Introduction to SYS/BIOS

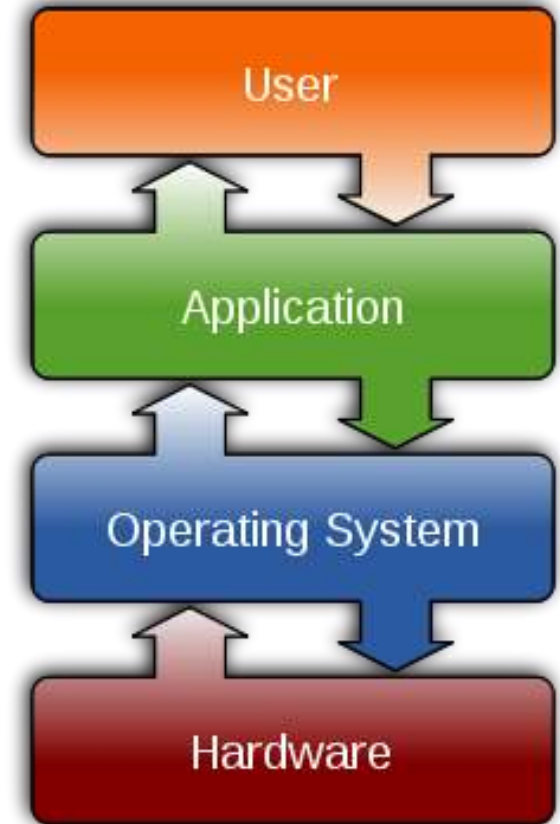# Outline

Outline

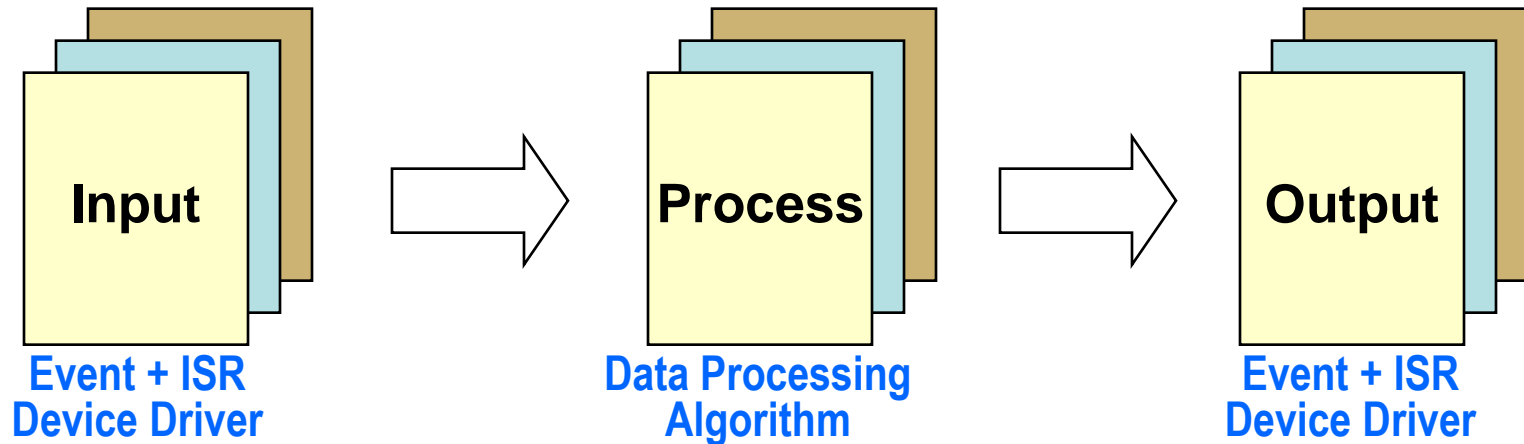◆ **Intro to SYS/BIOS**

  ◆ **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**

User

Application

Operating System

Hardware
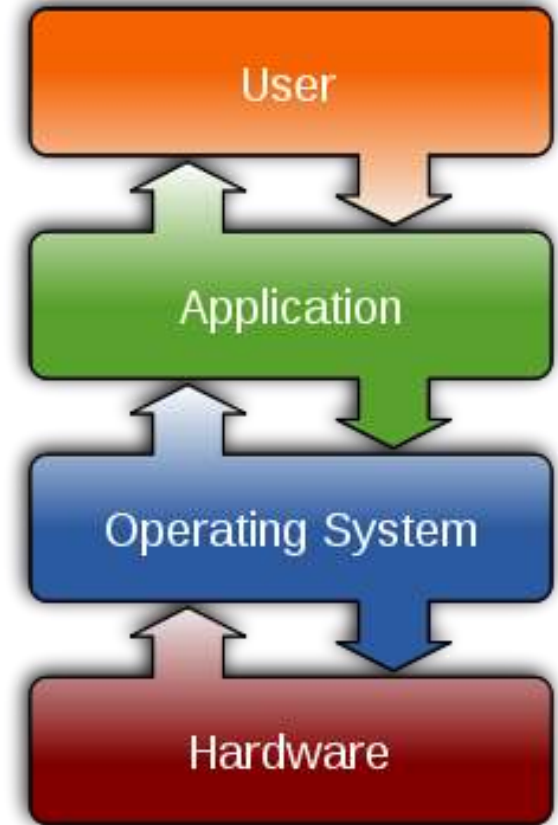
Need for an Operating System

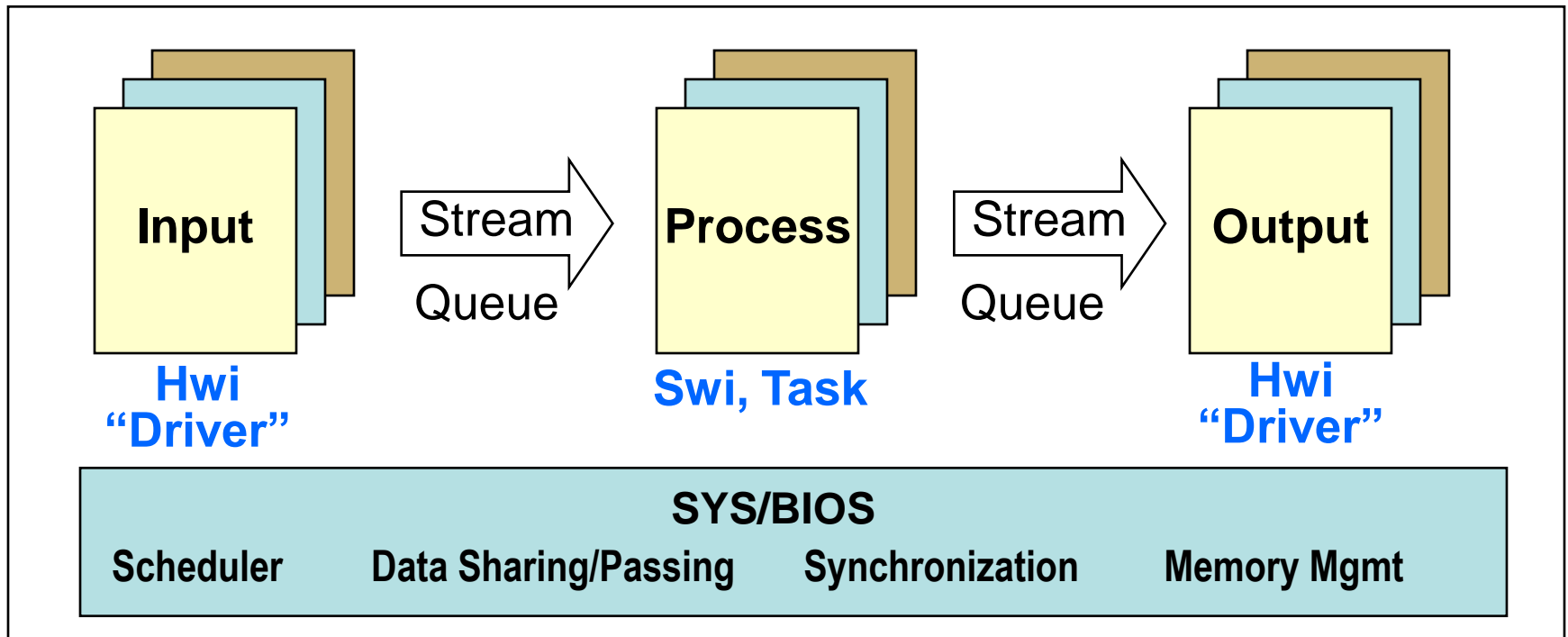|		
Input	**Process**	**Output**
Event + ISR Device Driver	Data Processing Algorithm	Event + ISR Device Driver

- **Simple system:** single I-P-O is easy to manage

- As system complexity increases (**multiple threads**):

  - ➤ **Can they all meet real time ?**
  - ➤ **Priorities of threads/algos ?**
  - ➤ **Synchronization of events?**
  - ➤ **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 easy to use ?**
  - ➤ **How much does it cost ?**
  - ➤ **Is it reliable?**
  - ➤ **Data sharing/passing ?**
  - ➤ **What code overhead exists?**

SYS/BIOS Overview

Input	Stream Queue	Process	Stream Queue	Output
Hwi "Driver"		**Swi, Task**		**Hwi "Driver"**

SYS/BIOS

Scheduler　　Data Sharing/Passing　　Synchronization　　Memory Mgmt

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

- Pre-emptive <u>Scheduler</u> to design system to meet real-time (including sync/priorities)

- <u>Modular</u> – pre-defined interface for inter-thread communications

- <u>Reliable</u> – 1000s of applications have used it for more than 10 years

- <u>Footprint</u> – deterministic, small code size, can choose which modules you desire

- Cost – <u>free of charge</u>
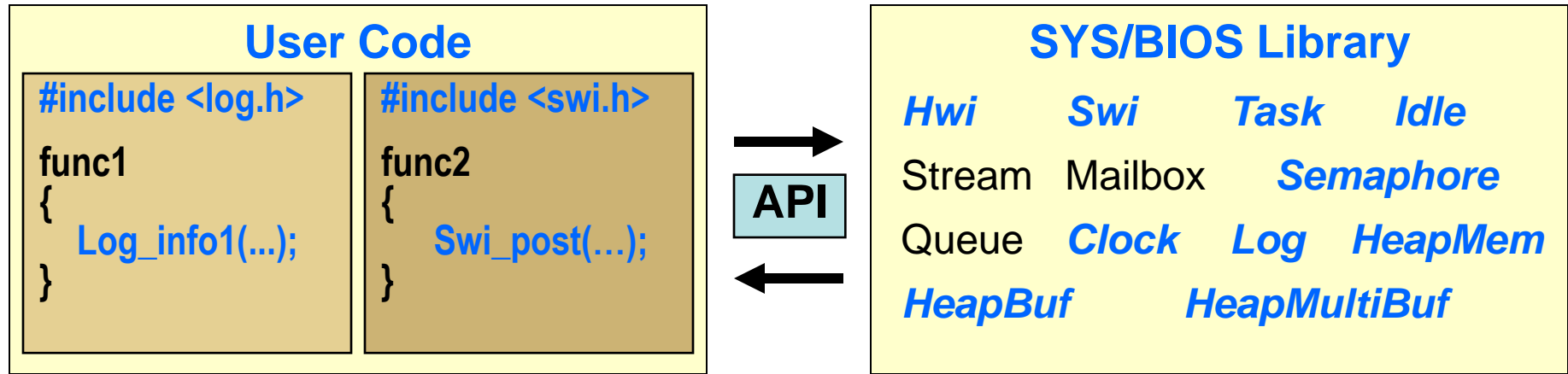
SYS/BIOS Modules & Services
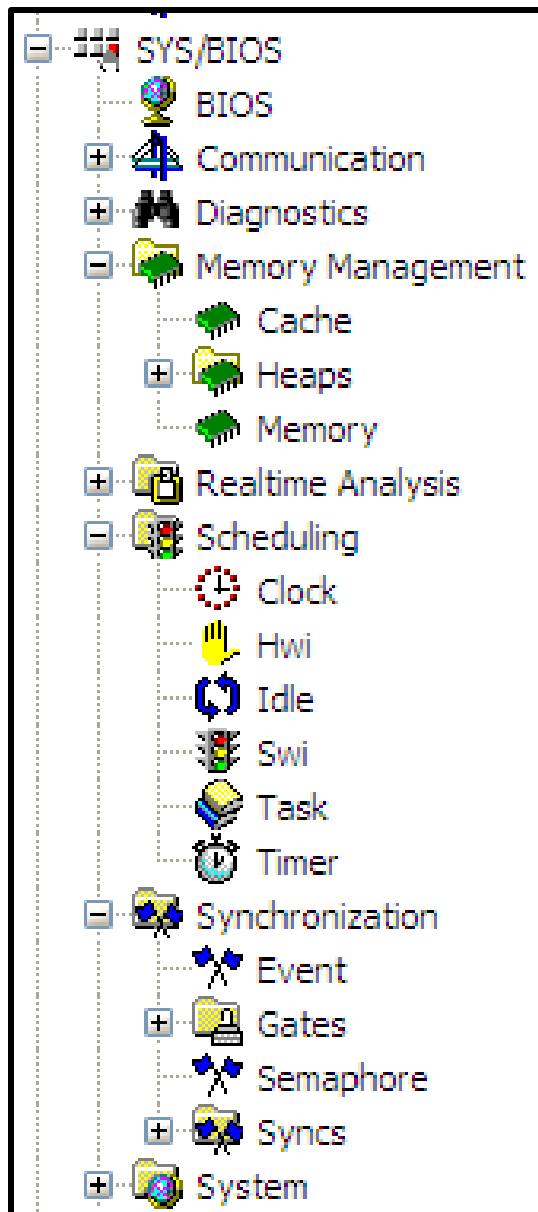
BIOS Configuration

- **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

User Code	SYS/BIOS Library

User Code

```
#include <log.h>

func1
{
    Log_info1(...);
}
```

```
#include <swi.h>

func2
{
    Swi_post(...);
}
```

API

SYS/BIOS Library

Hwi *Swi* *Task* *Idle*

Stream Mailbox *Semaphore*

Queue *Clock* *Log* *HeapMem*

HeapBuf *HeapMultiBuf*

◆ 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*

pointer *structure....*

| handle | → | element1
element2
... |

object
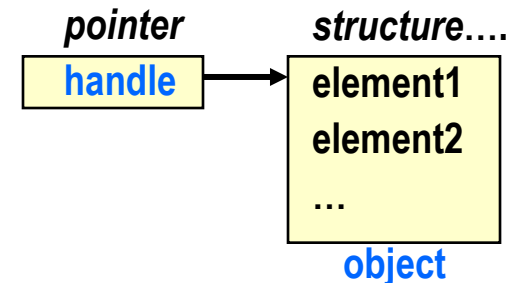
Definitions / Vocabulary

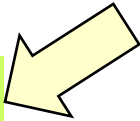◆ In this workshop, we'll be using these terms often:

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

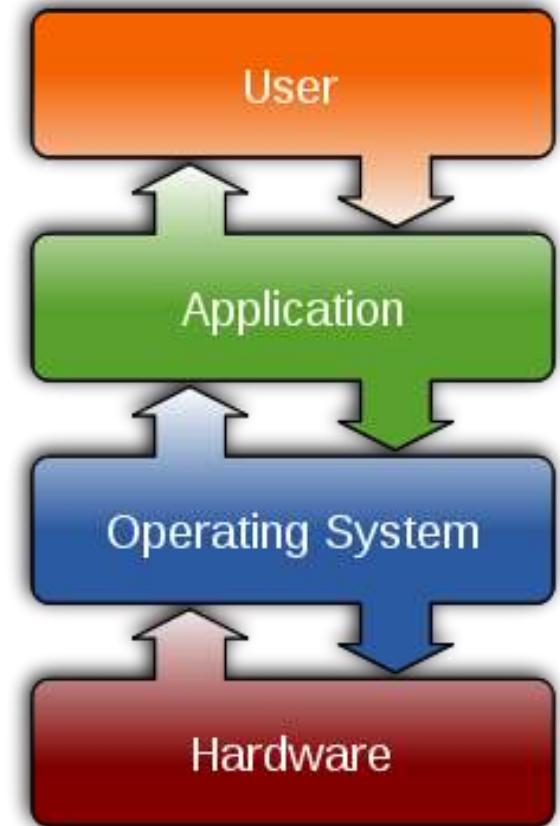➢ *Function that executes within a specific context (regs, stack, PRIORITY)*

API

➢ *Application Programming Interface – "methods" for interacting with library routines and data objects*
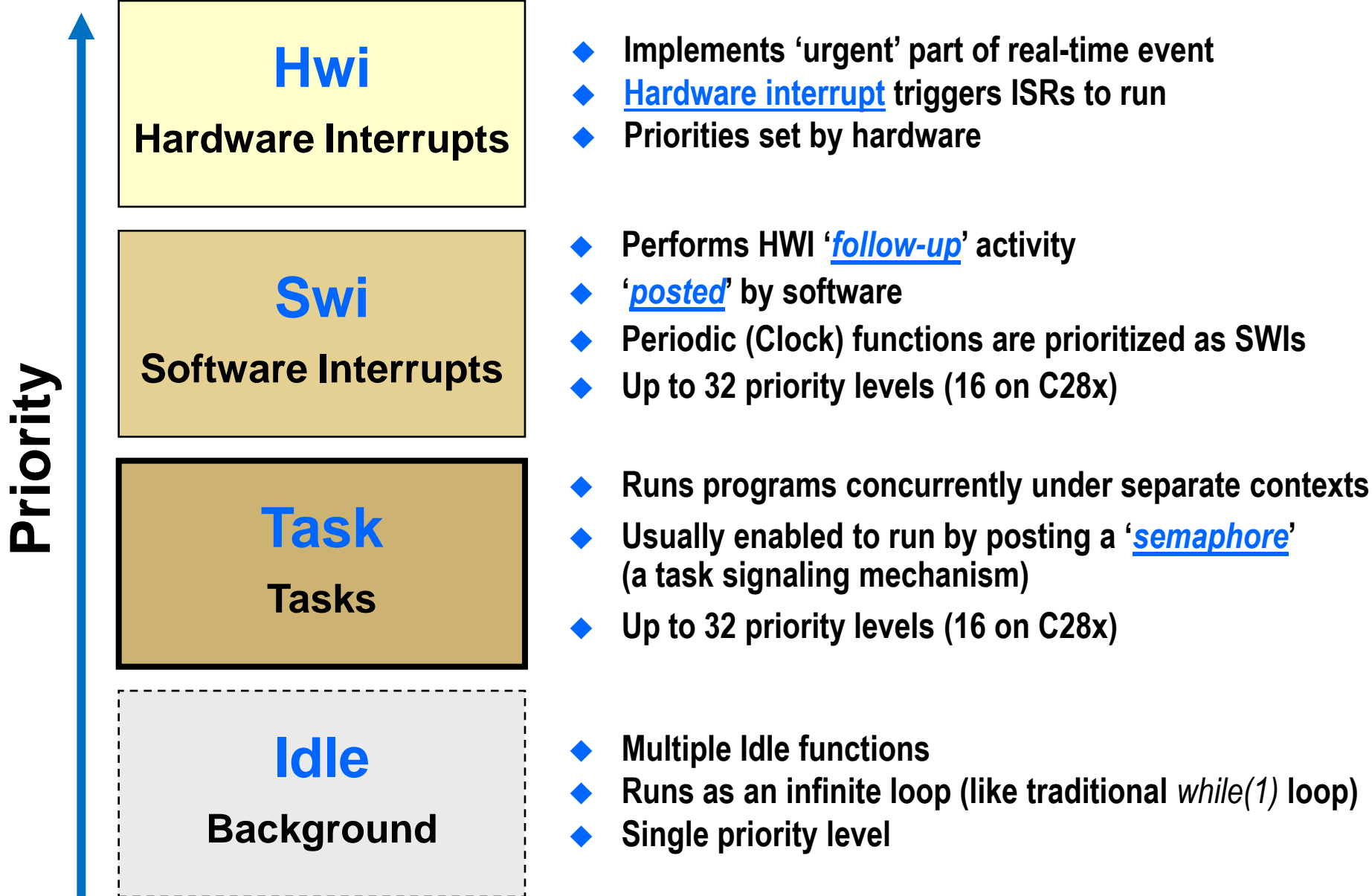
RTOS vs GP/OS

	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**

User

Application

Operating System

Hardware

SYS/BIOS Thread Types

Priority ↑

Hwi
Hardware Interrupts

- Implements 'urgent' part of real-time event
- <u>Hardware interrupt</u> triggers ISRs to run
- Priorities set by hardware

Swi
Software Interrupts

- Performs HWI '*follow-up*' activity
- '*posted*' by software
- Periodic (Clock) functions are prioritized as SWIs
- Up to 32 priority levels (16 on C28x)

Task
Tasks

- Runs programs concurrently under separate contexts
- Usually enabled to run by posting a '*semaphore*' (a task signaling mechanism)
- Up to 32 priority levels (16 on C28x)

Idle
Background

- Multiple Idle functions
- Runs as an infinite loop (like traditional *while(1)* loop)
- Single priority level

Hwi's Signaling Swi/Task

INTx

Hwi:
 urgent code
Swi_post();
  [OR]
Semaphore_post();

Swi (or Task)

New paradigm: "Hwi (ISR) handles *URGENT* activity, then posts follow-up thread"

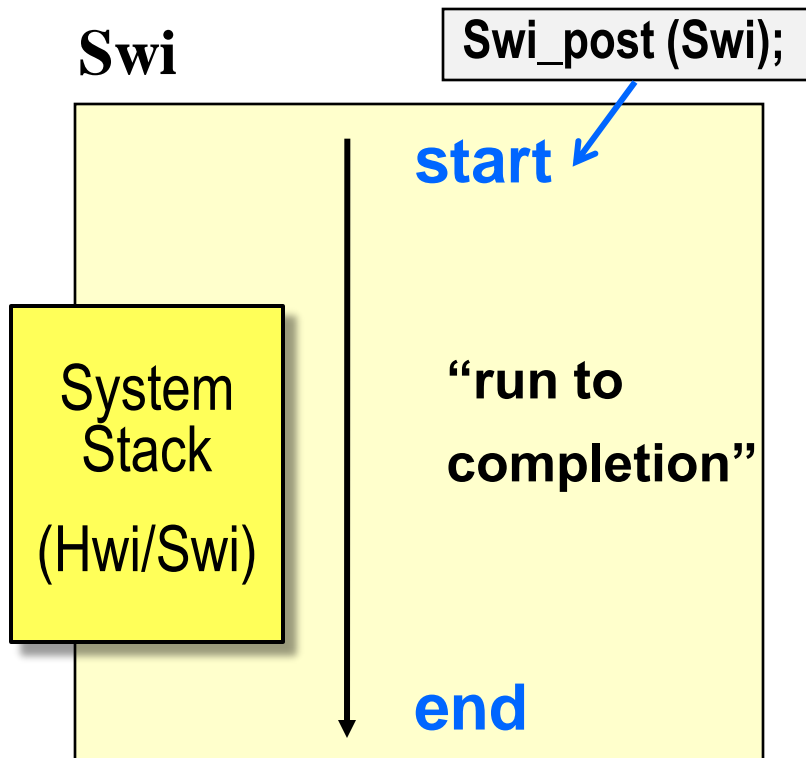◄── ints disabled ──► ◄────── rather than all this time ──────►

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

Swi

Swi_post (Swi);

Task

Semaphore_post (Sem);

start

System Stack (Hwi/Swi)

"run to completion"

end

Semaphore_pend → **Pause** (blocked state)

start

end

Private Stack (Task)

- **Similar to hardware interrupt, but triggered when posted**
- **All Swi's share system software stack with Hwi's**
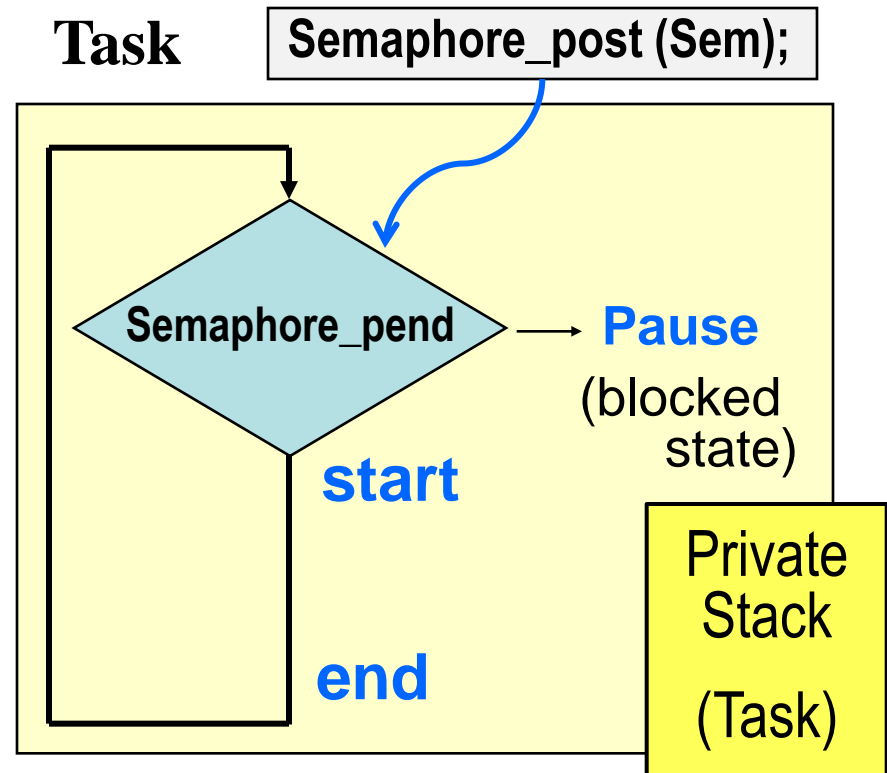
- **Unblocking triggers execution (also could be mailbox, events, etc.)**
- **Each Task has its own stack, which allows them to pause (i.e. block)**
- **Topology: prologue, loop, epilogue…**

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**
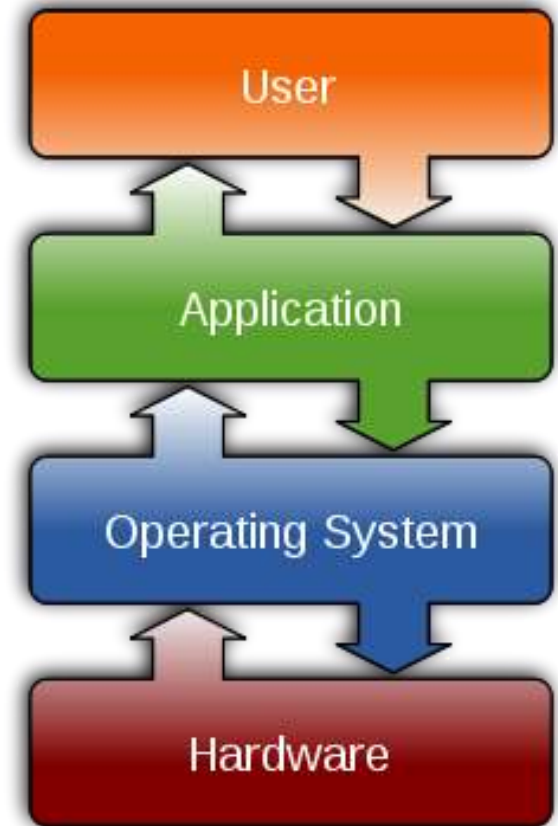
Thread (Object) Creation in BIOS

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);
```
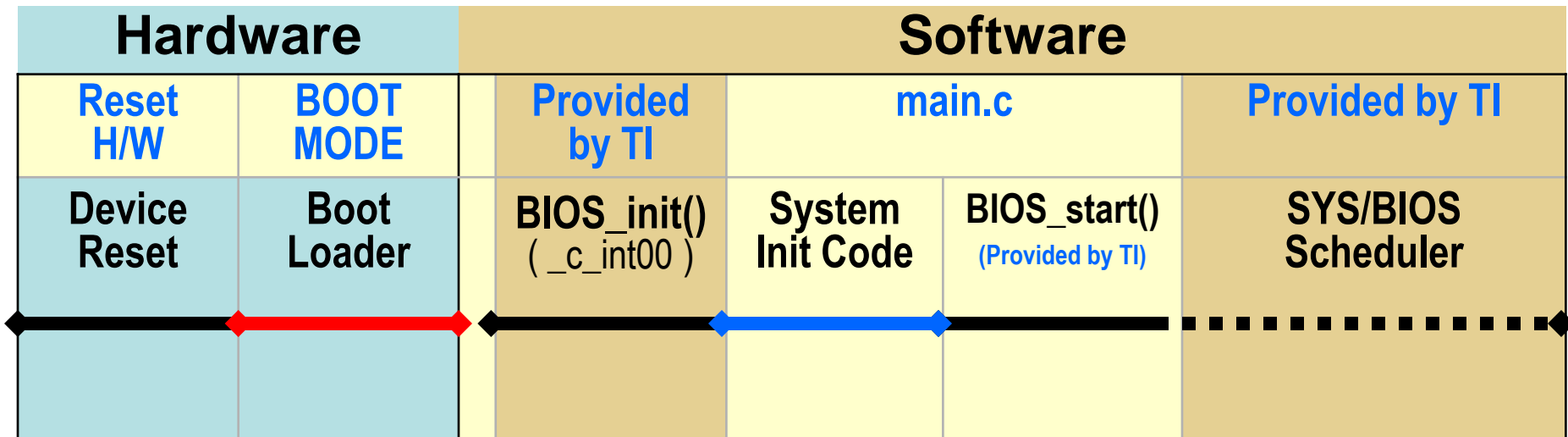
`app.cfg`

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

Hardware		Software			
Reset H/W	**BOOT MODE**	**Provided by TI**	**main.c**		**Provided by TI**
Device Reset	**Boot Loader**	**BIOS_init()** (_c_int00)	**System Init Code**	**BIOS_start()** (Provided by TI)	**SYS/BIOS Scheduler**

- ◆ **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)**
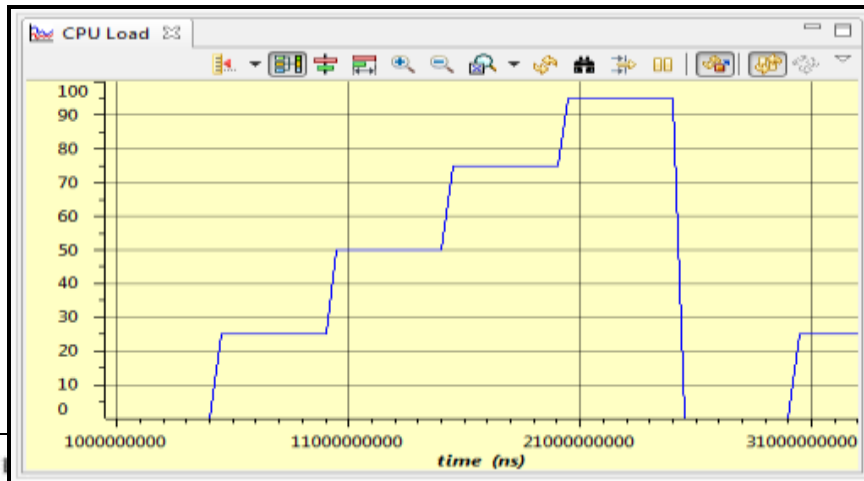 - ◆ **Platforms**
  - ◆ **For More Info…..**
- ◆ **BIOS Threads**
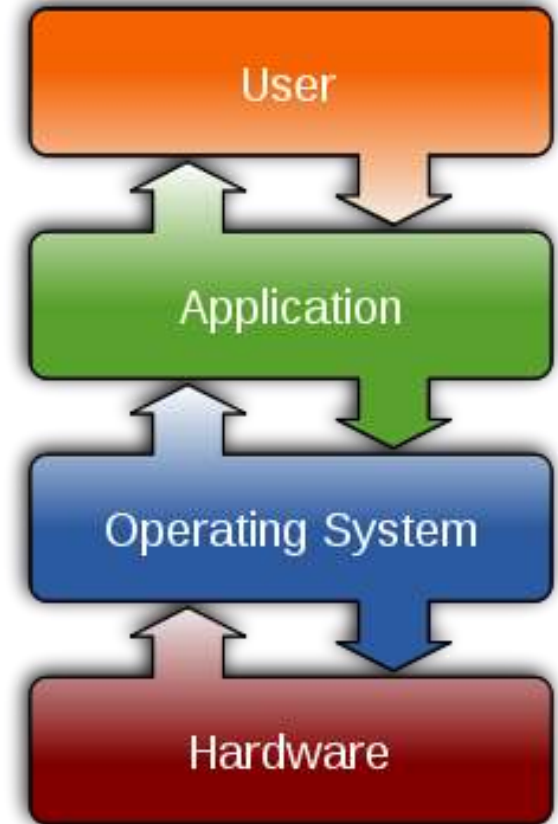
Built-in Real-Time Analysis Tools

- ◆ 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)
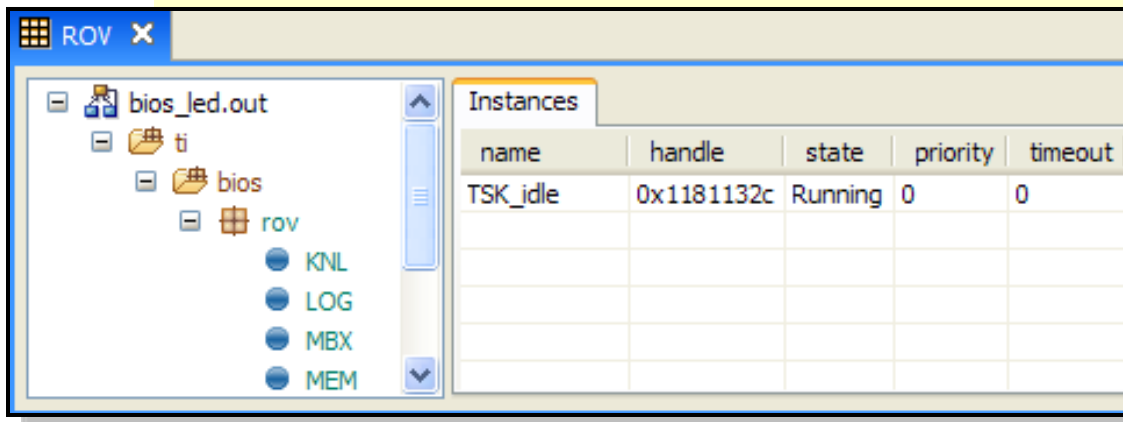
RunTime Obj View (ROV)

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

CPU/Thread Load Graph

- ◆ **Analyze time NOT spent in Idle**
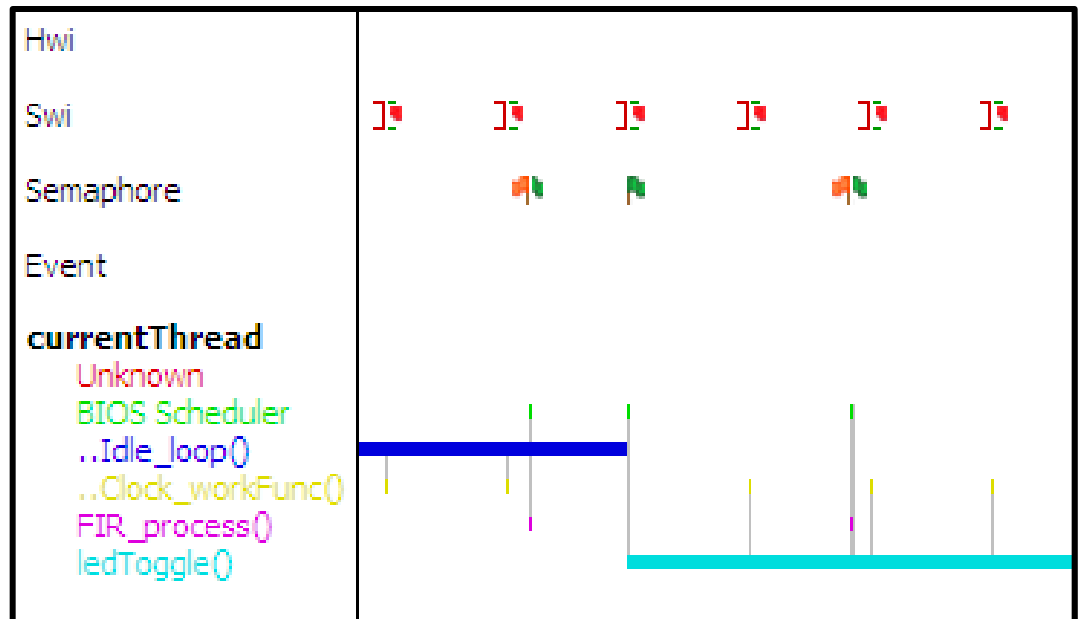
Built-in Real-Time Analysis Tools

Logs

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

|	Raw Logs ✕		
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

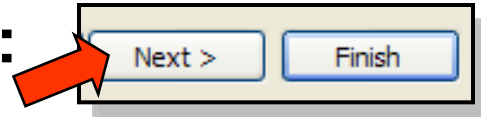- **View system events down to the CPU cycle…**
- **Calculate benchmarks**

Hwi
Swi
Semaphore
Event
currentThread
Unknown
BIOS Scheduler
..Idle_loop()
..Clock_workFunc()
FIR_process()
ledToggle()

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)**
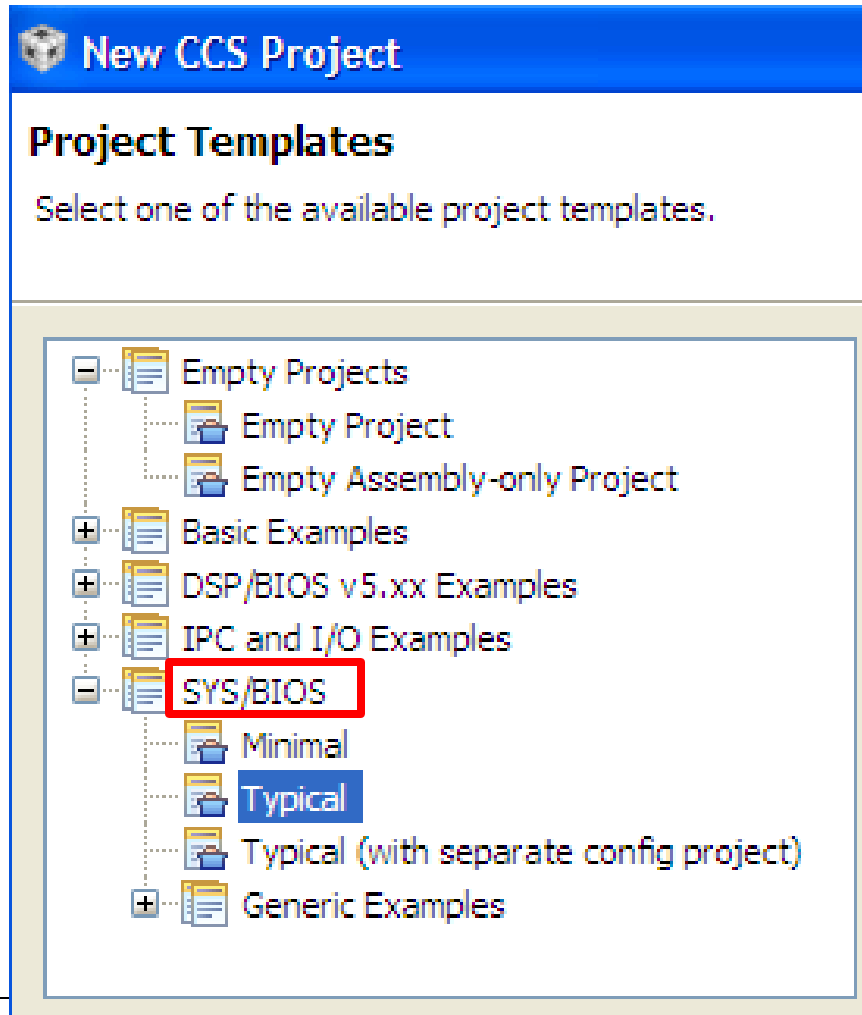 - **Platforms**
  - **For More Info…..**
- **BIOS Threads**

User

Application

Operating System

Hardware
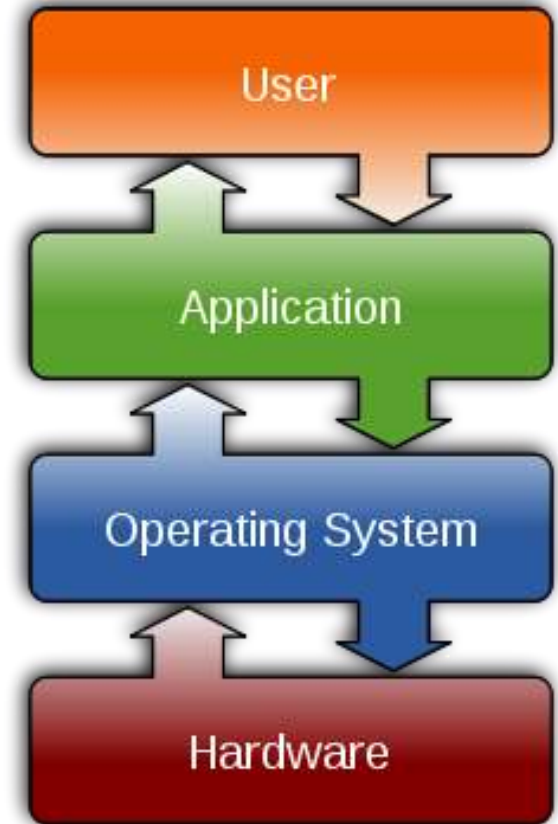
Building a **NEW** SYS/BIOS Project

◆ **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…)

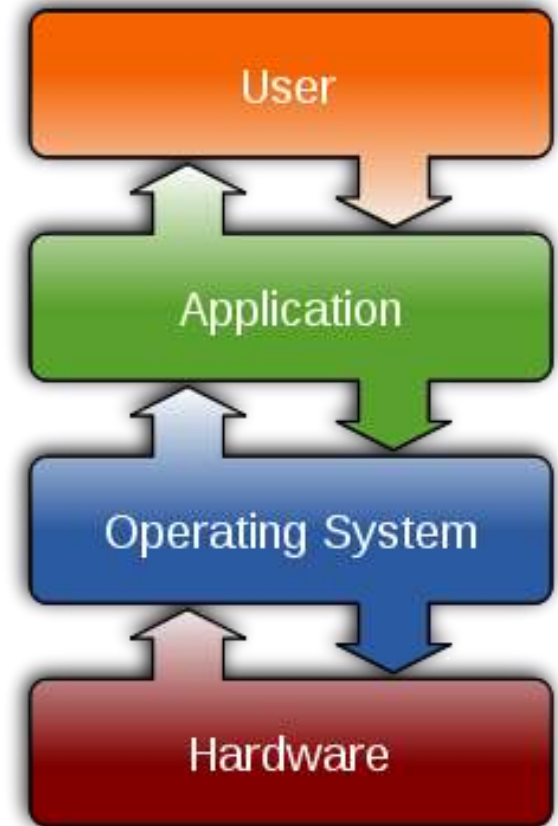- Source files (main.c) that contains appropriate #includes of header files

SYS/BIOS Project Settings

- ◆ Select versions for XDC, IPC, SYS/BIOS, xDAIS

- ◆ Select "Platform" file (similar to the .tcf seed file for memory)
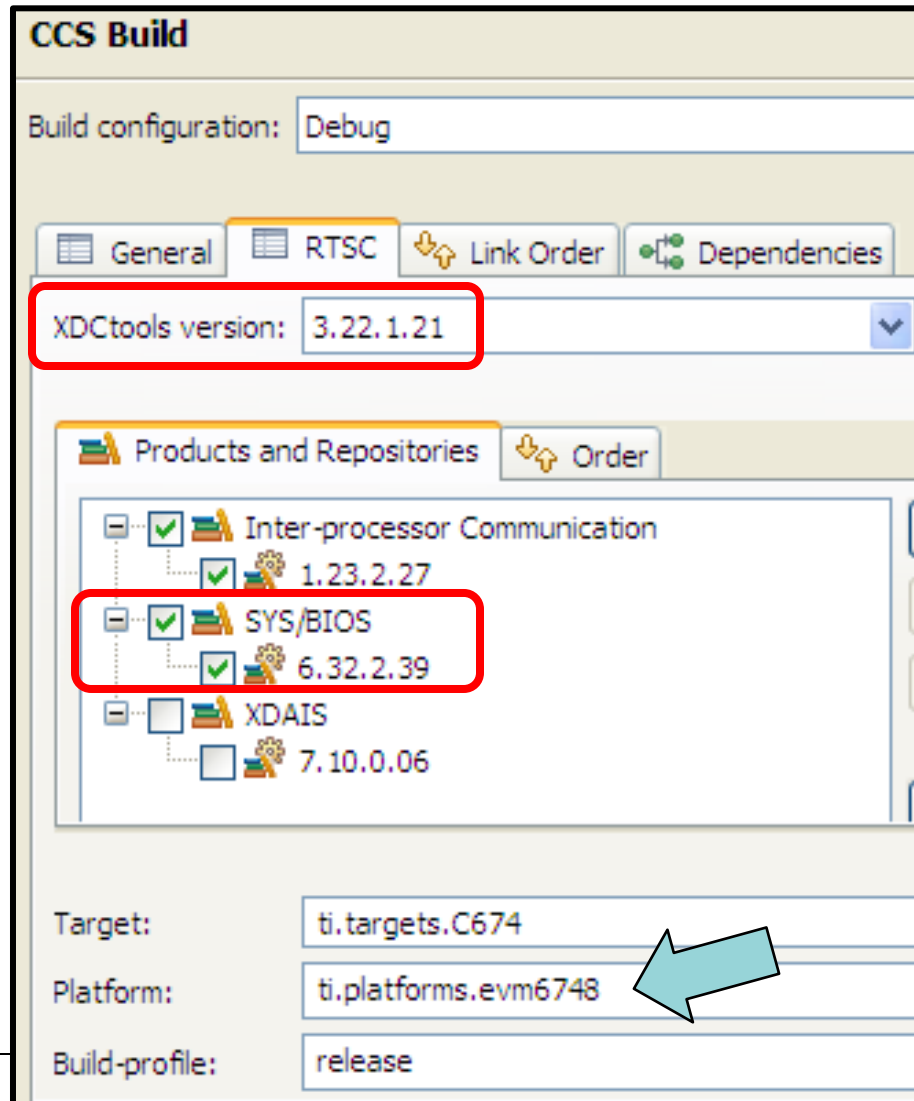
**CCS Build**

Build configuration: Debug

| General | RTSC | Link Order | Dependencies |

XDCtools version: 3.22.1.21

Products and Repositories | Order

- ☑ Inter-processor Communication
  - ☑ 1.23.2.27
- ☑ SYS/BIOS
  - ☑ 6.32.2.39
- ☐ XDAIS
  - ☐ 7.10.0.06

Target: ti.targets.C674

Platform: ti.platforms.evm6748

Build-profile: release

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**

User

Application

Operating System

Hardware

Static BIOS Configuration

◆ **Users interact with the CFG file via the GUI – XGCONF:**
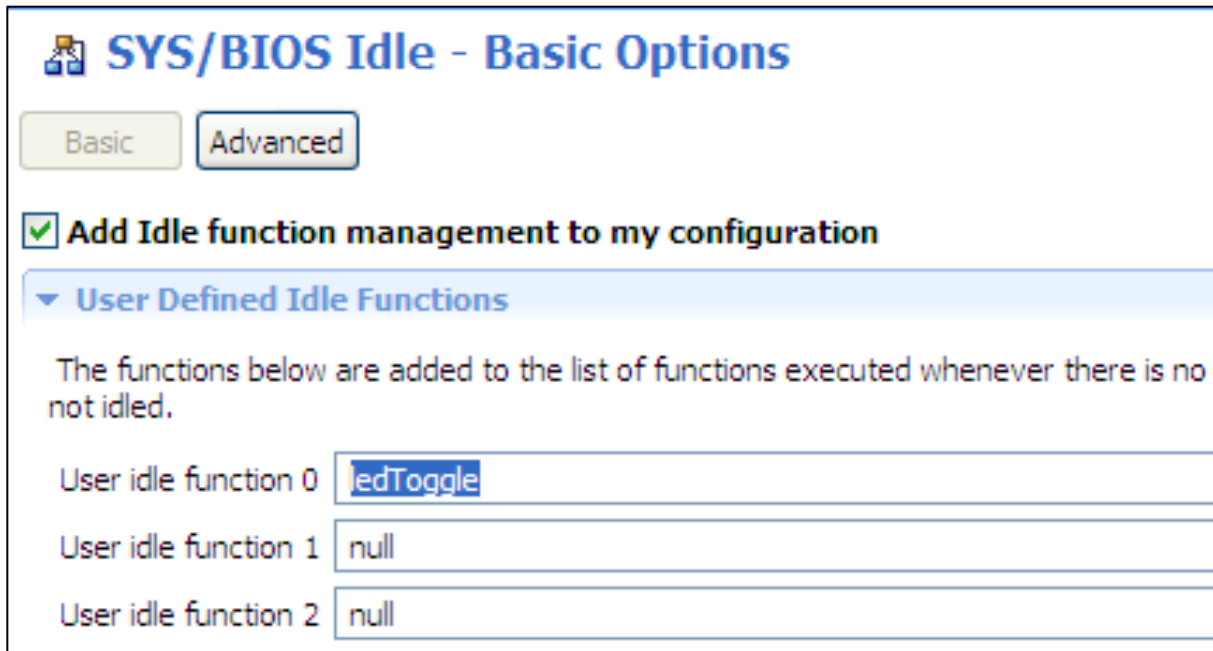
- XGCONF shows "Available Products" – Right-click and "Use Mod"
- "Mod" shows up in Outline view – Right-click and "Add New"
- All graphical changes in GUI displayed in **.cfg** source code
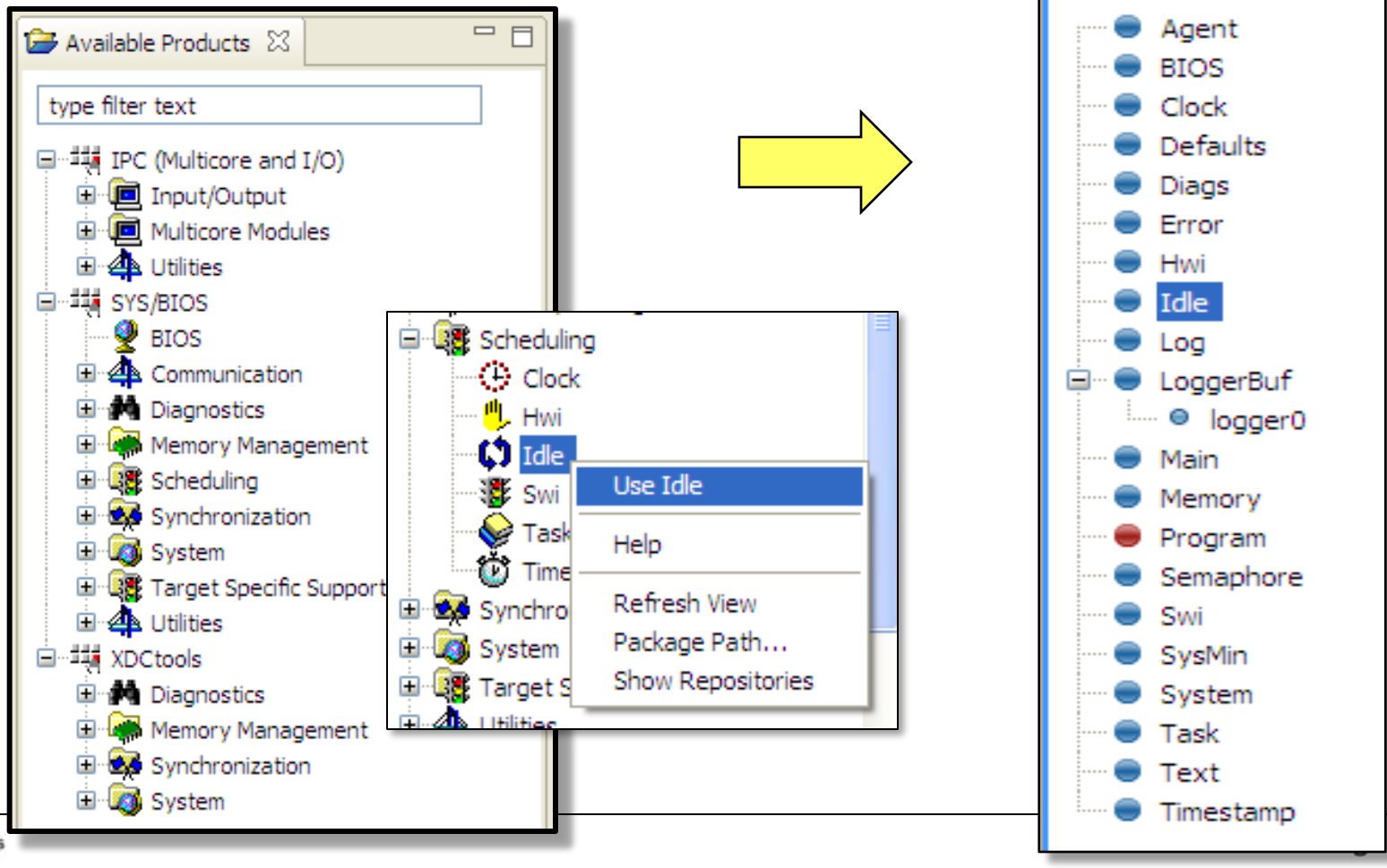
Static Config – .CFG Files

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

- When you "Add New", you get a dialogue box to set up parameters
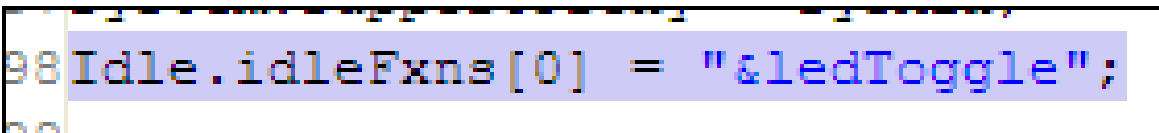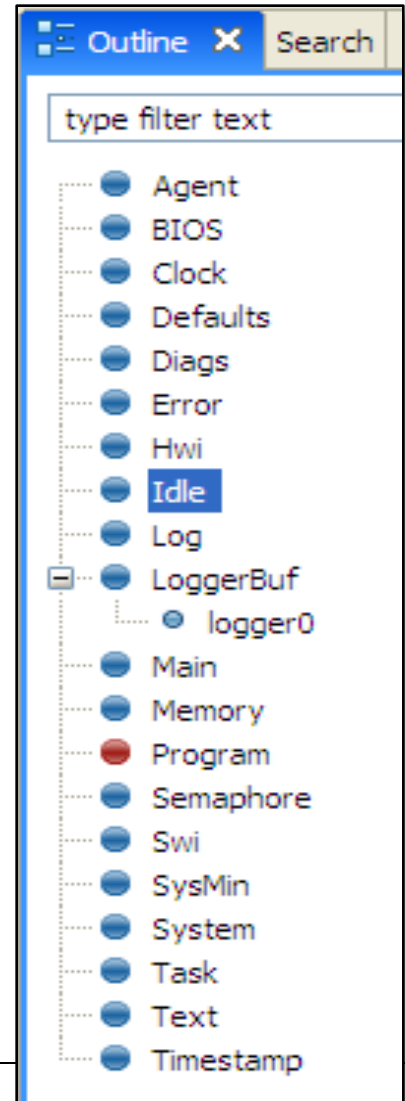- Two views: "Basic" and "Advanced"
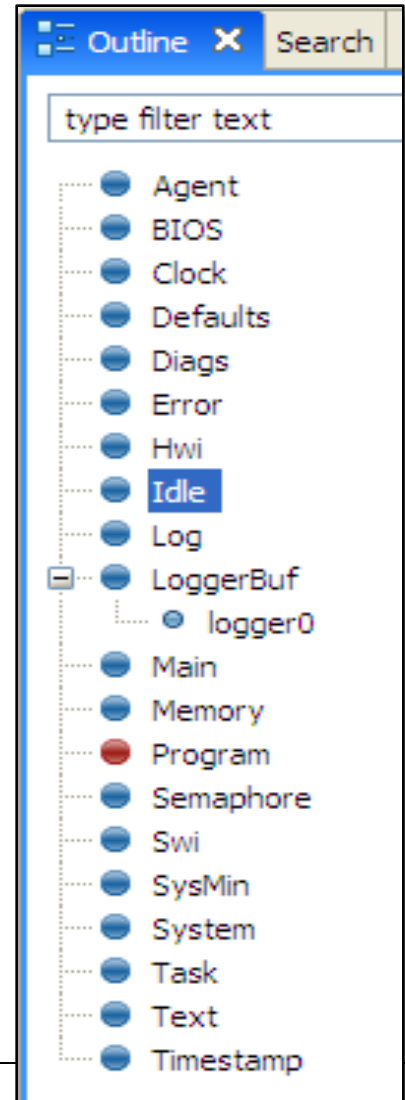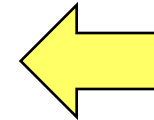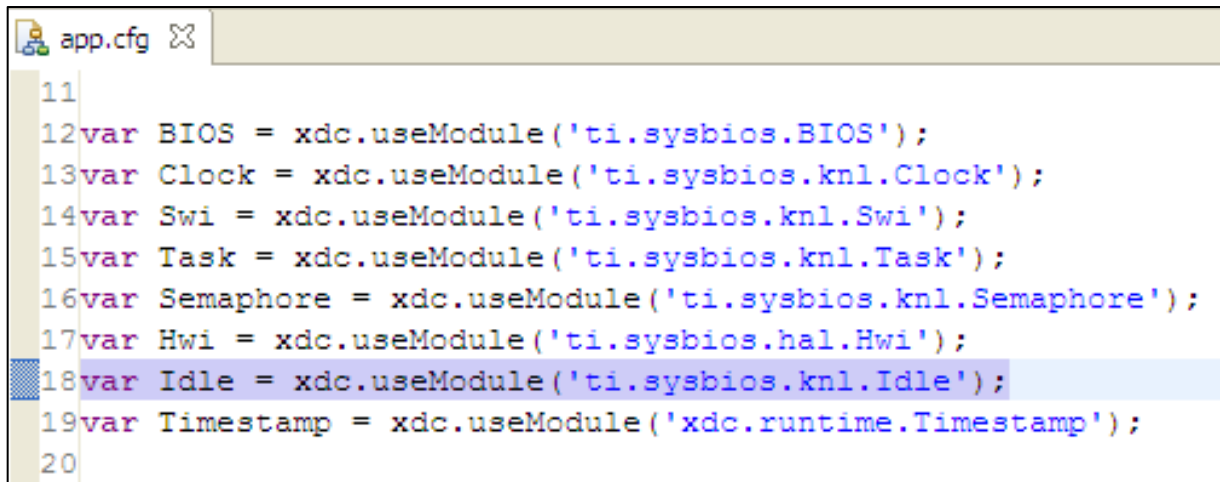
.CFG Files (XDC script)

◆ 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

```
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.idleFxns[0] = "&ledToggle";
```

Outline ✕ Search

type filter text

- ● Agent
- ● BIOS
- ● Clock
- ● Defaults
- ● Diags
- ● Error
- ● Hwi
- ● **Idle**
- ● Log
- ● LoggerBuf
    - ○ logger0
- ● Main
- ● Memory
- ● Program
- ● Semaphore
- ● Swi
- ● SysMin
- ● System
- ● Task
- ● Text
- ● Timestamp

Configuration Build Flow (CFG)

- <u>SYS/BIOS</u> – user configures system with <u>CFG file</u>
- The rest is "under the hood"
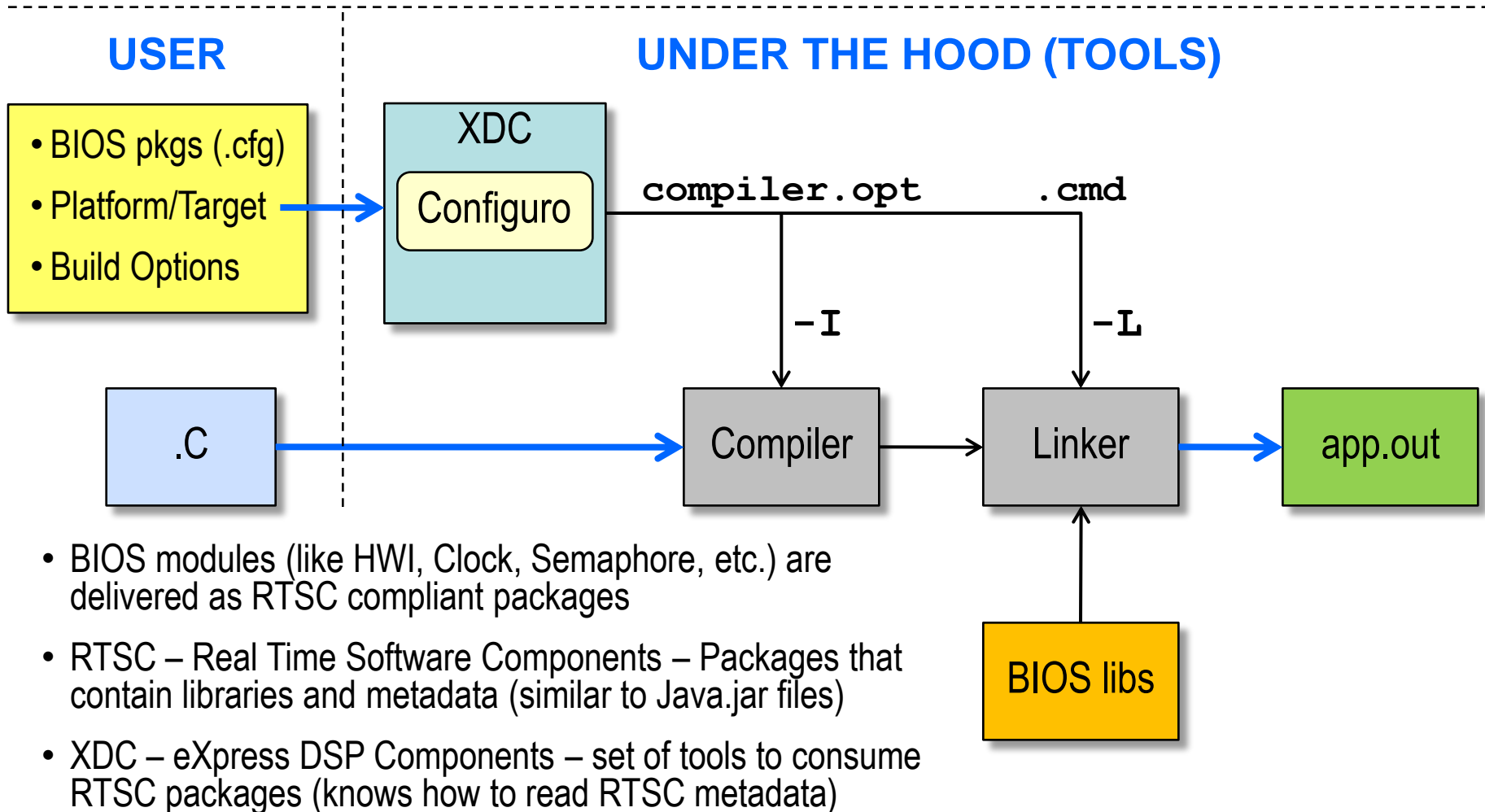
USER   **UNDER THE HOOD (TOOLS)**

- BIOS modules (like HWI, Clock, Semaphore, etc.) are delivered as RTSC compliant packages

- RTSC – Real Time Software Components – Packages that contain libraries and metadata (similar to Java.jar files)

- XDC – eXpress DSP Components – set of tools to consume RTSC packages (knows how to read RTSC metadata)

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…..**
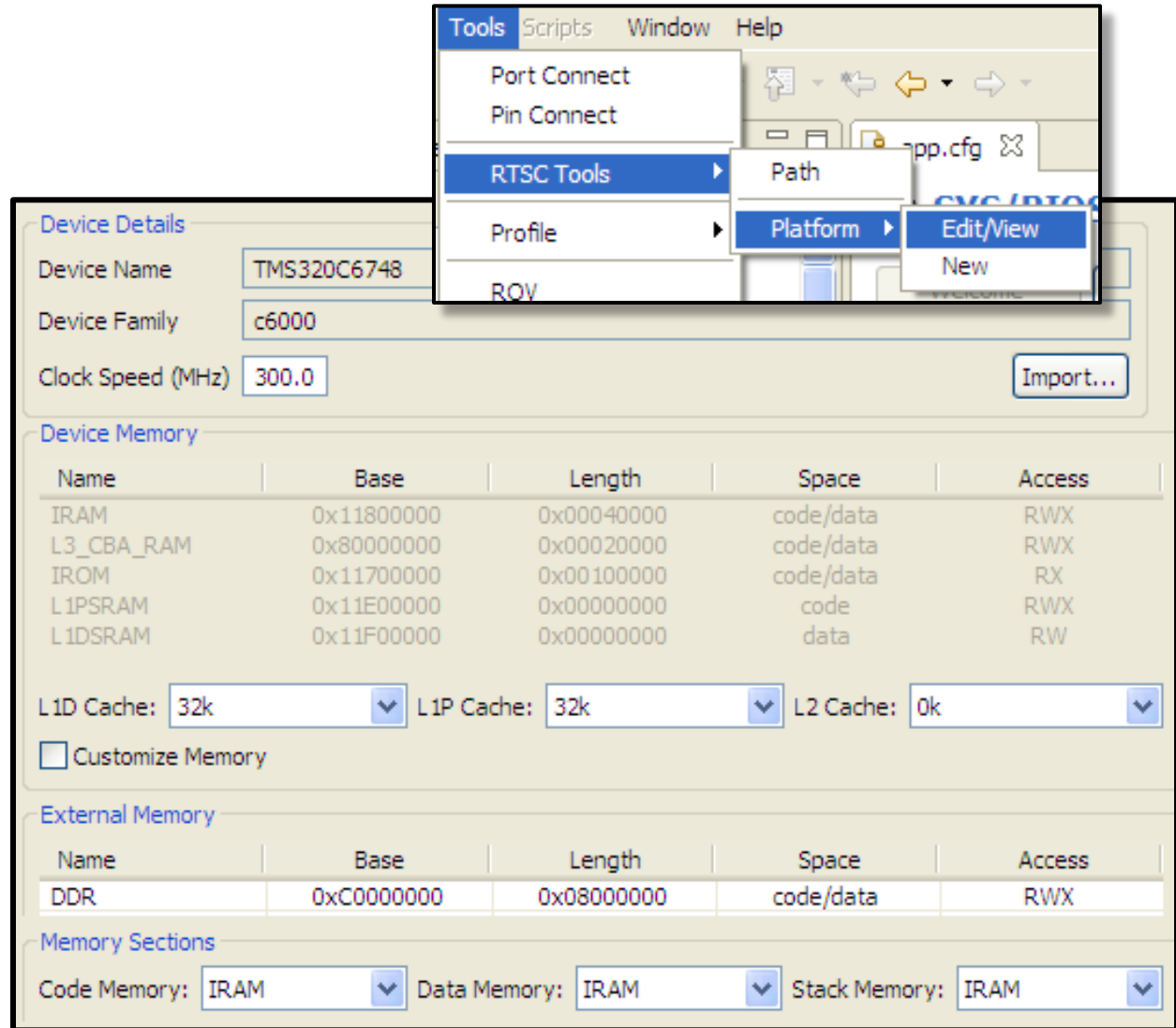- **BIOS Threads**
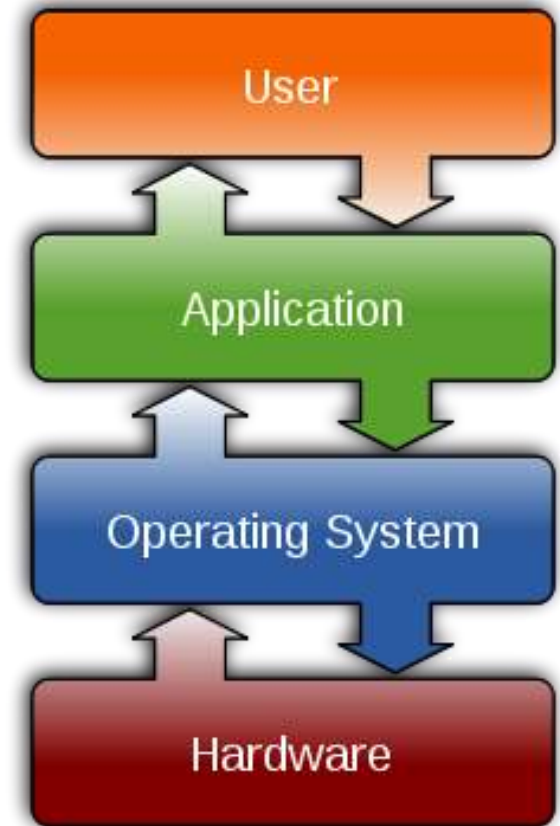
Platform (Memory Config)

Memory Config

◆ Create Internal Memory Segments (e.g. IRAM)

◆ Configure cache

◆ Define External Memory Segments

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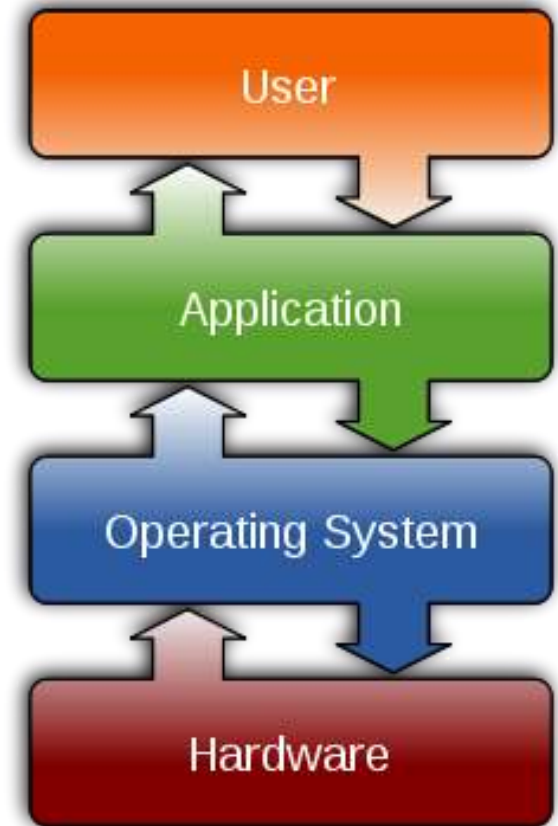
◆ Can link code, data and stack to any defined mem segment

Custom Platform

◆ Use "Import" button to copy "seed" platform and then customize

Tools | Scripts | Window | Help

Port Connect
Pin Connect
RTSC Tools ▶   Path
Profile ▶   Platform ▶   Edit/View
ROV   New

Device Details

Device Name	TMS320C6748
Device Family	c6000
Clock Speed (MHz)	300.0

Import...

Device Memory

Name	Base	Length	Space	Access
IRAM	0x11800000	0x00040000	code/data	RWX
L3_CBA_RAM	0x80000000	0x00020000	code/data	RWX
IROM	0x11700000	0x00100000	code/data	RX
L1PSRAM	0x11E00000	0x00000000	code	RWX
L1DSRAM	0x11F00000	0x00000000	data	RW

L1D Cache: 32k   L1P Cache: 32k   L2 Cache: 0k

☐ Customize Memory

External Memory

Name	Base	Length	Space	Access
DDR	0xC0000000	0x08000000	code/data	RWX

Memory Sections

Code Memory: IRAM   Data Memory: IRAM   Stack Memory: IRAM

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	
SYSBIOS6: SYS/BIOS 6.x Real-Time Operating System (previously DSP/BIOS v6)	**Get Software**	ACTIVE	

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**
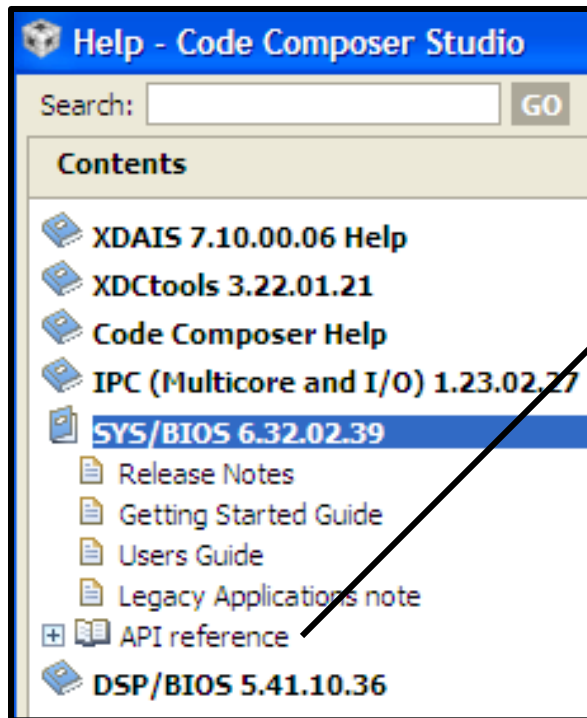
- User Guides
- API Reference (knl)
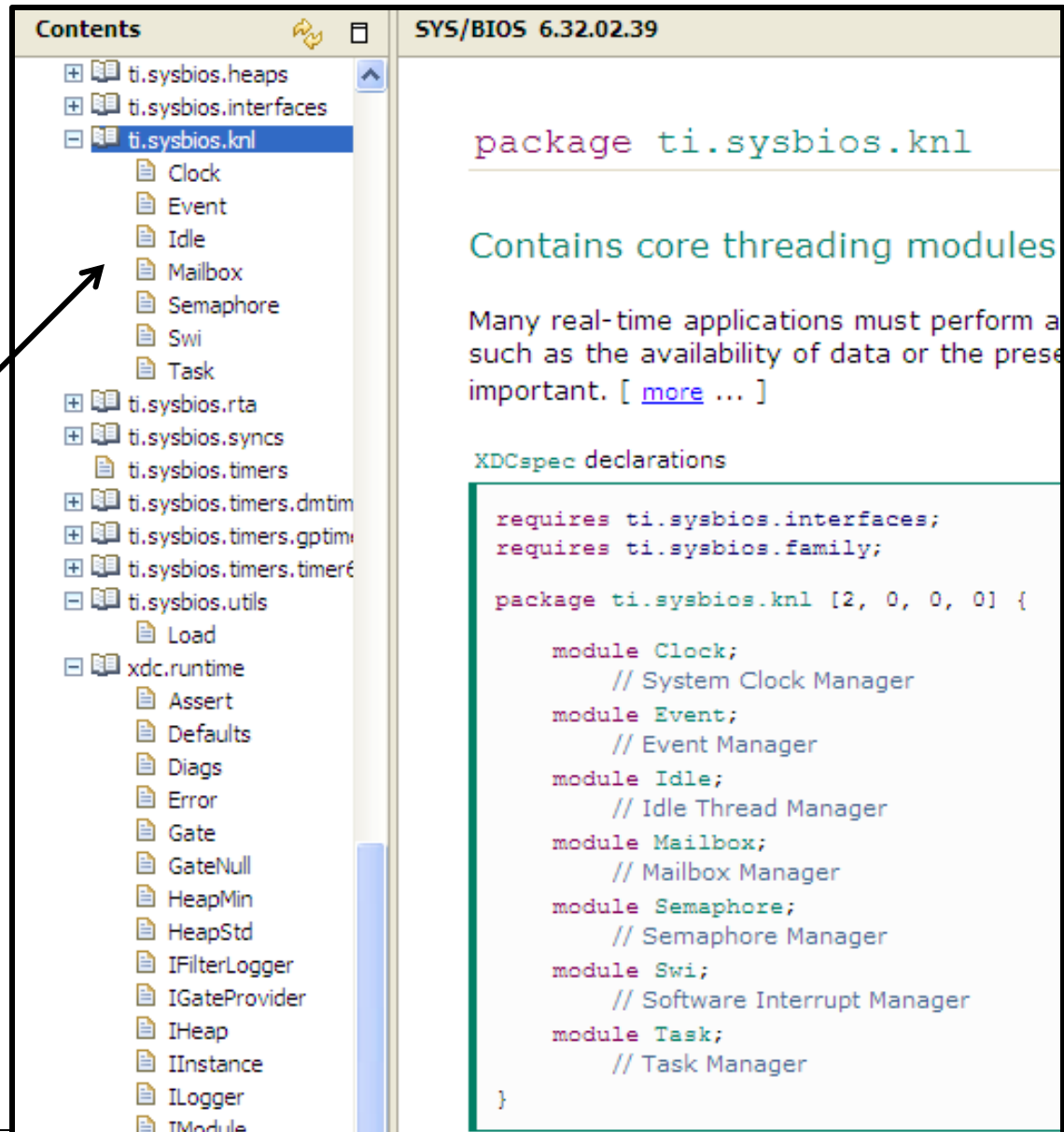
Download Latest Tools

◆ **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 Sotware Development Kits (DVSDK)
DSP Link and SysLink • 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)**
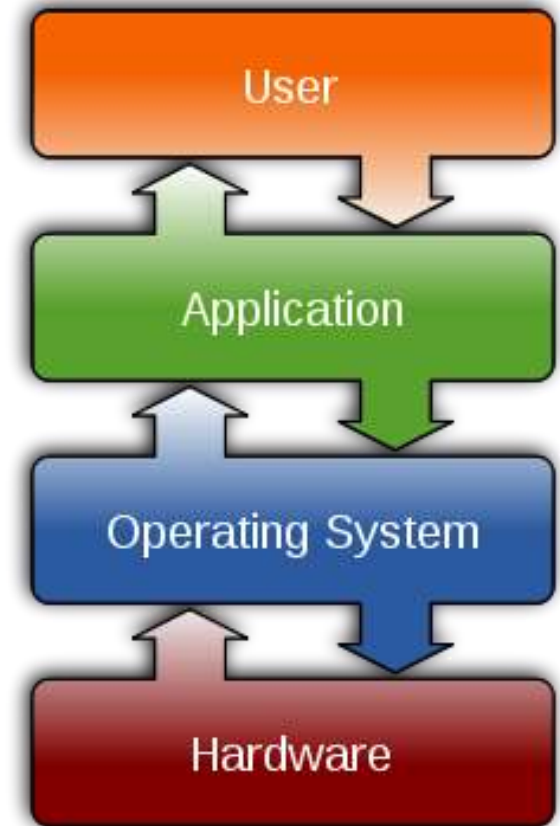
Hwi Scheduling

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
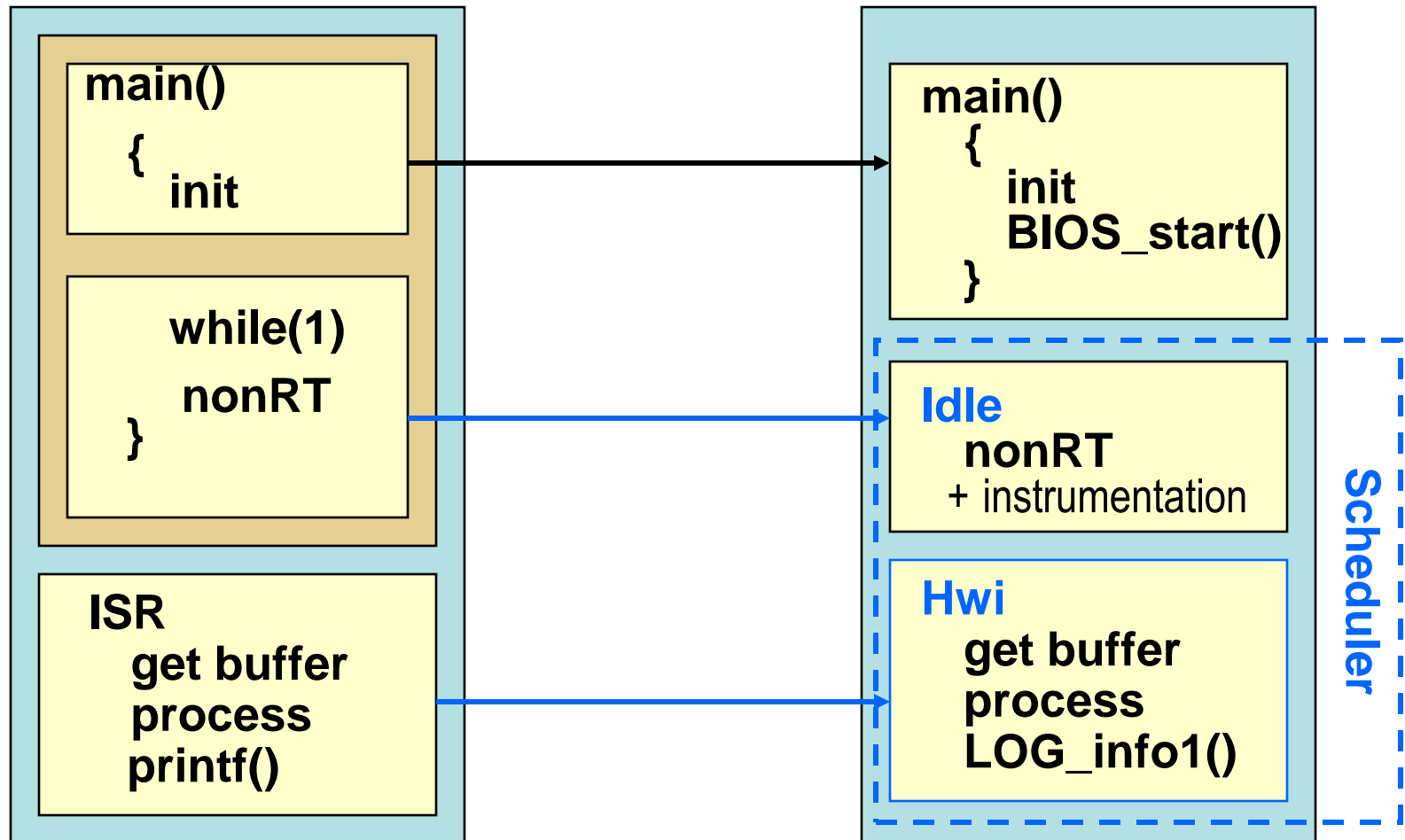
```
main()
{
    init
```

```
    while(1)
    nonRT
}
```

```
ISR
    get buffer
    process
    printf()
```

```
main()
{
    init
    BIOS_start()
}
```

```
Idle
    nonRT
    + instrumentation
```

```
Hwi
    get buffer
    process
    LOG_info1()
```

Scheduler

◆ **Idle** events run in sequence when no **Hwi**s 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 <u>Peripheral</u> (Ex: McASP)
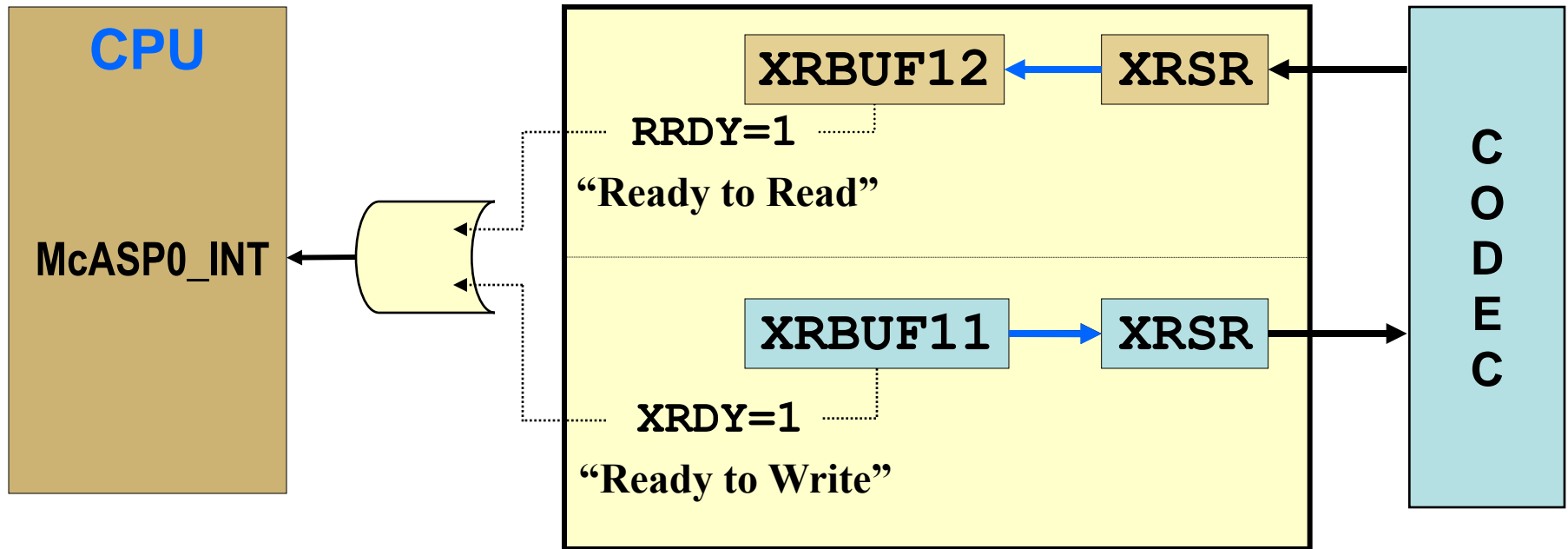
◆ Peripheral (e.g. McASP on C6748) causes an interrupt to the CPU to indicate "service required".

◆ 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 HWI$_5$

1 **Use Hwi module** *(Available Products)* **, insert new Hwi** *(Outline View)*
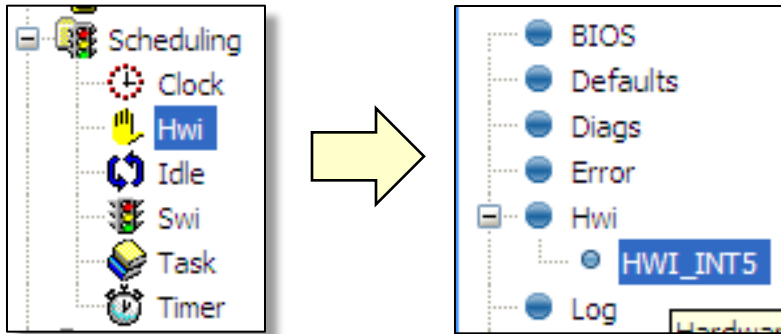
Remember, BIOS objects
can be created via the GUI,
script code or C code (dynamic)

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

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

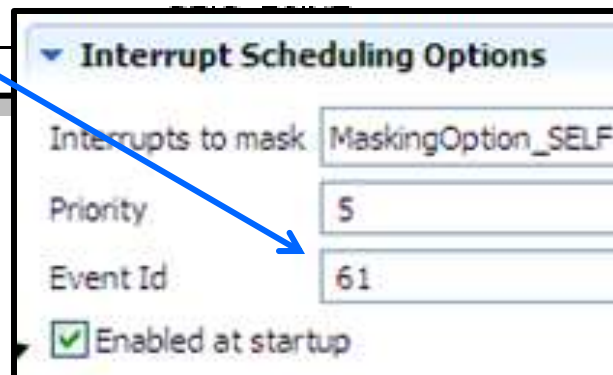To enable INT at startup, check the box

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
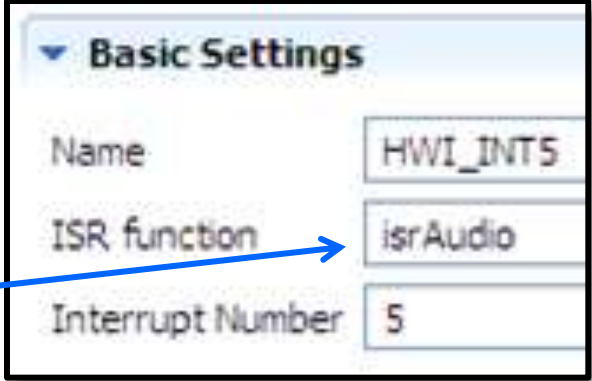
Priority   5

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

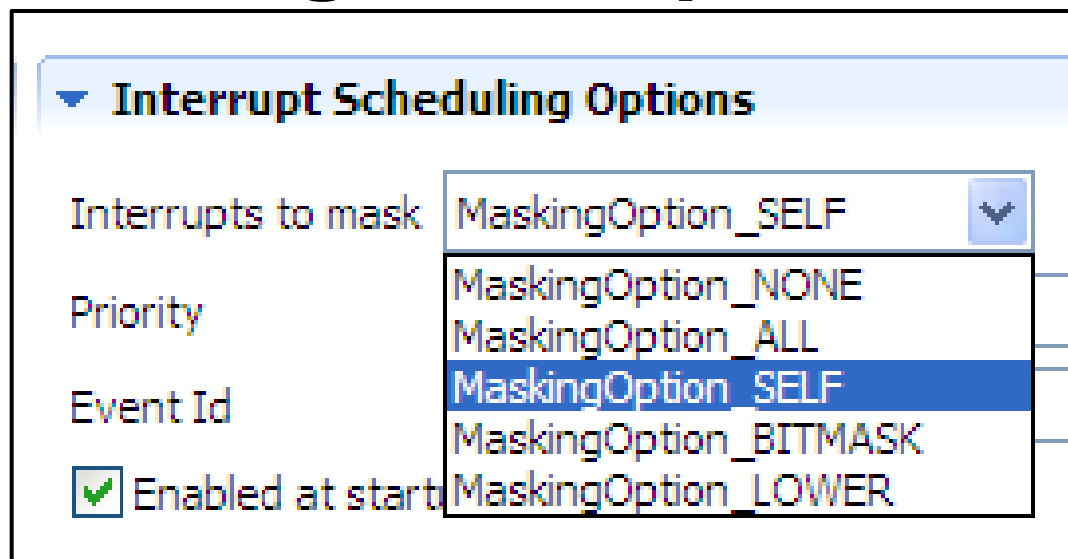| Name	HWI_INT5
ISR function	isrAudio
Interrupt Number	5

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

Interrupt Scheduling Options

Interrupts to mask	MaskingOption_SELF ▾
	MaskingOption_NONE
Priority	MaskingOption_ALL
	MaskingOption_SELF
Event Id	MaskingOption_BITMASK
	MaskingOption_LOWER
☑ Enabled at start	

◆ **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:

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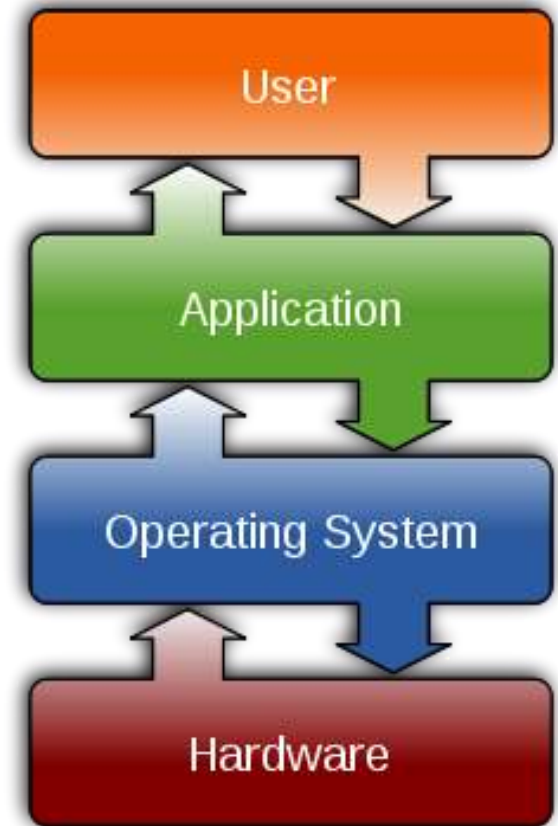
Swi Scheduling

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:

| INT ! | → | Hard R/T Process | → | **Post Swi** | → | Cleanup, RET | **Hwi** |

| **SWI Ready** | → | Continue Processing ... | **Swi** |

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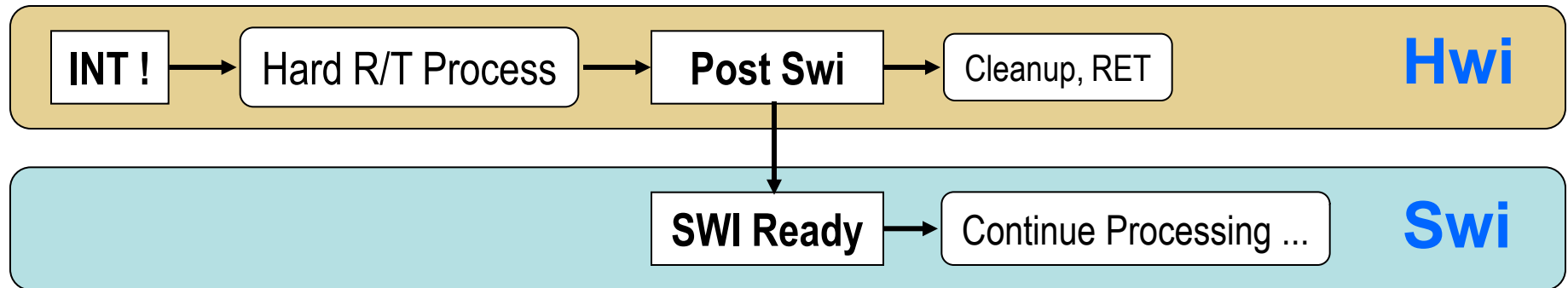
- ◆ **Fast response to INTs**
- ◆ **Min context switching**
- ◆ **High priority for CPU**
- ◆ **Limited # of Hwi possible**

isrAudio:

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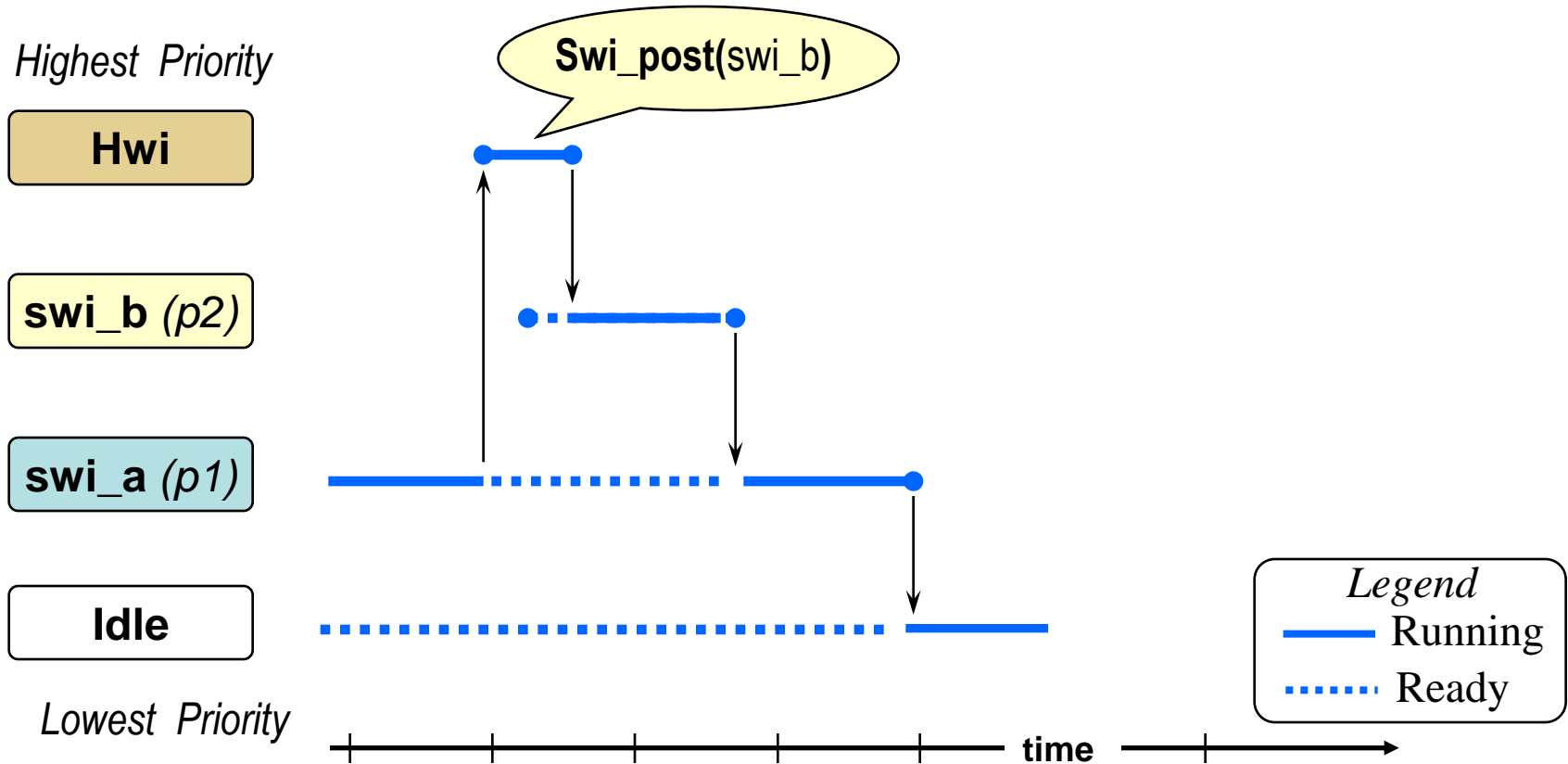
Swi

- ◆ **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
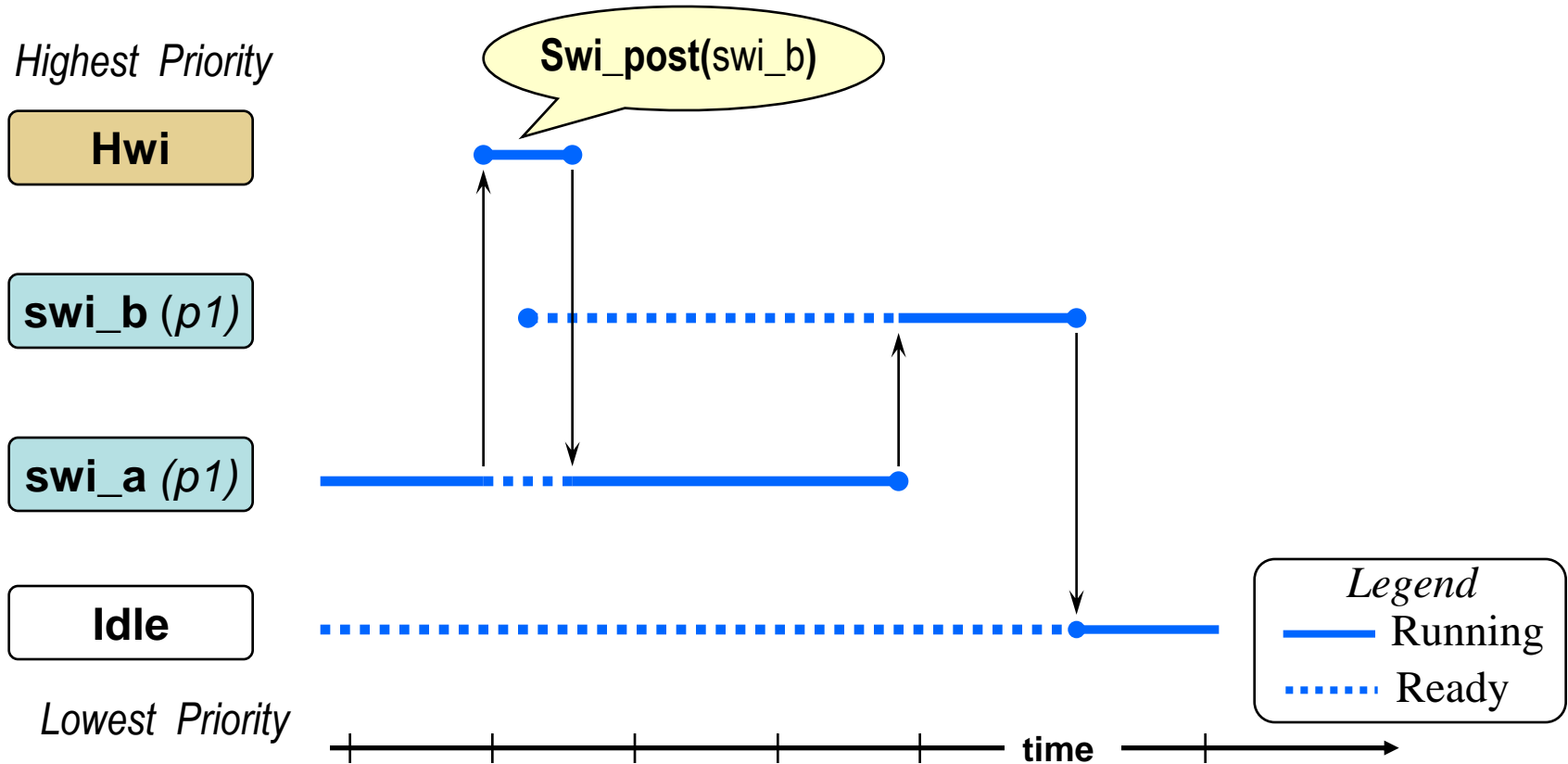
◆ Processes of same priority are scheduled first-in first-out (FIFO)

◆ Having threads at the SAME priority offers certain advantages – such as resource sharing (without conflicts)

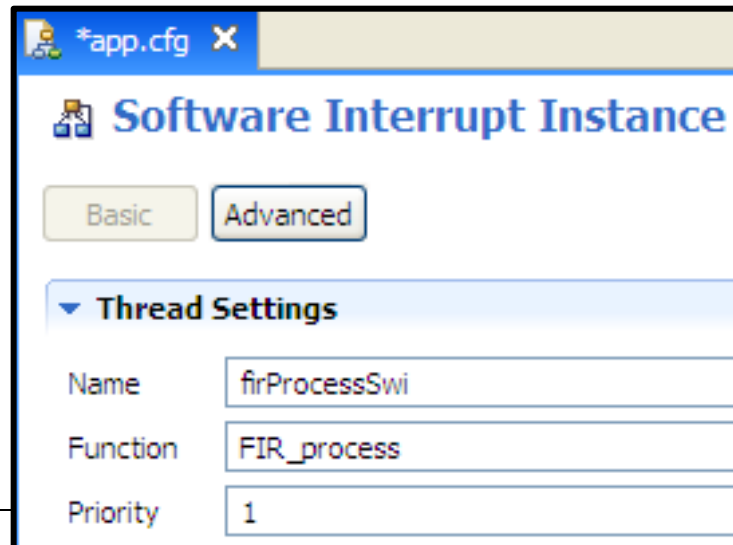How do you configure a SWI?

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)*

Scheduling
- Clock
- Hwi
- Idle
- **Swi**
- Task
- Timer

Swi
- firProcessSwi
- SysMin
- System
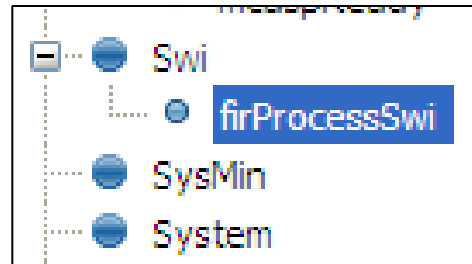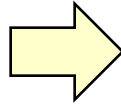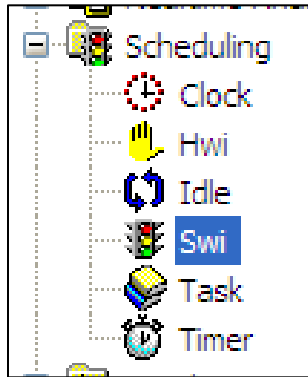
Remember, BIOS objects
can be created via the GUI,
script code or C code (dynamic)

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

*app.cfg ✕

Software Interrupt Instance

Basic | Advanced

▼ **Thread Settings**

Name	firProcessSwi
Function	FIR_process
Priority	1

Let's move on to Tasks...

SYS/BIOS Swi APIs

Other useful Swi APIs:

`Swi_inc()`	Post, increment count
`Swi_dec()`	Decrement count, post if 0
`Swi_or()`	Post, OR bit (signature)
`Swi_andn()`	ANDn bit, post if all posted
`Swi_getPri()`	Get any Swi Priority
`Swi_enable`	Global Swi enable
`Swi_disable()`	Global Swi disable
`Swi_restore()`	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)**
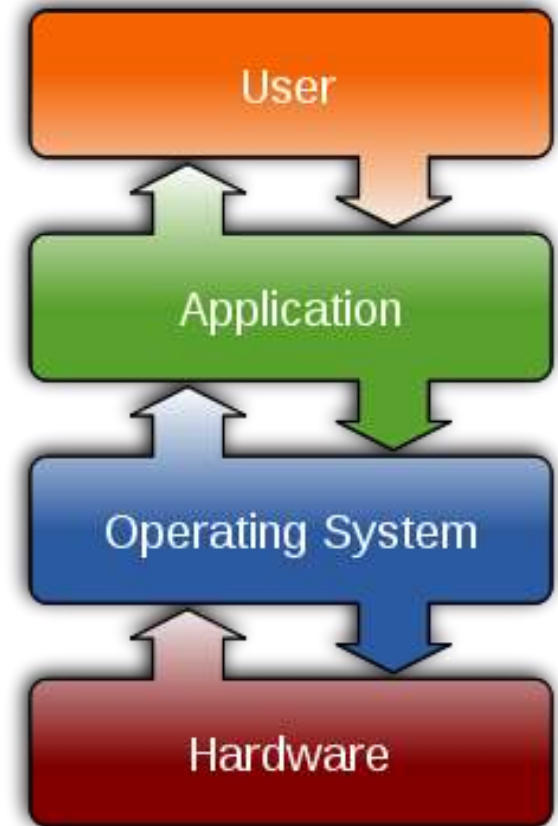
Task Scheduling

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 */                      ◆ Initialization (runs once only)

    while ('condition'){          ◆ Processing loop – (optional: cond)

        Semaphore_pend()          ◆ Wait for resources to be available

        /* Process  */            ◆ Perform desired algo work...

    }

/* Epilog */                      ◆ Shutdown (runs once - at most)

}
```

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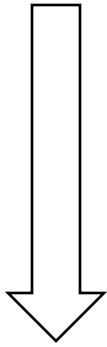◆ `Semaphore_pend()` *blocks* Task until semaphore (flag) is posted

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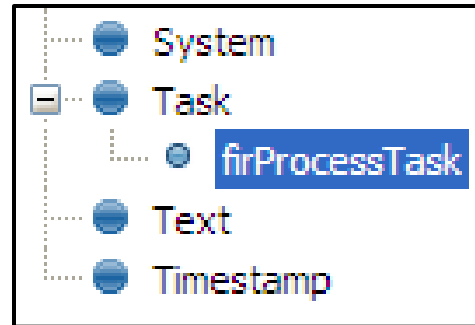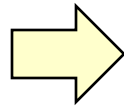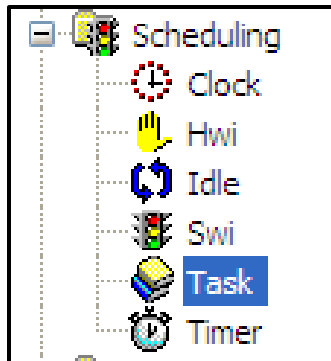
Remember, BIOS objects can be created via the GUI, script code or C code (dynamic)

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 thi
☑ Task is vital

Stack Control Options

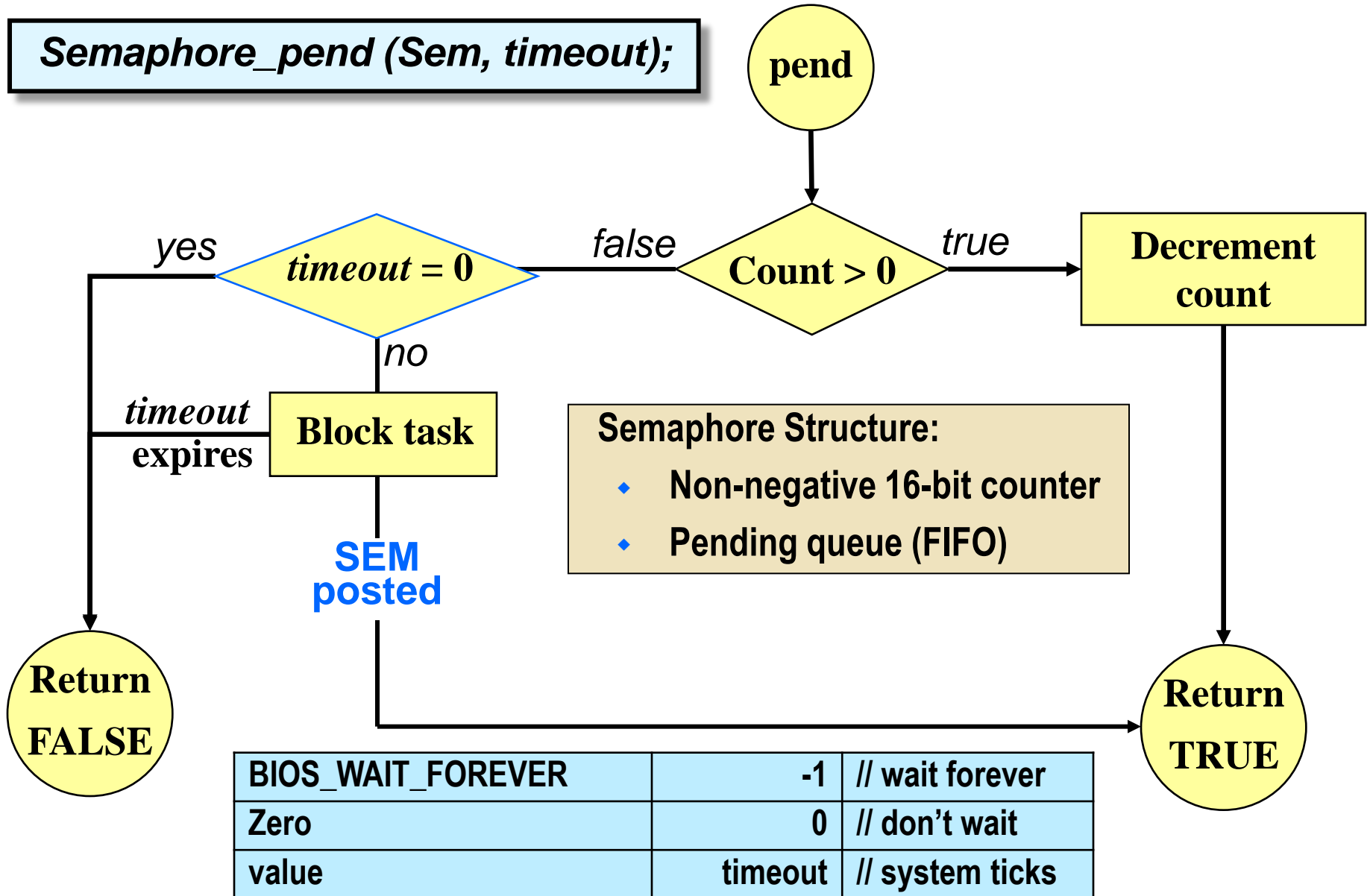|	
Stack size	2048

Outline

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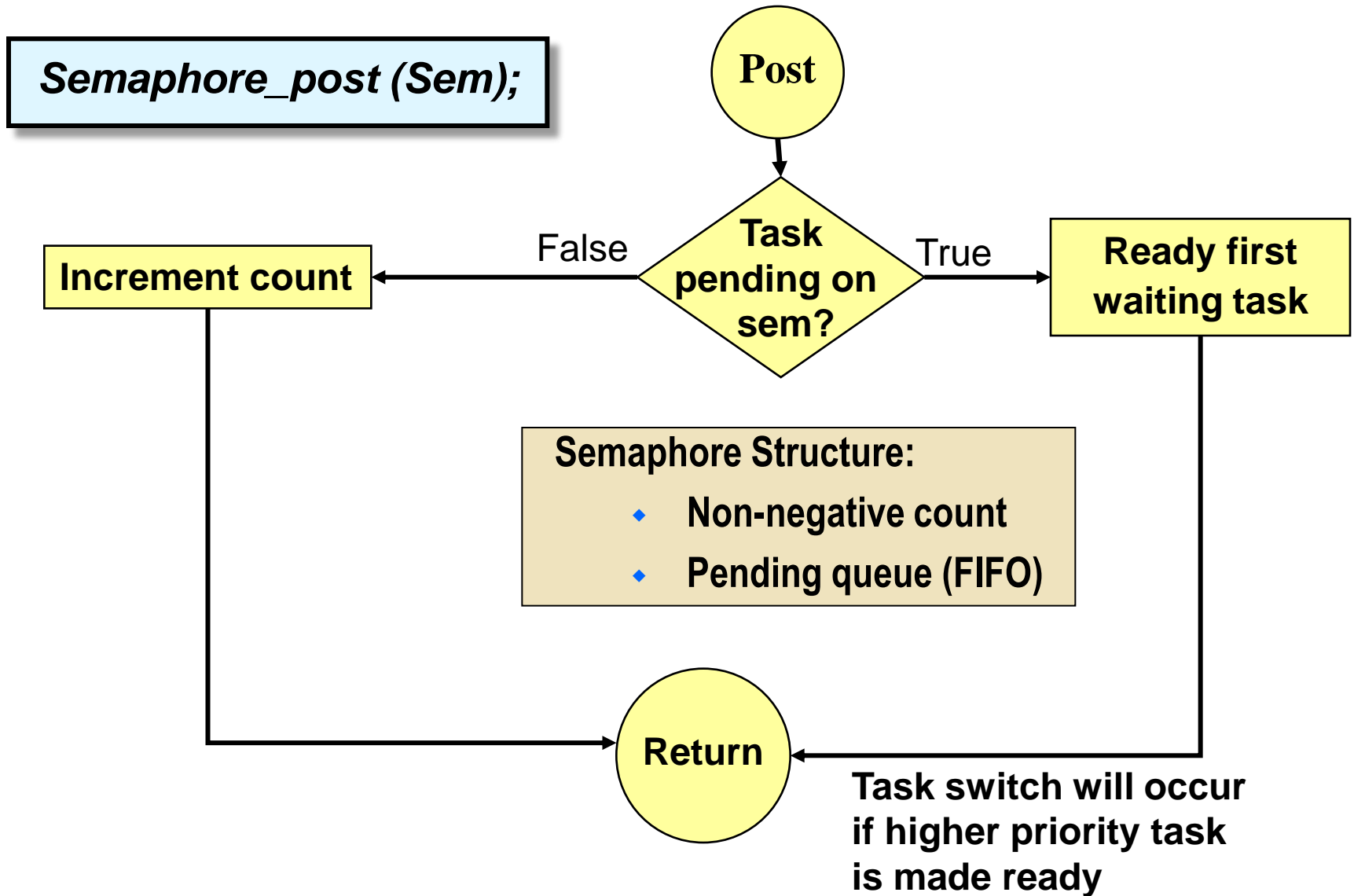 - **Tasks (TSK)**
  - **Semaphores (SEM)**
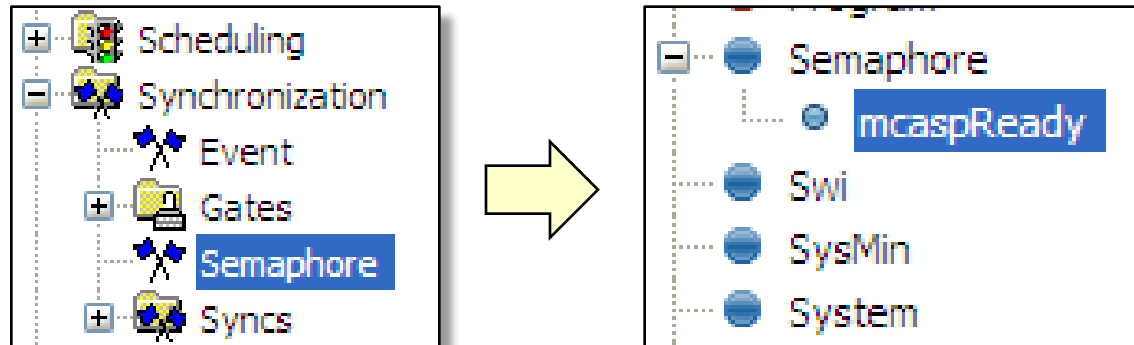
Semaphore Pend

Semaphore_pend (Sem, timeout);

pend

$timeout = 0$

— yes → Return FALSE

— no → Block task

— false → Count > 0

— true → Decrement count

timeout expires → Block task

SEM posted → Return TRUE

Semaphore Structure:

- Non-negative 16-bit counter
- Pending queue (FIFO)

Return FALSE

Return TRUE

BIOS_WAIT_FOREVER	-1	// wait forever
Zero	0	// don't wait
value	timeout	// system ticks

Semaphore Post

Post

Semaphore_post (Sem);

Increment count ← False — **Task pending on sem?** — True → **Ready first waiting task**

Semaphore Structure:
- Non-negative count
- Pending queue (FIFO)

Return

Task switch will occur if higher priority task is made ready

How do you configure a Semaphore?

Configuring a _Semaphore_ – Statically via GUI

Example: Create mcaspReady, counting

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

(2) **Configure Semaphore – Object name, initial count, type:**

SYS/BIOS Semaphore/Task APIs

Other useful Semaphore APIs:

`Semaphore_getCount()`	Get semaphore count

Other useful Task APIs:

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

Questions?