaphore_pend () *blocks* Task until semaphore (flag) is posted

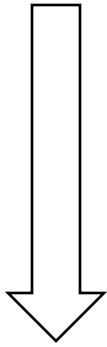
[Let's compare/contrast Swi & Task...](#)

Swi vs. Task

Swi

`_post` →

```
void mySwi () {  
    // set local env  
  
    *** RUN ***  
}
```




- “Ready” when POSTED
- Initial state NOT preserved – must set each time **Swi** is run
- CanNOT block (runs to completion)
- Context switch speed (~140c)
- All **Swi's** share system stack w/Hwi
- Use: as follow-up to Hwi and/or when memory size is an absolute premium

Task

`_create` →

```
void myTask () {  
    // Prologue (set Task env)  
    while(cond) {  
        Semaphore_pend() ;  
        *** RUN ***  
    }  
    // Epilogue (free env)  
}
```

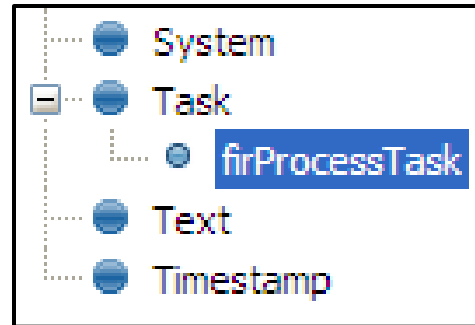
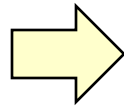
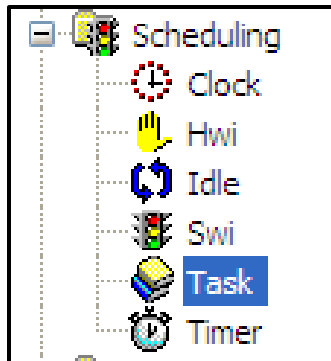


- “Ready” when CREATED (BIOS_start or dynamic)
- P-L-E structure handy for resource creation (P) and deletion (E), initial state preserved
- Can block/suspend on semaphore (flag)
- Context switch speed (~160c)
- Uses its OWN stack to store context
- Use: Full-featured sys, CPU w/more speed/mem

Configuring a Task – Statically via the GUI

Example: Create `firProcessTask`, tie to `FIR_process()`, priority 2

- 1 Use Task module (*Available Products*) , insert new Task (*Outline View*)



Remember, BIOS objects can be created via the GUI, script code or C code (dynamic)

- 2 Configure Task – Object name, function, priority, stack size:

Thread Settings

Name: `firProcessTask`

Function: `FIR_process`

Priority: `2`

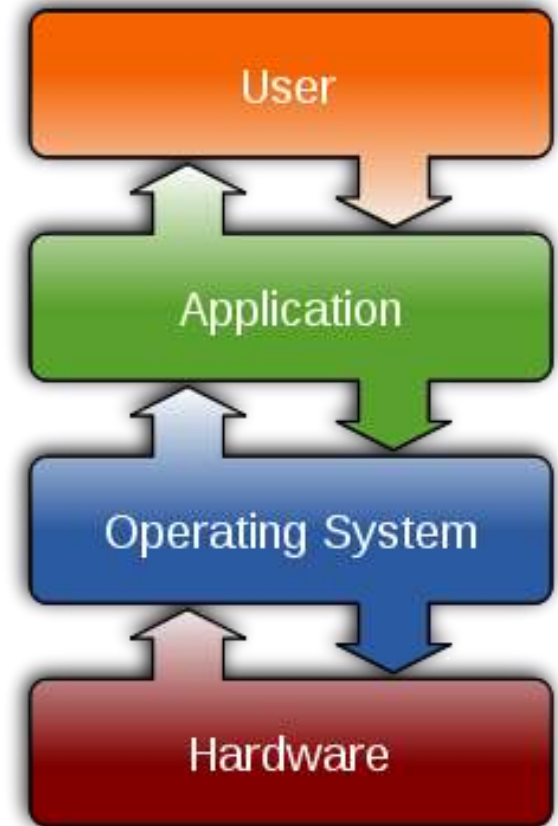
Use the vital flag to prevent system exit until this task is complete:
☒ Task is vital

Stack Control Options

Stack size: `2048`

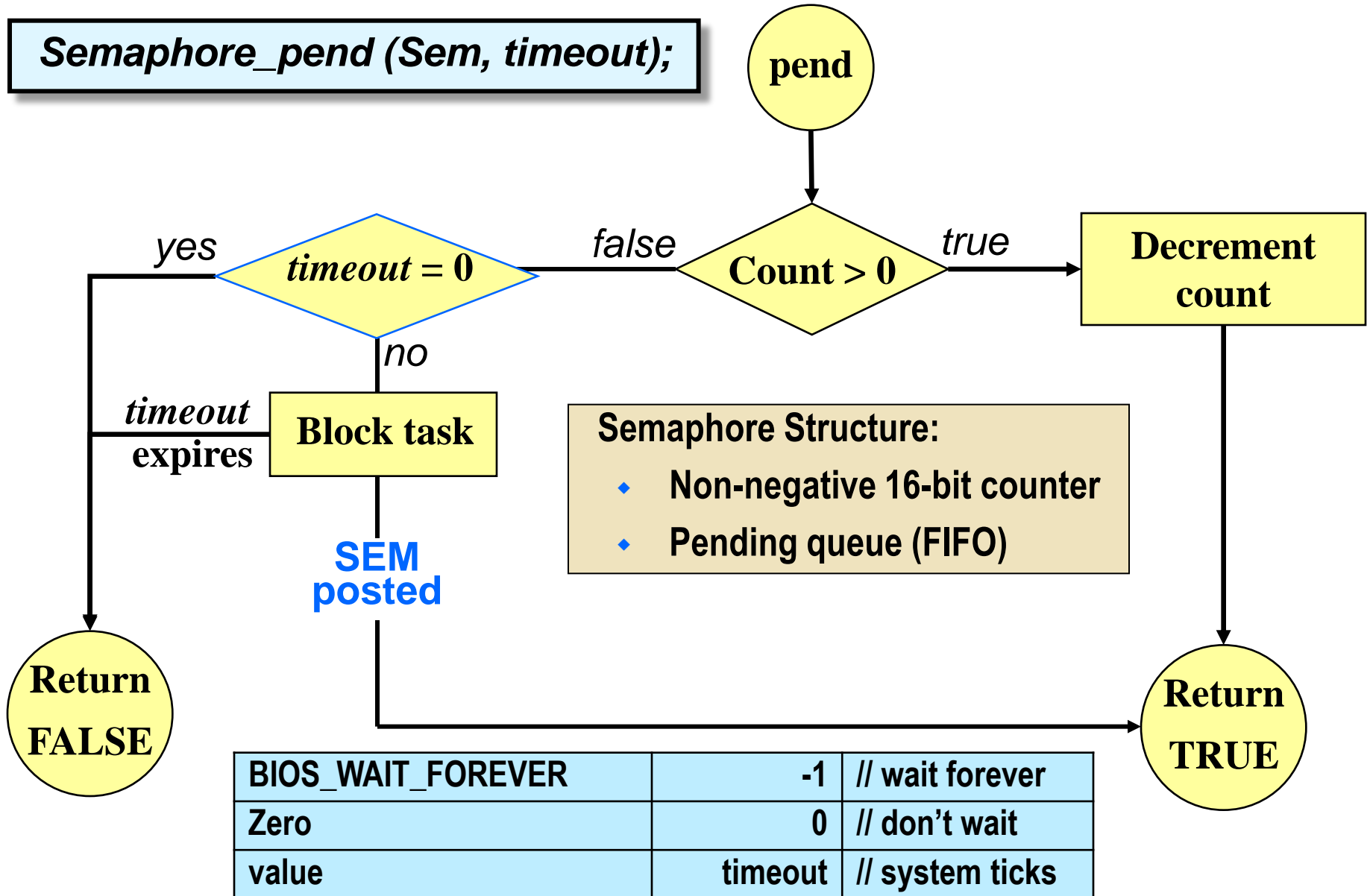
Outline

- ◆ Intro to SYS/BIOS
- ◆ BIOS Threads
 - ◆ Hardware Interrupts (HWI)
 - ◆ Software Interrupts (SWI)
 - ◆ Tasks (TSK)
 - ◆ Semaphores (SEM)

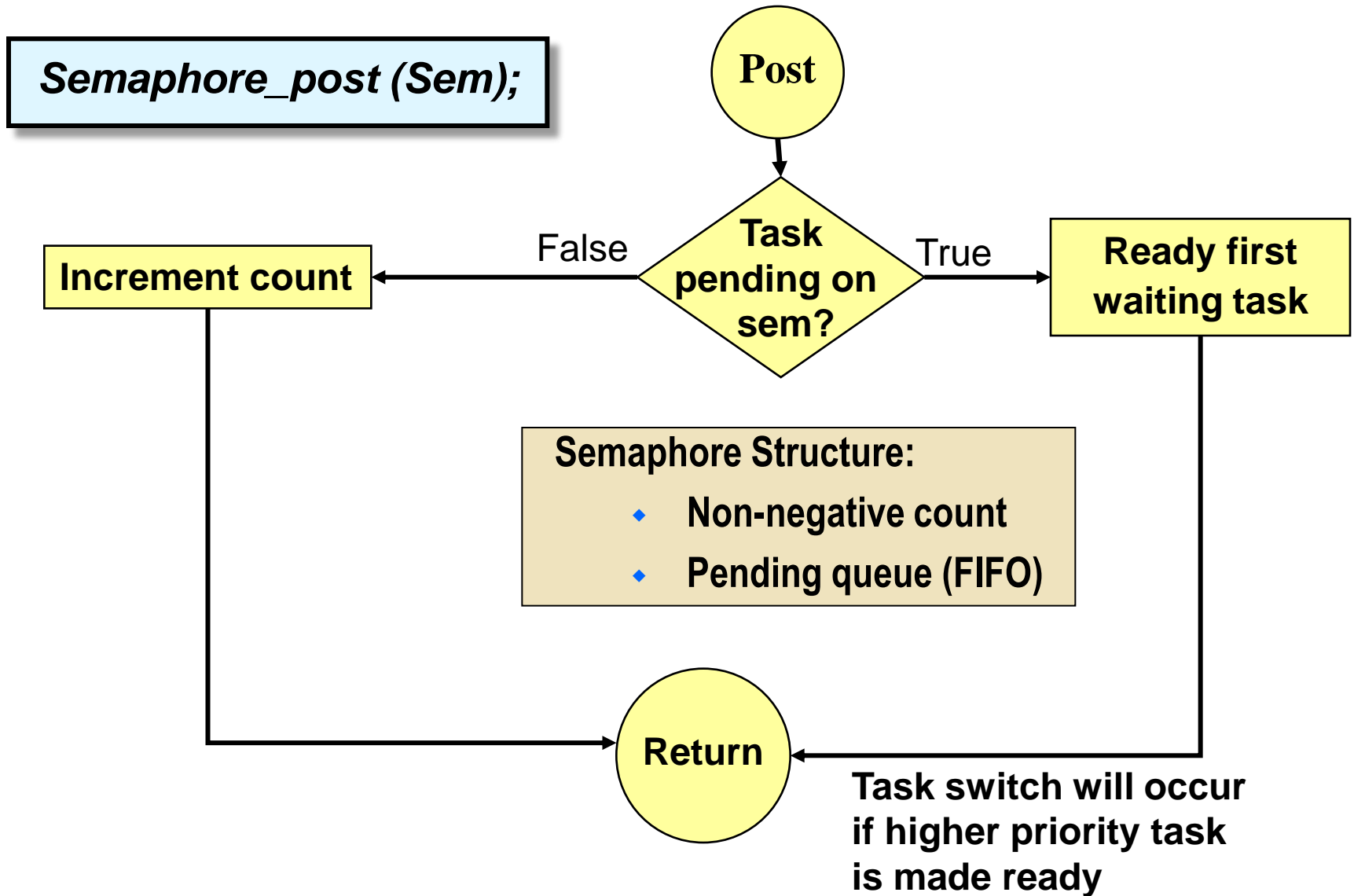


Semaphore Pend

Semaphore_pend (Sem, timeout);



Semaphore Post

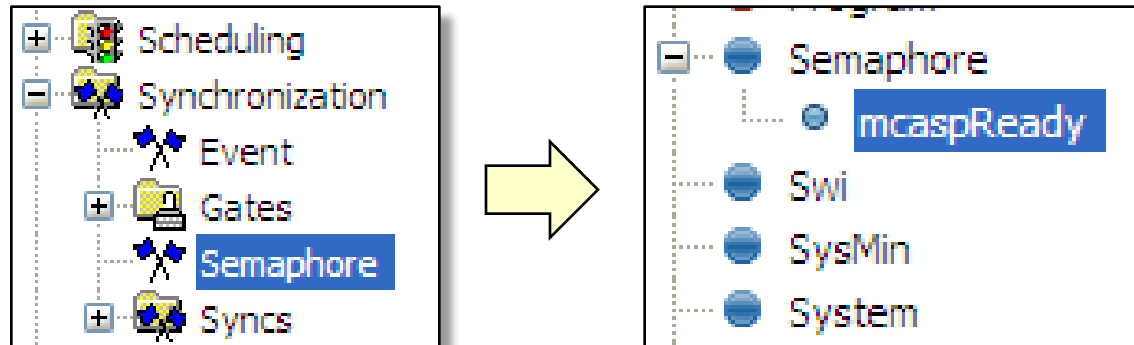


How do you configure a Semaphore?

Configuring a Semaphore – Statically via GUI

Example: Create mcasReady, counting

- 1 Use Semaphore (Available Products) , insert new Semaphore (Outline View)



- 2 Configure Semaphore – Object name, initial count, type:

The screenshot shows the 'Semaphore Instance' configuration dialog. The 'Basic' tab is selected. Under 'Required Settings', the 'Name' field is set to 'mcaspReady', the 'Initial count' is set to '0', and the 'Semaphore type' is set to 'Counting semaphore' (radio button selected).

SYS/BIOS Semaphore/Task APIs

Other useful Semaphore APIs:

<code>Semaphore_getCount()</code>	Get semaphore count
-----------------------------------	---------------------

Other useful Task APIs:

<code>Task_sleep()</code>	Sleep for N system ticks
<code>Task_yield()</code>	Yield to same pri Task
<code>Task_setPri()</code>	Set Task priority
<code>Task_getPri()</code>	Get Task priority
<code>Task_get/setEnv()</code>	Get/set Task Env
<code>Task_enable()</code>	Enable Task Mgr
<code>Task_disable()</code>	Disable Task Mgr
<code>Task_restore()</code>	Restore Task Mgr

Questions?