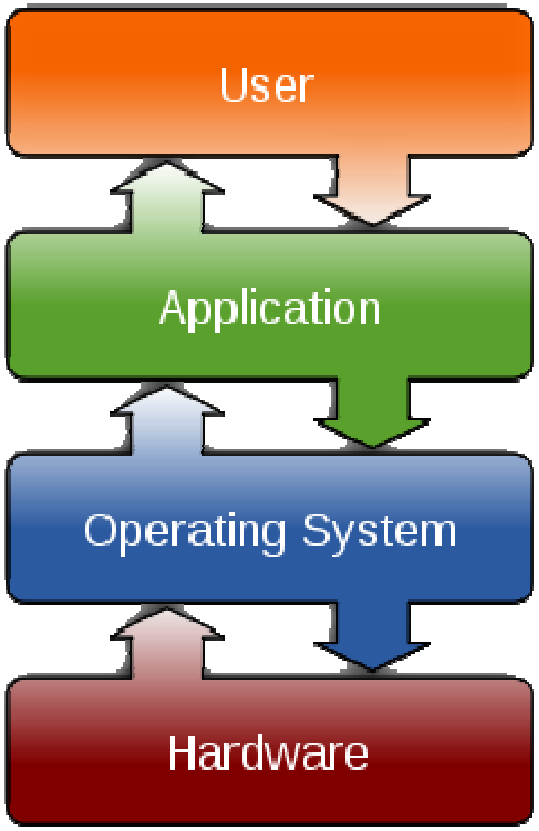
Introduction to SYS/BIOS

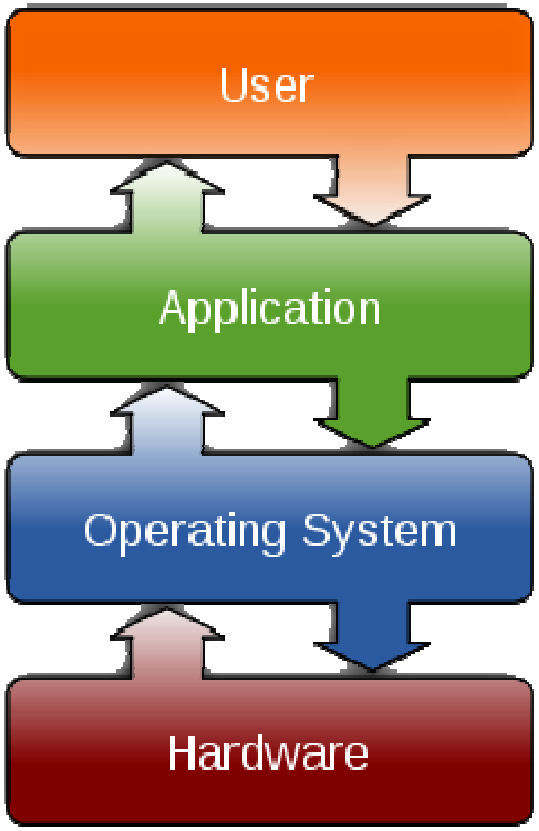
Outline

- **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 Information
- **BIOS Threads**

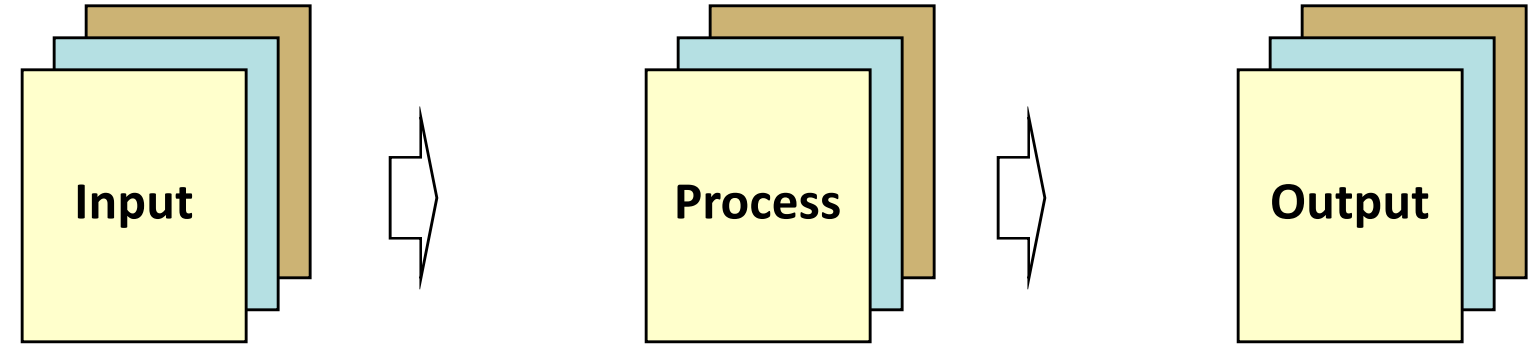


Outline

- 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 Information
- BIOS Threads

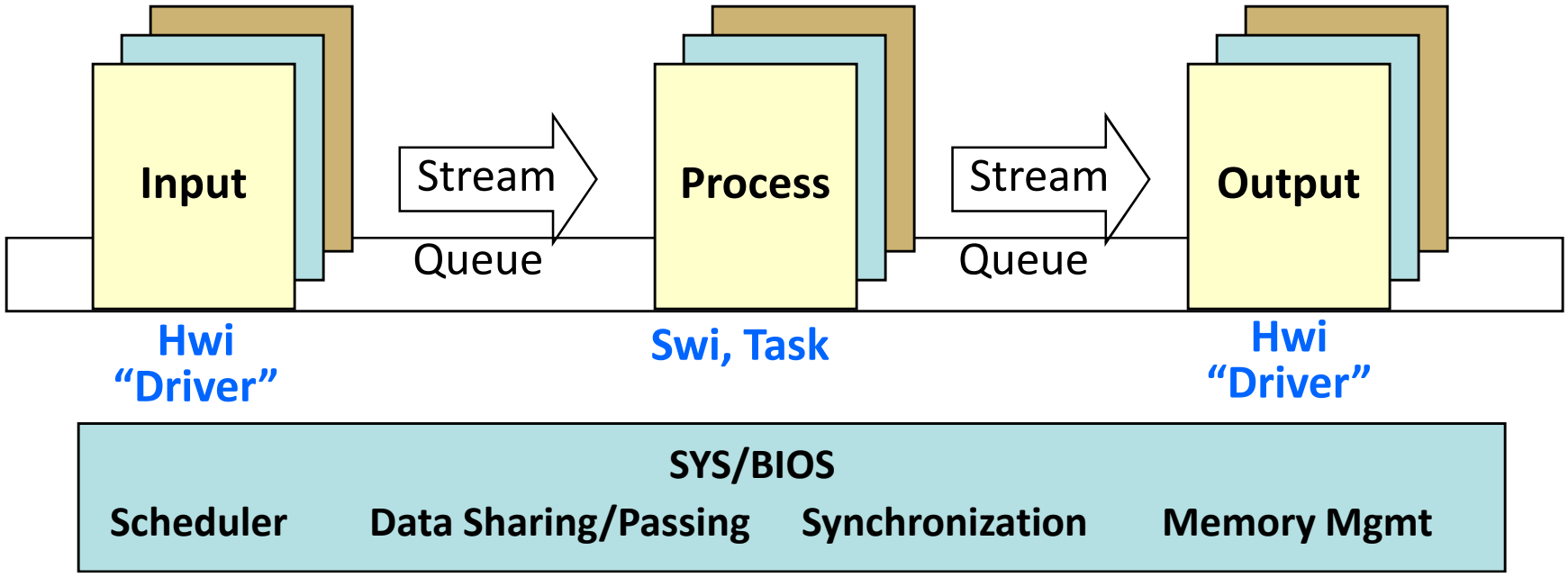


Need for an Operating System



- Simple system: single I-P-O is easy to manage
- As system complexity increases (multiple threads):
  - Can they all meet real time ?
  - Priorities of threads/algos ?
  - Synchronization of events?
  - Data sharing/passing ?
- Two 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?

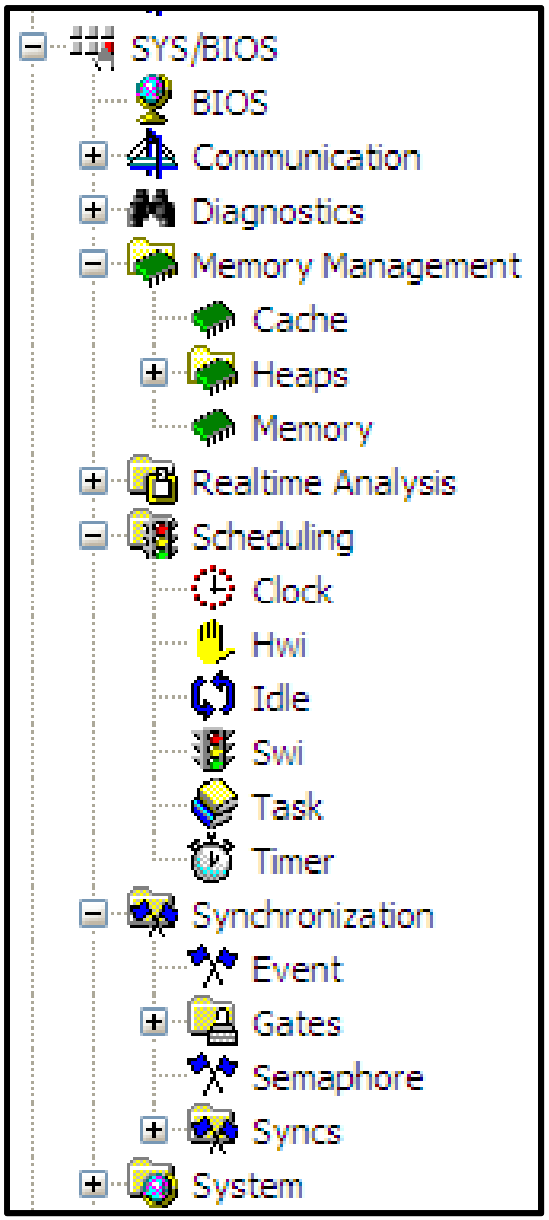
SYS/BIOS Overview



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 – Include only what is needed**
- **API - 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**

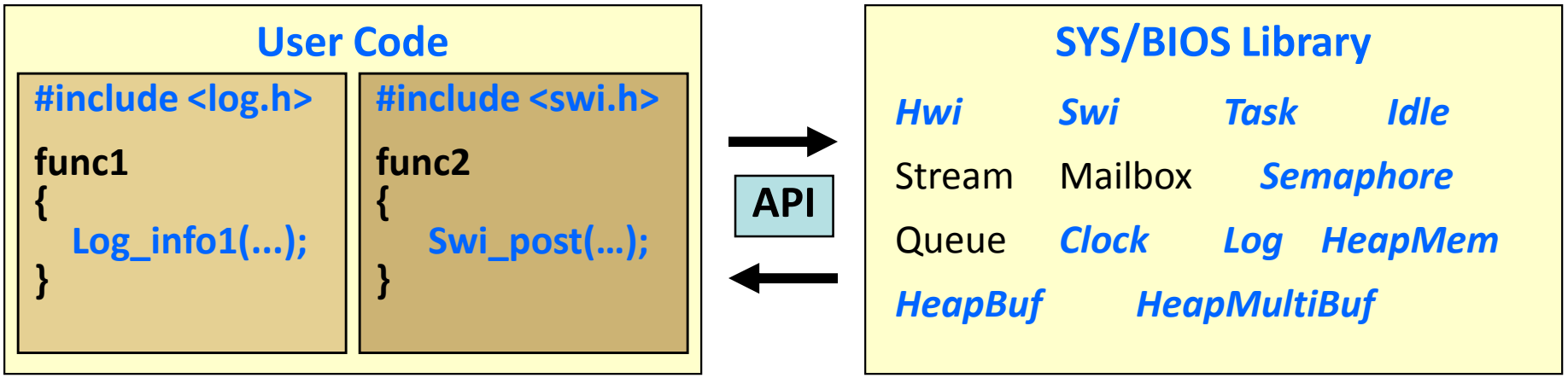
SYS/BIOS Modules & Services



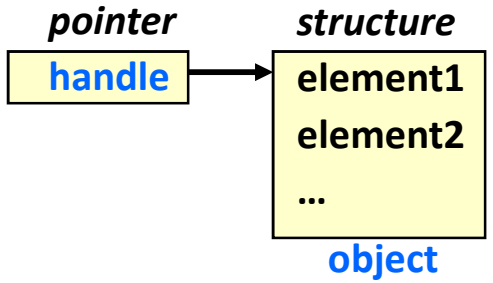
BIOS Configuration

- **Memory Management**
  - **Cache**
  - **Heaps**
- **Realtime Analysis**
  - **Logs**
  - **Loads**
  - **Execution Graph**
- **Scheduling**
  - **All thread types**
- **Synchronization**
  - **Events**
  - **Gates**
  - **Semaphores**

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:

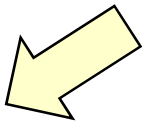
Real-time System

- *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

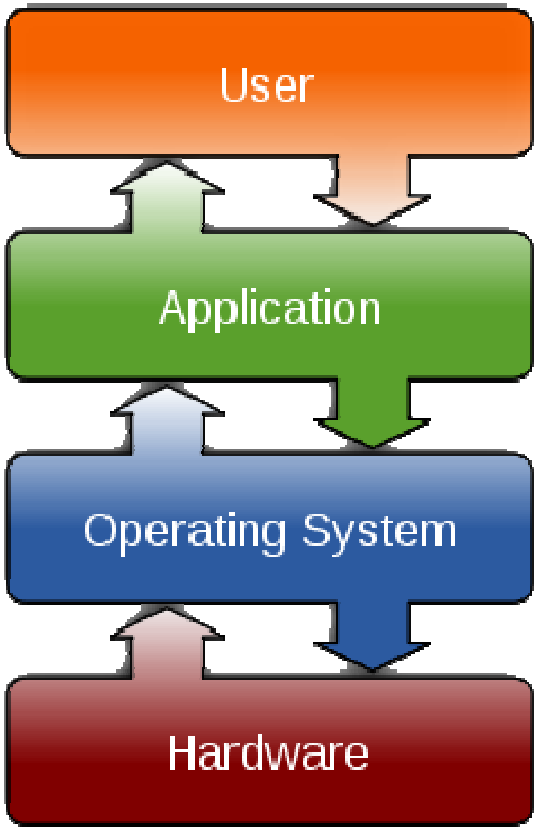
- *Application Programming Interface provides methods for interacting with library routines and data objects*

Comparing RTOS and GP/OS

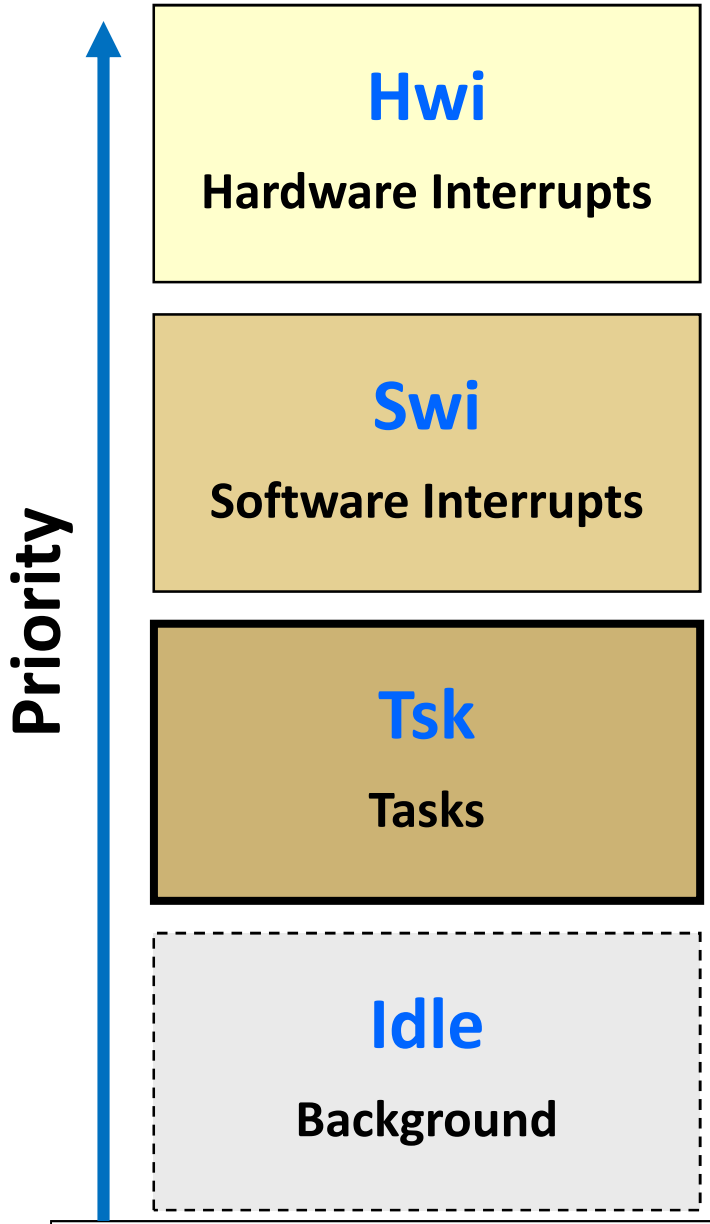
	GP/OS (e.g. Linux)	RTOS (e.g. SYS/BIOS)
Scope	General	Specific
Size	Large: 5M-50M	Small: 5K-50K
Event response	1ms to .1ms	100 – 10 ns
File management	FAT, etc.	FatFS
Dynamic Memory	Yes	Yes
Threads	Processes, pThreads, Ints	Hwi, Swi, Task, Idle
Scheduler	Time Slicing	Preemption
Host Processor	ARM, x86,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



SYS/BIOS Thread Types



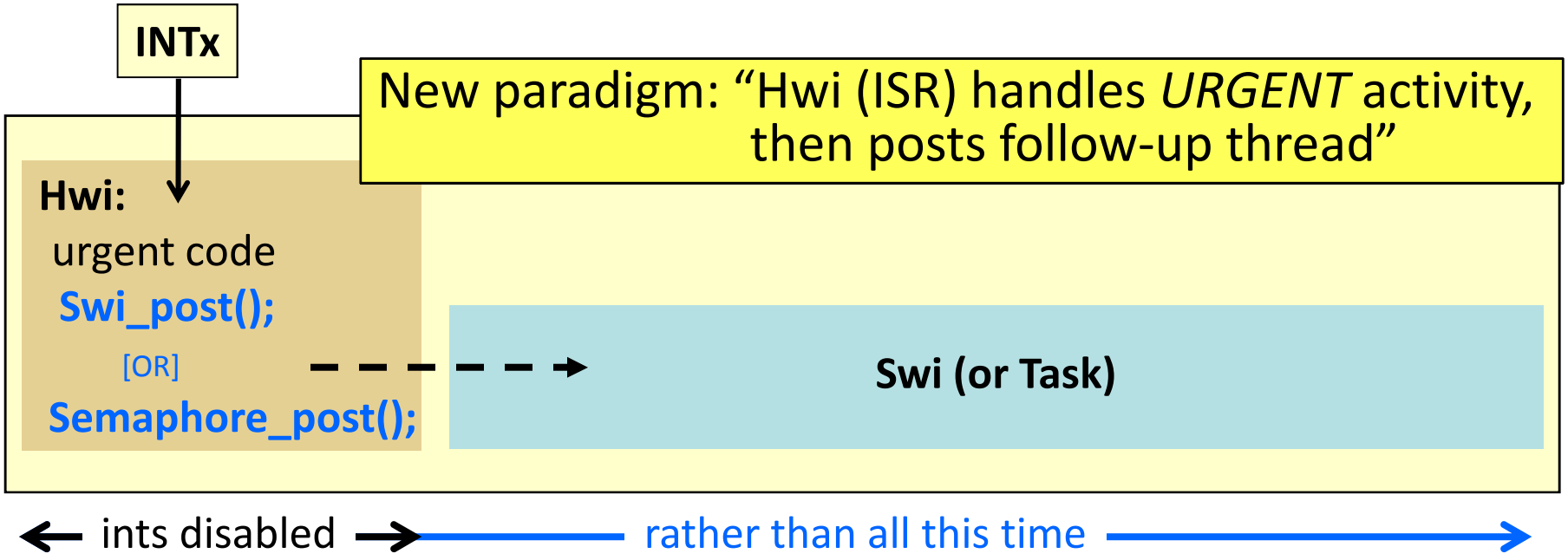
- Implements 'urgent' part of real-time event
- Hardware interrupt triggers ISRs to run
- Priorities set by hardware

- Performs HWI *follow-up* activity
- *Posted* by software
- Periodic (Clock) functions are prioritized as Swi processes
- Up to 32 priority levels

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

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

Hwi Signaling Swi/Task



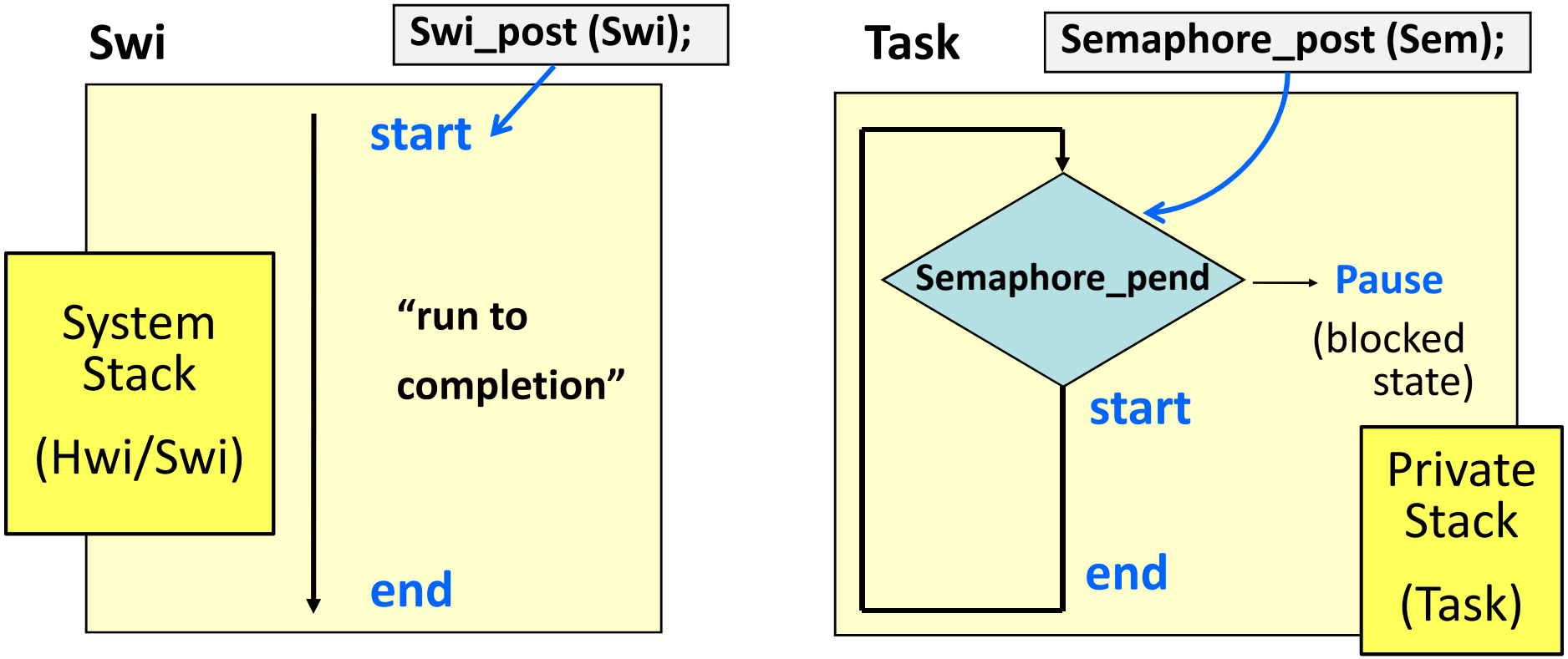
Hwi

- 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 and Tasks

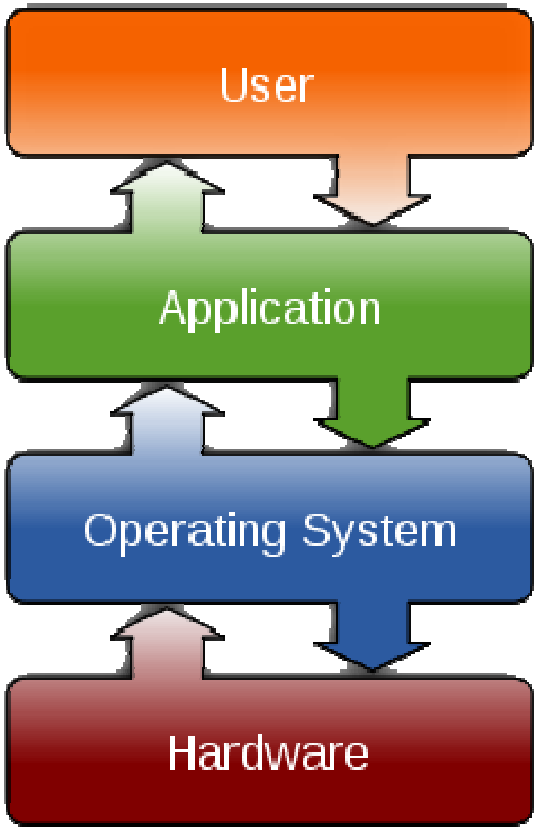


- Similar to hardware interrupt, but triggered when posted
- All Swi activities share system software stack with Hwi activities.

- 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 Information
- BIOS Threads



Thread (Object) Creation in BIOS

Users can create threads (BIOS resources or "objects"):

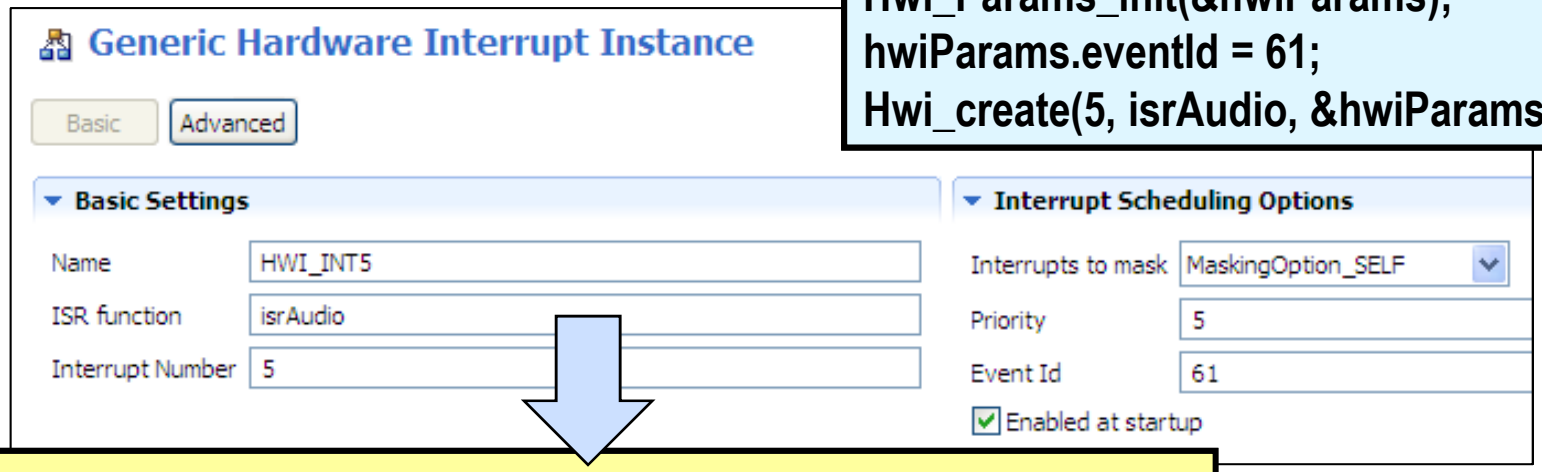
- Statically (via the GUI or .cfg script)
- Dynamically (via C code)
- BIOS doesn't care – but you might…

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)

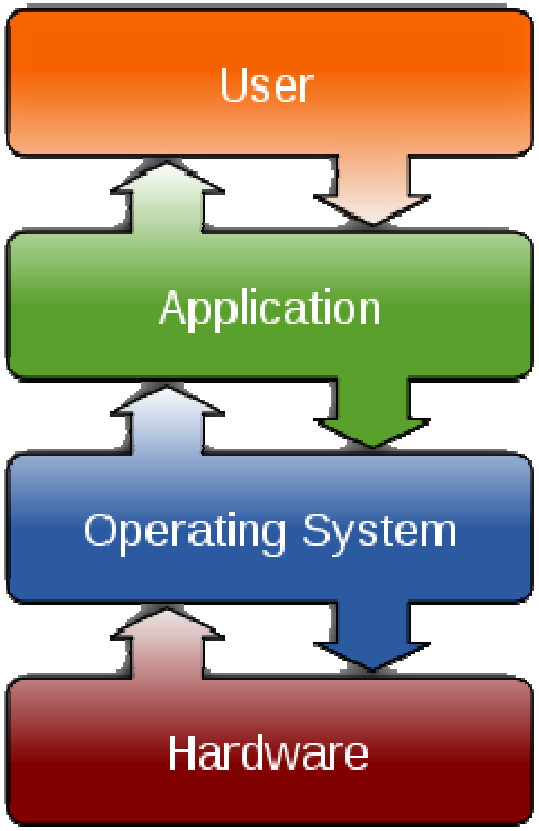



```
var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');
var hwiParams = new Hwi.Params();
hwiParams.eventId = 61;
Hwi.create(5, "&isrAudio", hwiParams);
```

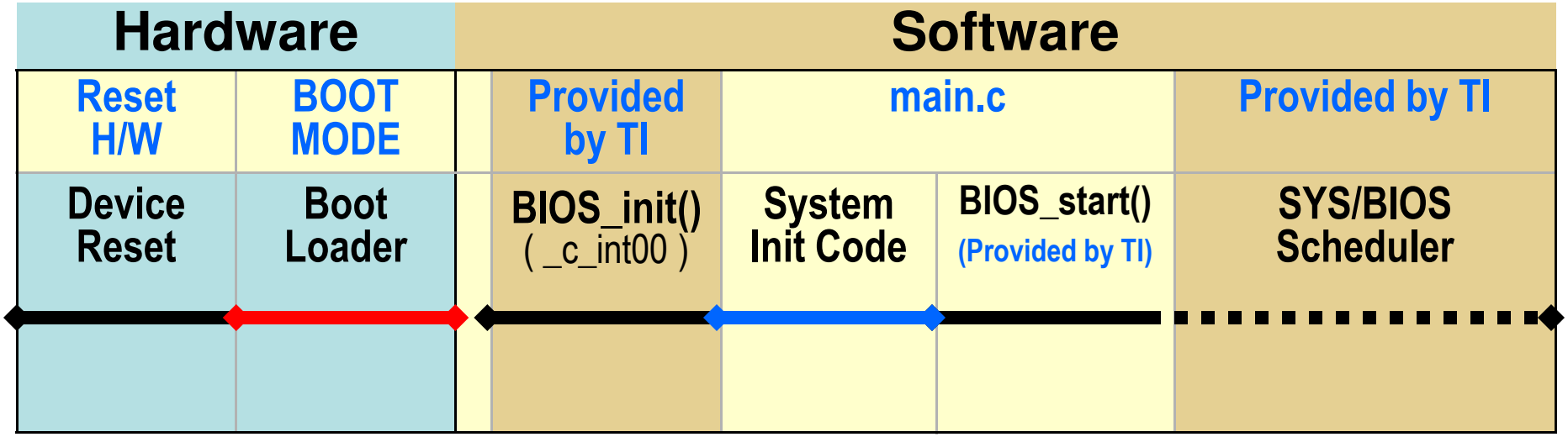
app.cfg

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 Information
- BIOS Threads



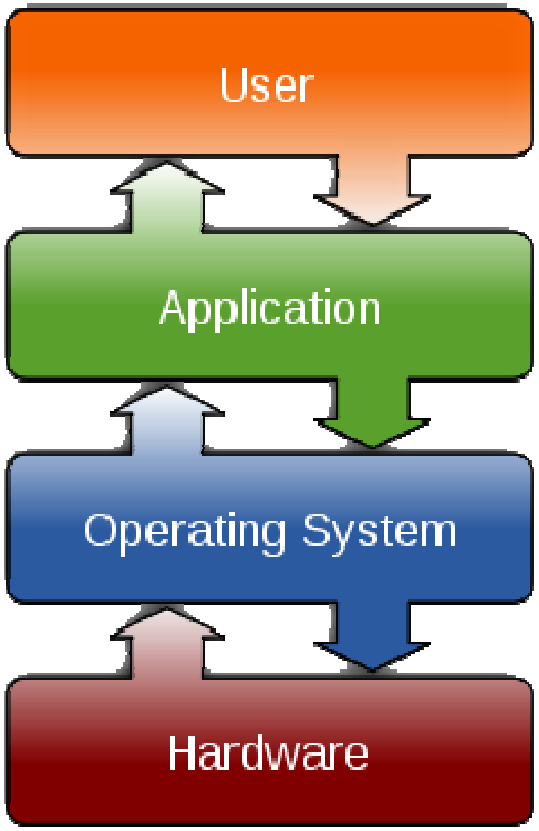
System Timeline



- RESET: Device is reset, then jumps to bootloader or code entry point (`c_int00`)
- BOOT MODE runs bootloader (if applicable)
- `BIOS_init()` configures 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

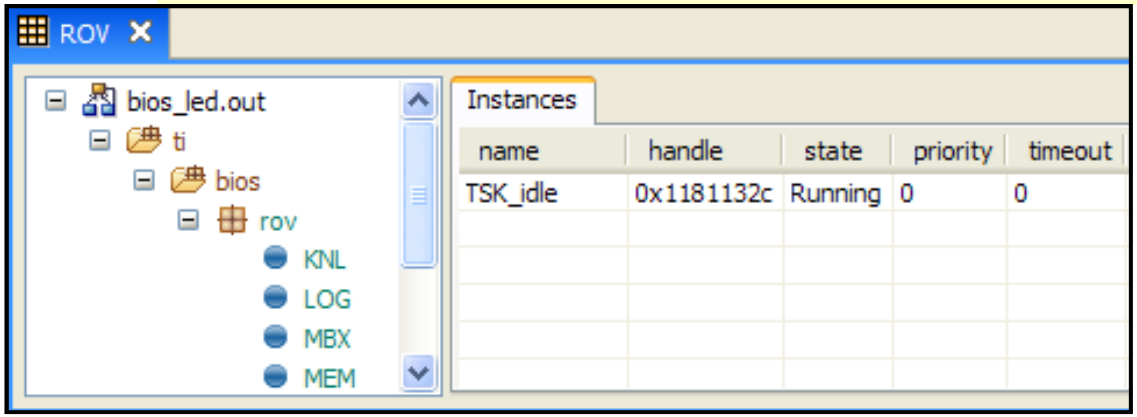
Outline

- 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 Information
- BIOS Threads



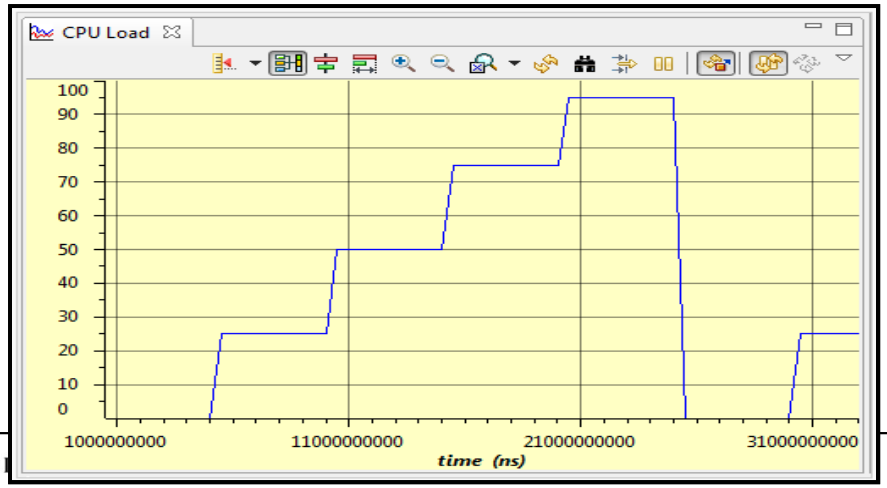
Built-in Real-Time Analysis Tools

- 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()`

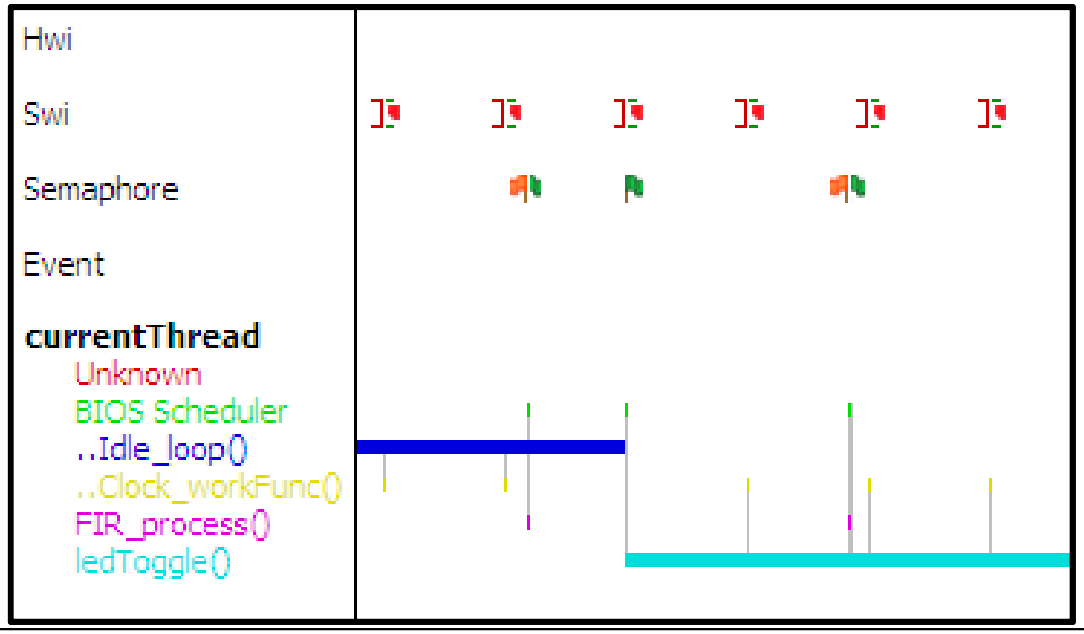
Raw Logs

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);
```

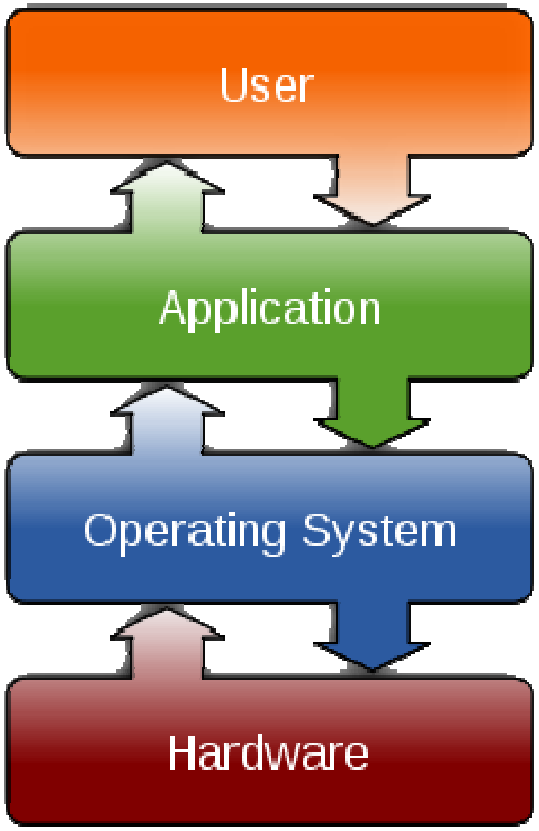
Execution Graph

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



Outline

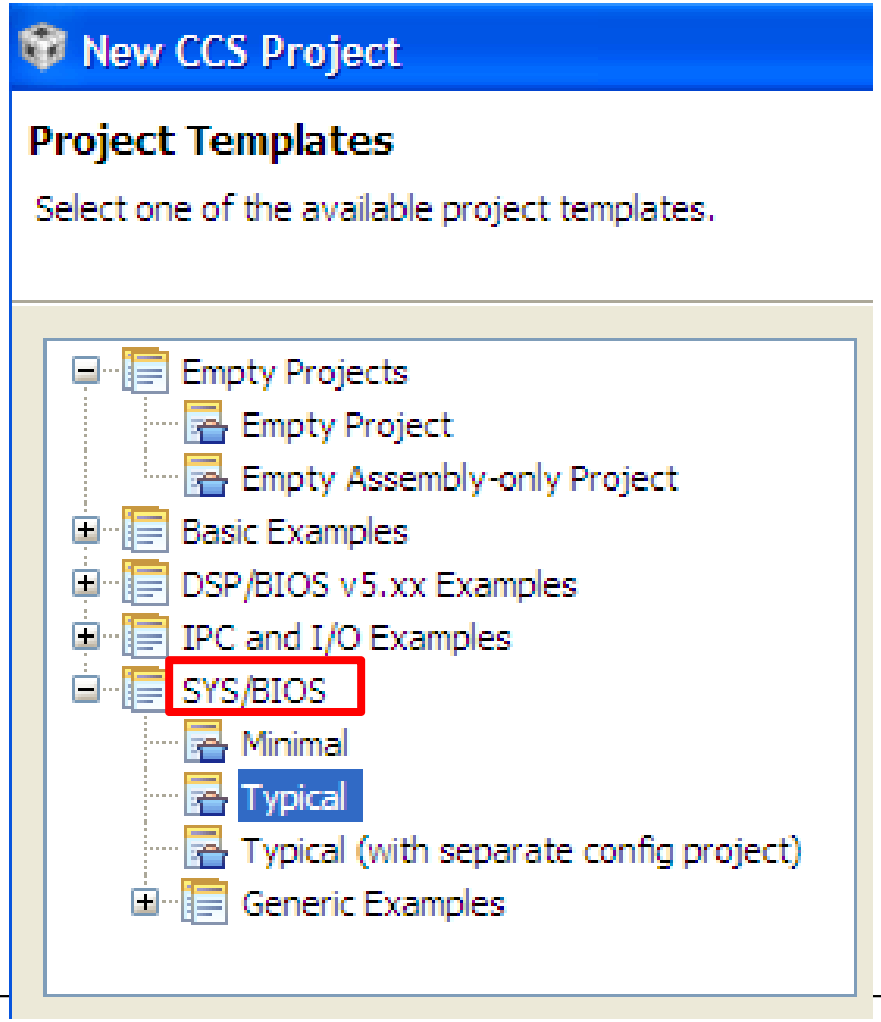
- Intro to SYS/BIOS
  - Overview
  - Threads and Scheduling
  - Creating a BIOS Thread
  - System Timeline
  - Real-Time Analysis Tools
  - **Create A New Project**
  - BIOS Configuration (.CFG)
  - Platforms
  - For More Information
- BIOS Threads



Building a NEW SYS/BIOS Project

- **Create CCS Project (as normal), then click:**

- **Select a SYS/BIOS Example:**

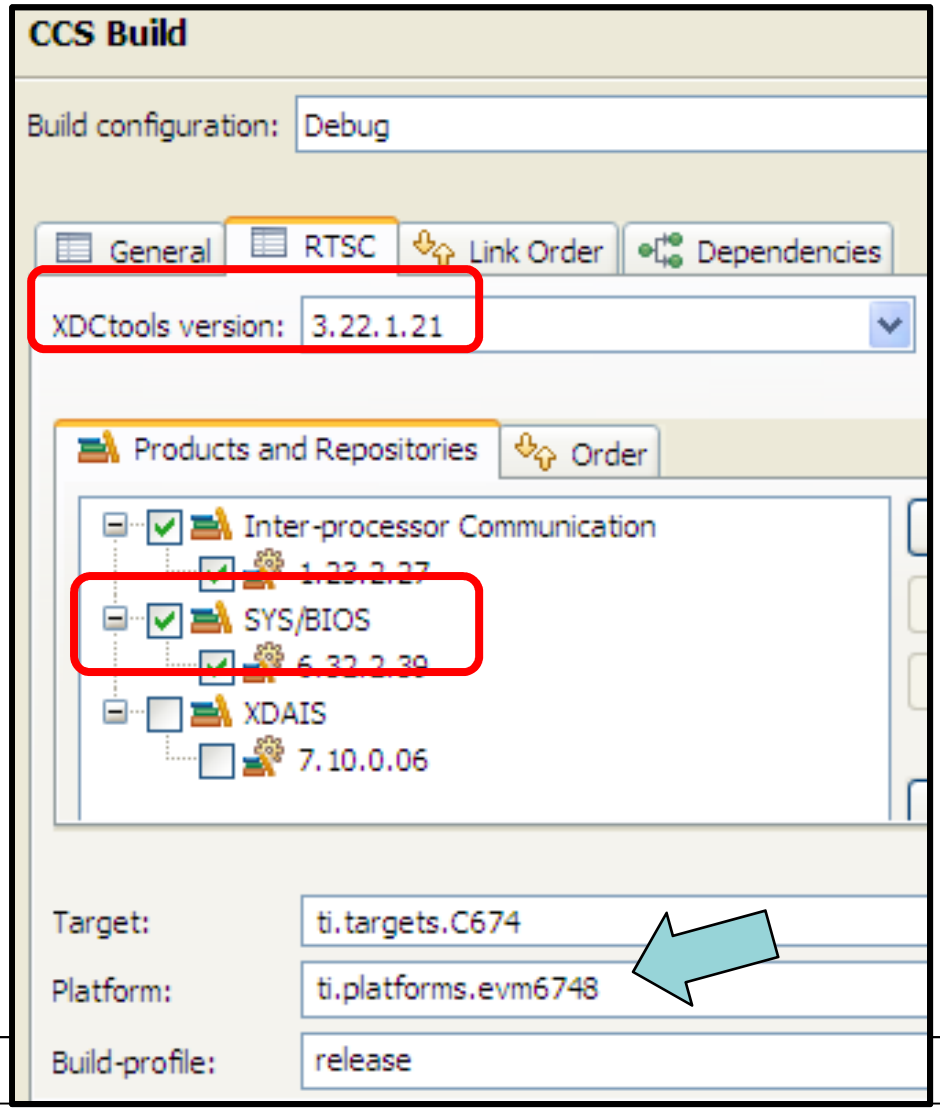


What is in the project created by *Typical*?

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

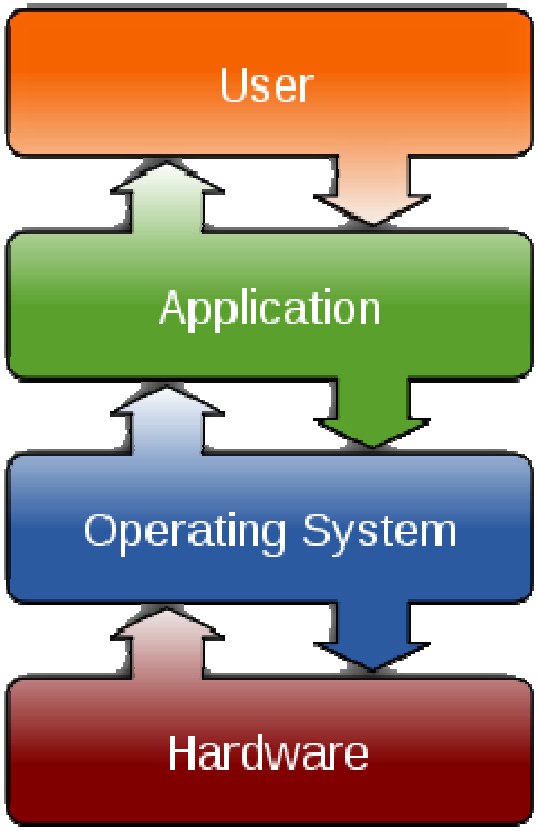
SYS/BIOS Project Settings

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



Outline

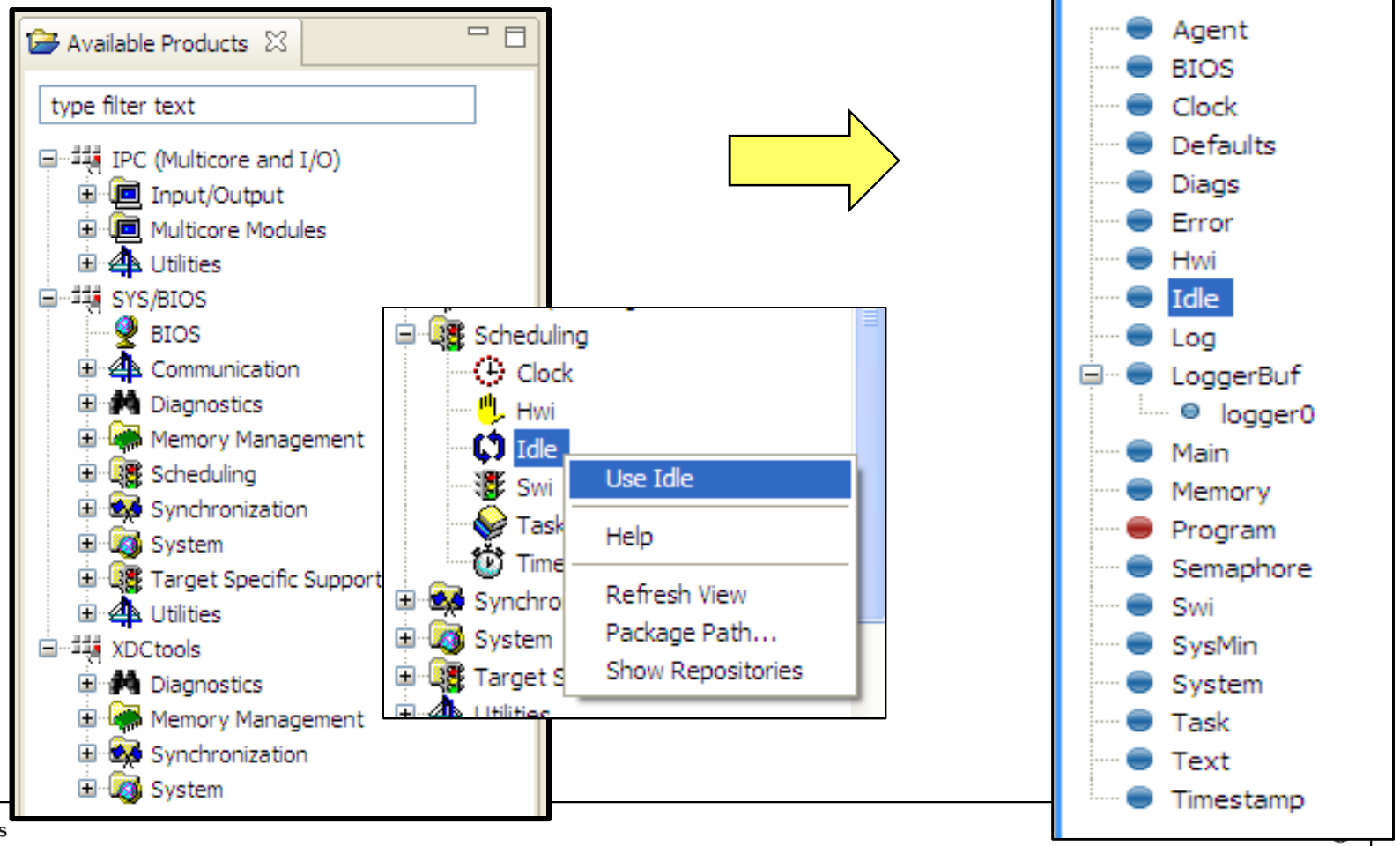
- Intro to SYS/BIOS
  - Overview
  - Threads and Scheduling
  - Creating a BIOS Thread
  - System Timeline
  - Real-Time Analysis Tools
  - Create A New Project
  - BIOS Configuration (.CFG)
  - Platforms
  - For More Information
- BIOS Threads



Static BIOS Configuration

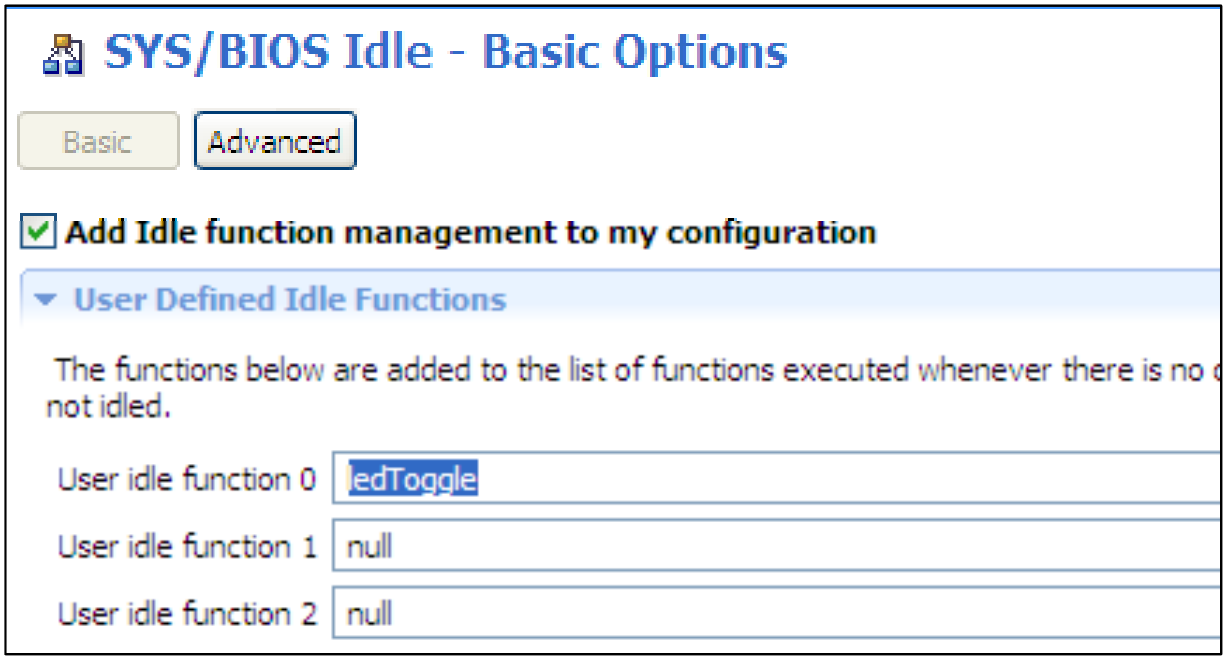
Users interact with the CFG file via the GUI – XGCONF:

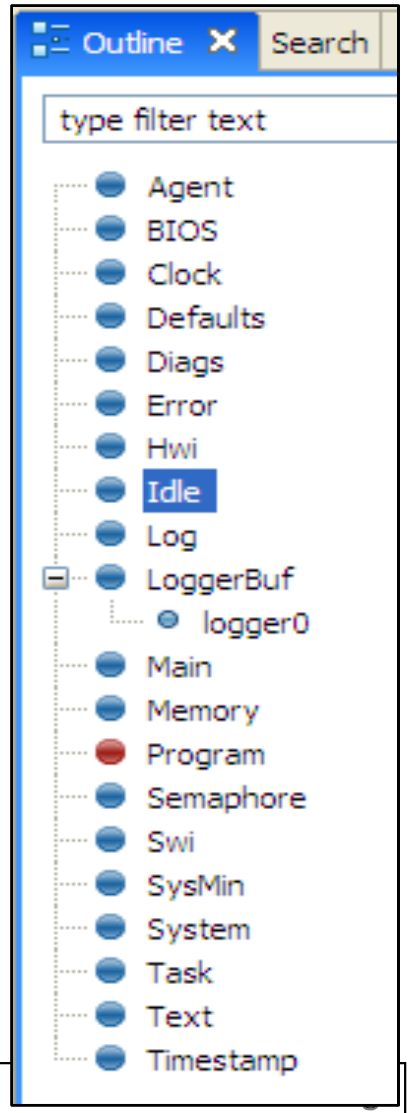
- XGCONF shows *Available Products*; Right-click and select *Use Mod.*
- *Mod* shows up in *Outline* view. Right-click and select *Add New.*
- All graphical changes in GUI are displayed in .cfg source code.



Static Config – .CFG Files

- **Users interact with the CFG file via the GUI – XGCONF**
  - When you *Add New*, a dialogue box is provided to set up parameters.
  - This window provides two views:
    - *Basic*
    - *Advanced*





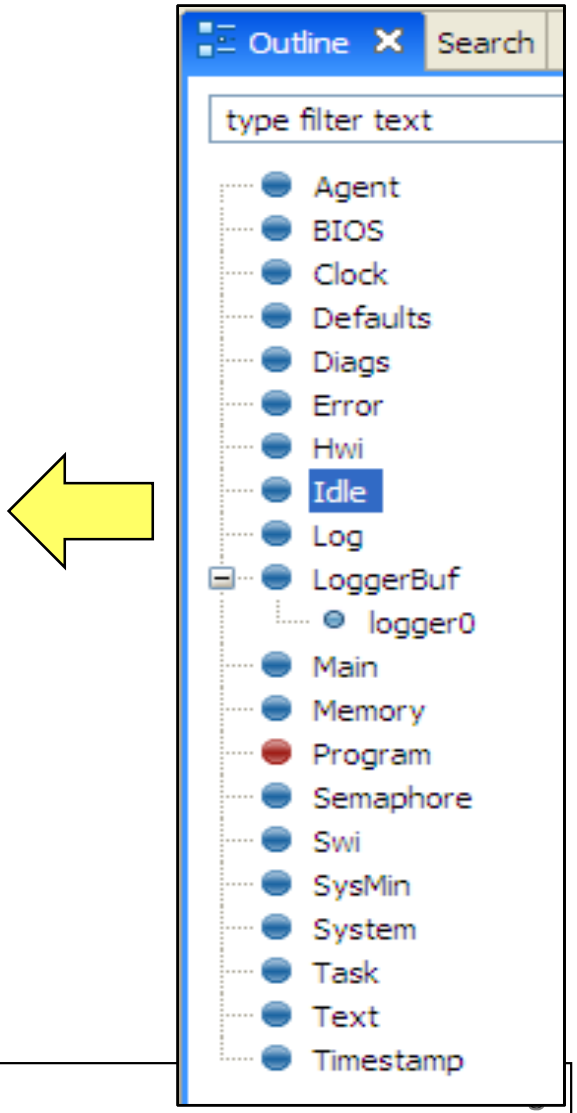
.CFG Files (XDC script)

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

app.cfg

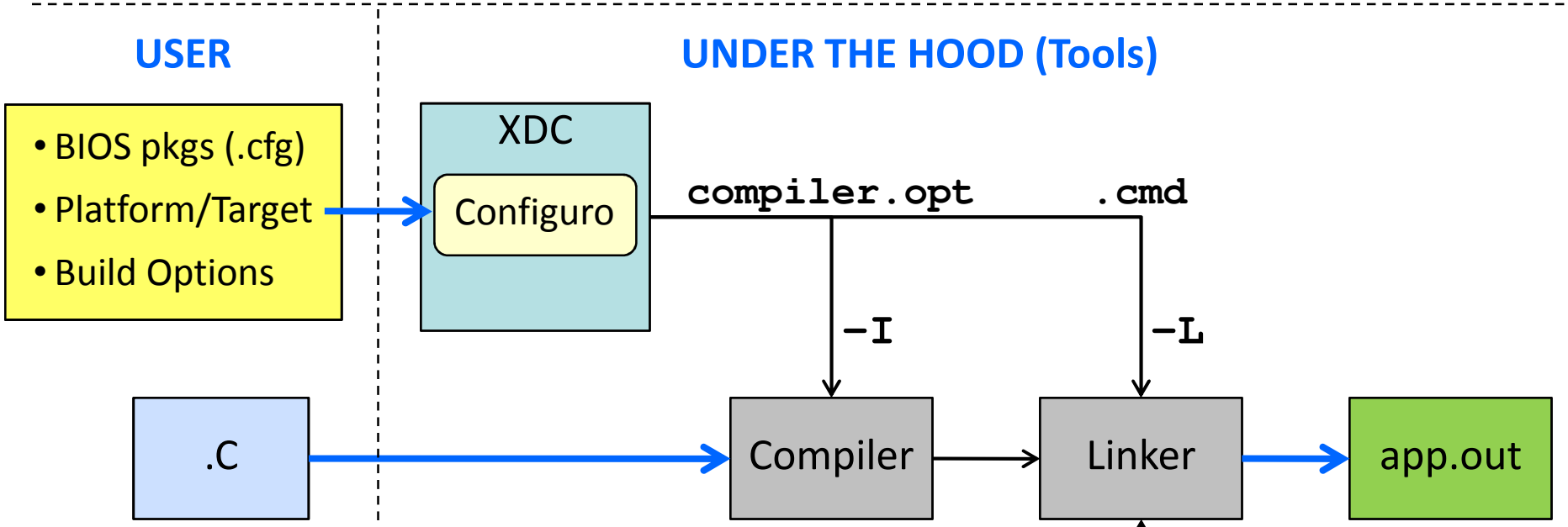
```
11
12var BIOS = xdc.useModule('ti.sysbios.BIOS');
13var Clock = xdc.useModule('ti.sysbios.knl.Clock');
14var Swi = xdc.useModule('ti.sysbios.knl.Swi');
15var Task = xdc.useModule('ti.sysbios.knl.Task');
16var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');
17var Hwi = xdc.useModule('ti.sysbios.hal.Hwi');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

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



Configuration Build Flow (CFG)

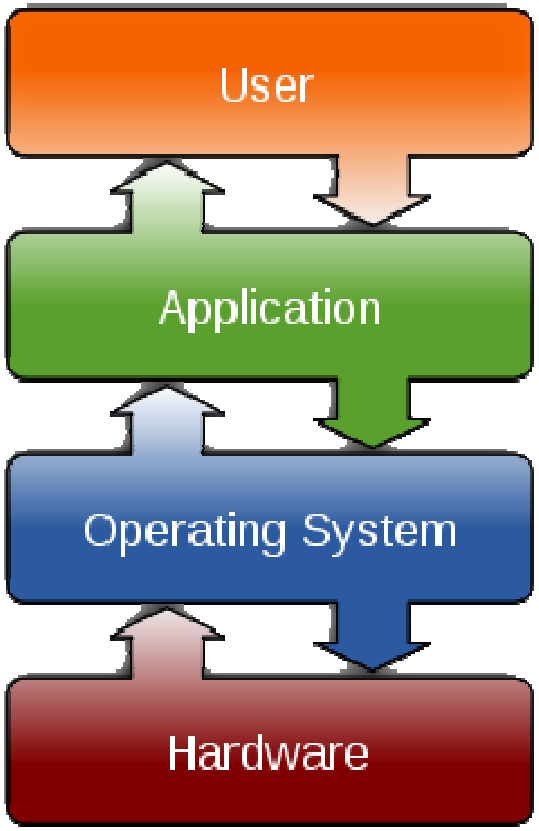
- SYS/BIOS: User configures system with CFG file
- The rest is "under the hood."



- BIOS modules (e.g., Hwi, Clock, Semaphore, etc.) are delivered as RTSC compliant packages.
- RTSC (Real Time Software Components) are packages that contain libraries and metadata (similar to Java.jar files)
- XDC (eXpress DSP Components) is a set of tools that 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 Information
- BIOS Threads



Platform (Memory Config)

Memory Configuration

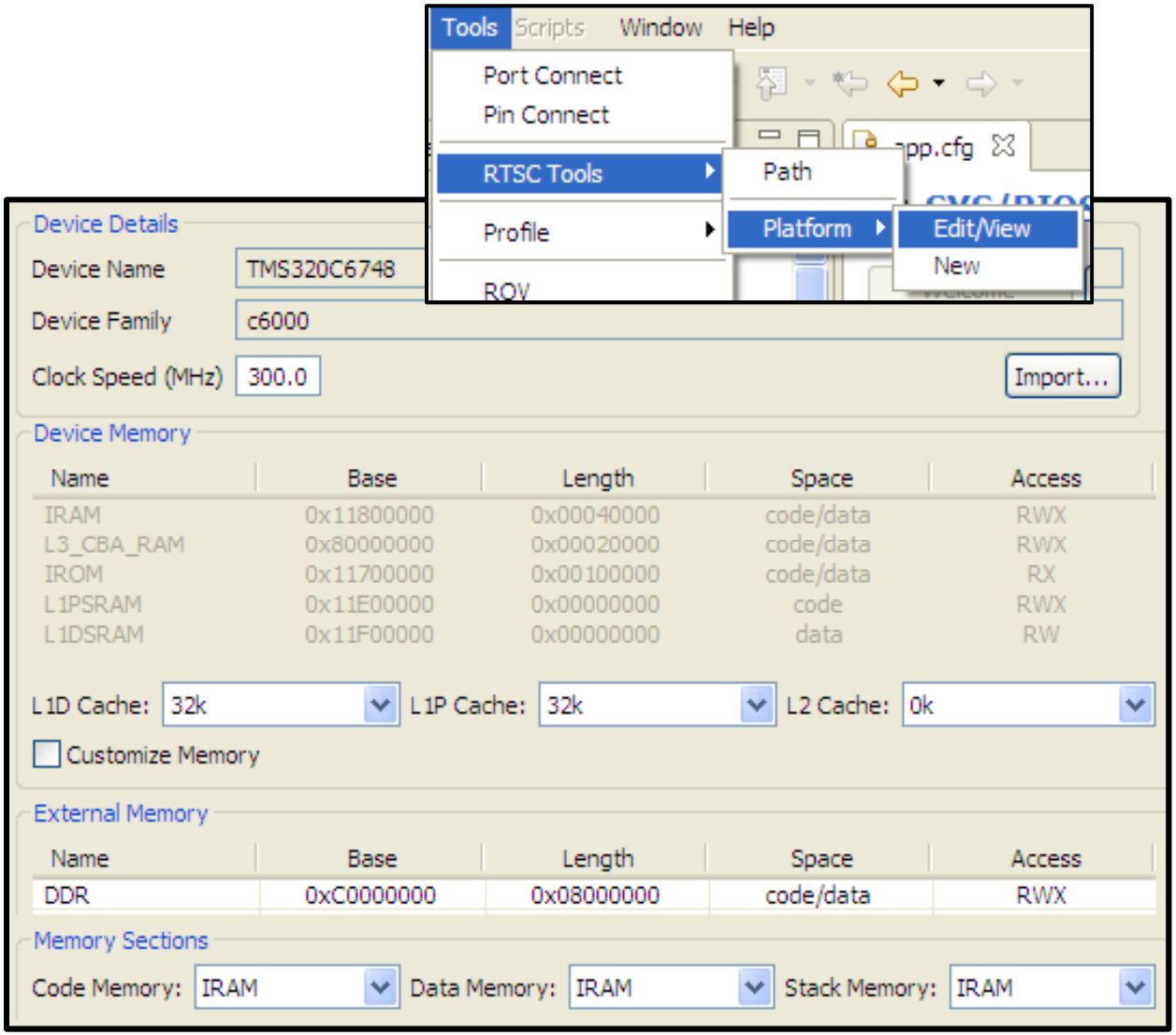
- Create Internal Memory Segments (e.g. IRAM)
- Configure cache
- Define External Memory Segments

Section Placement

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

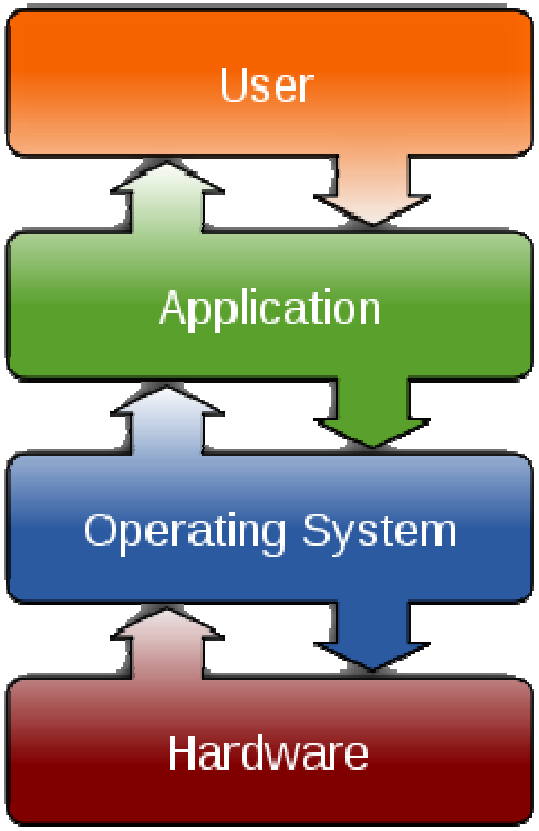
Custom Platform

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



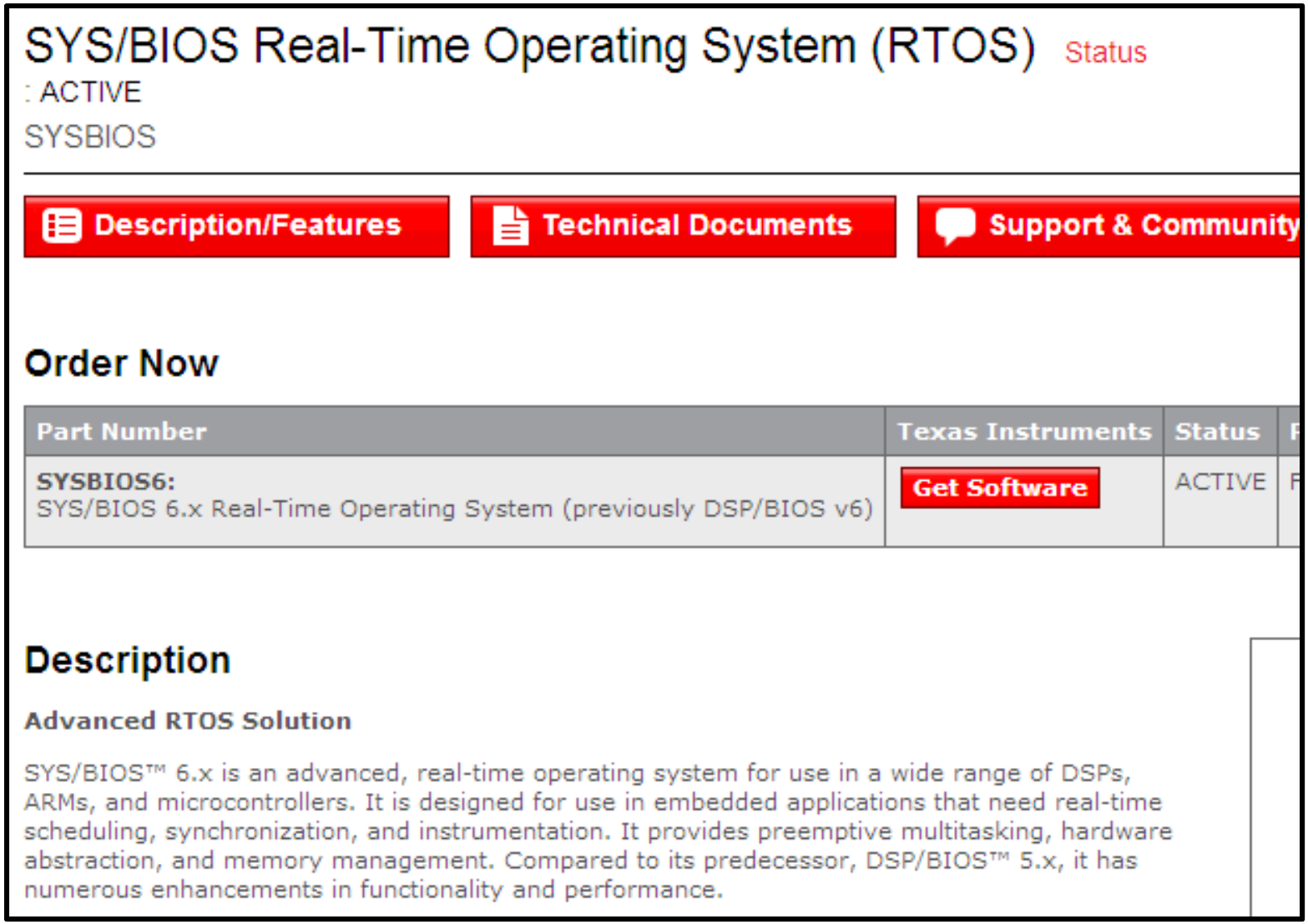
Outline

- 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 Information
- BIOS Threads



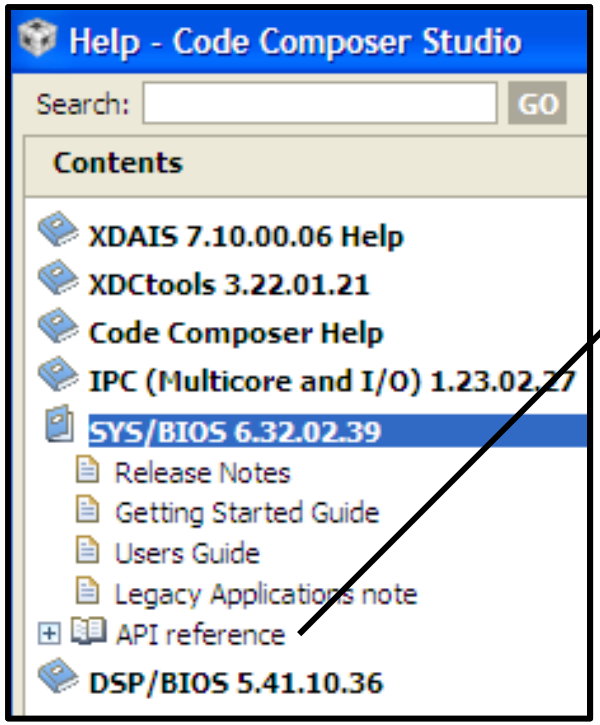
For More Information (1)

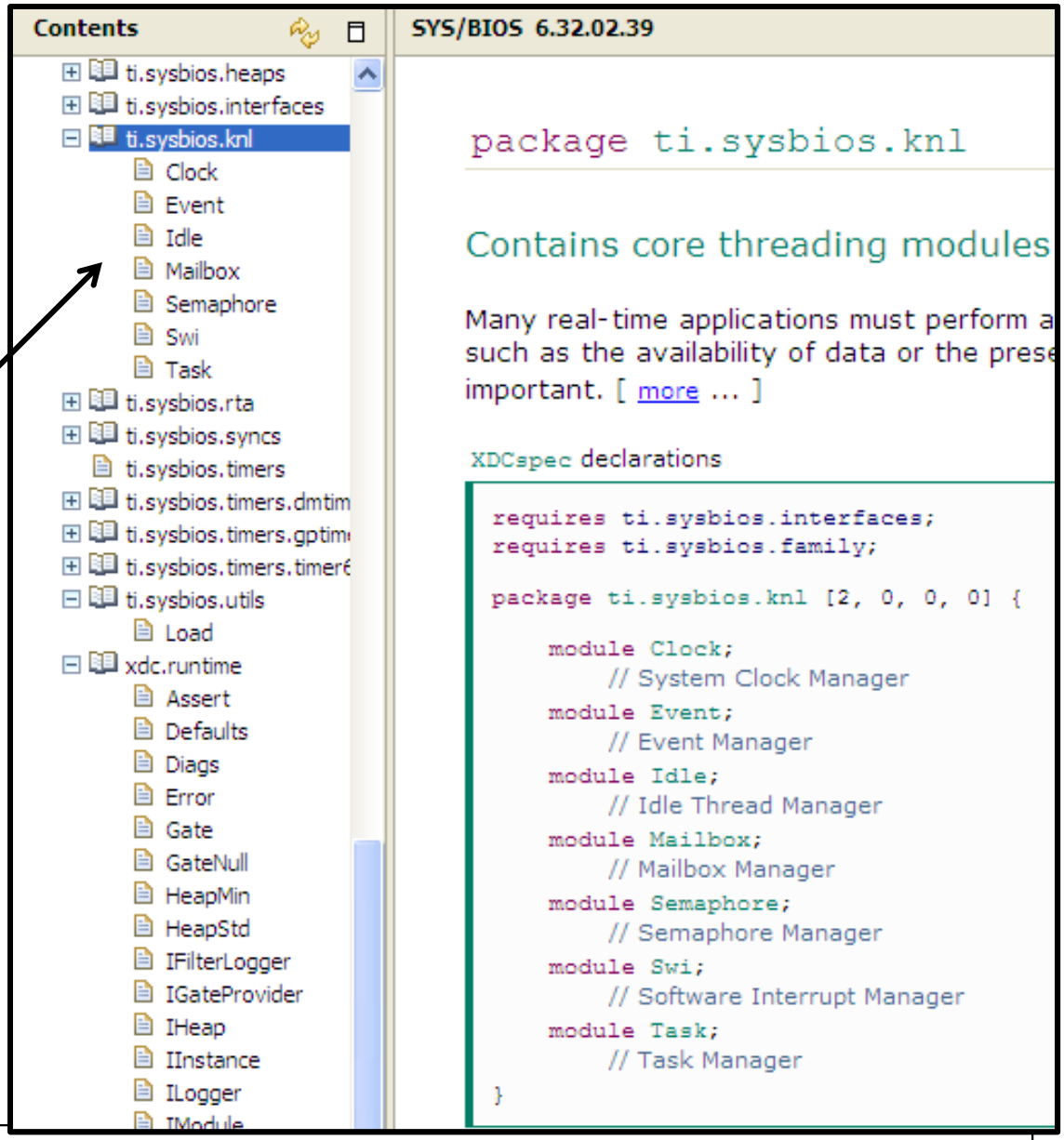
- **SYS/BIOS Product Page http://www.ti.com/sysbios**



For More Information (2)

- **CCS Help Contents**



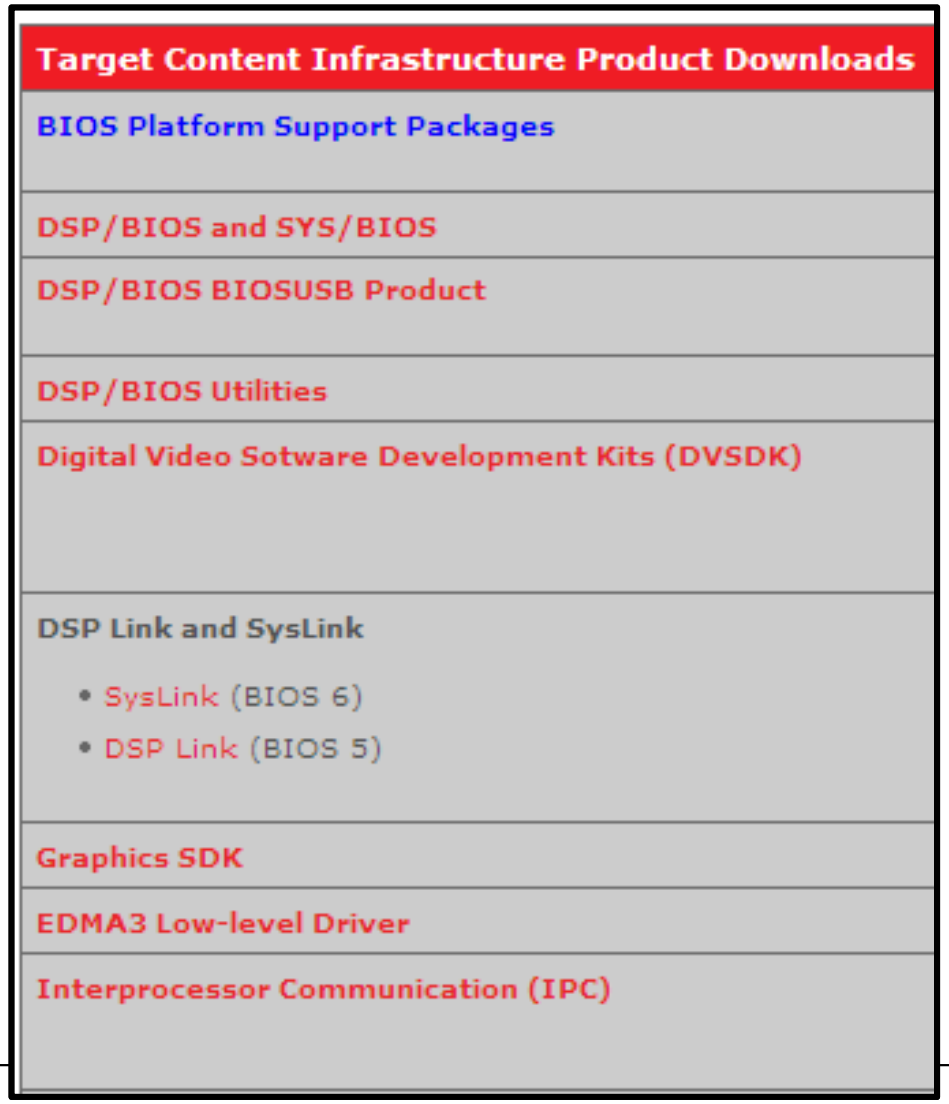


- User Guides
- API Reference (knl)

Download Latest Tools

- **Download Target Content**

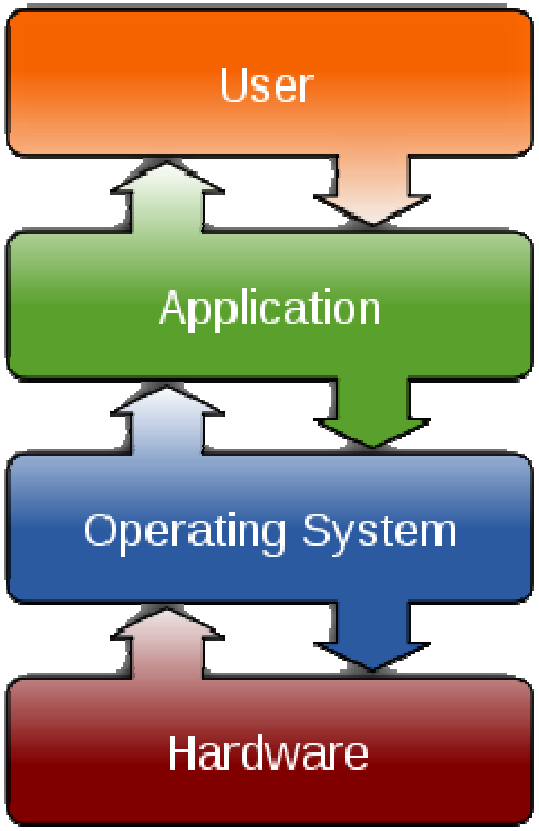
http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/



- 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
Hardware Interrupts

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

Swi
Software Interrupts

- Up to 32 priority levels
- Any number possible, all preemptive

Tsk
Tasks

- Up to 32 priority levels
- Any number possible, all preemptive

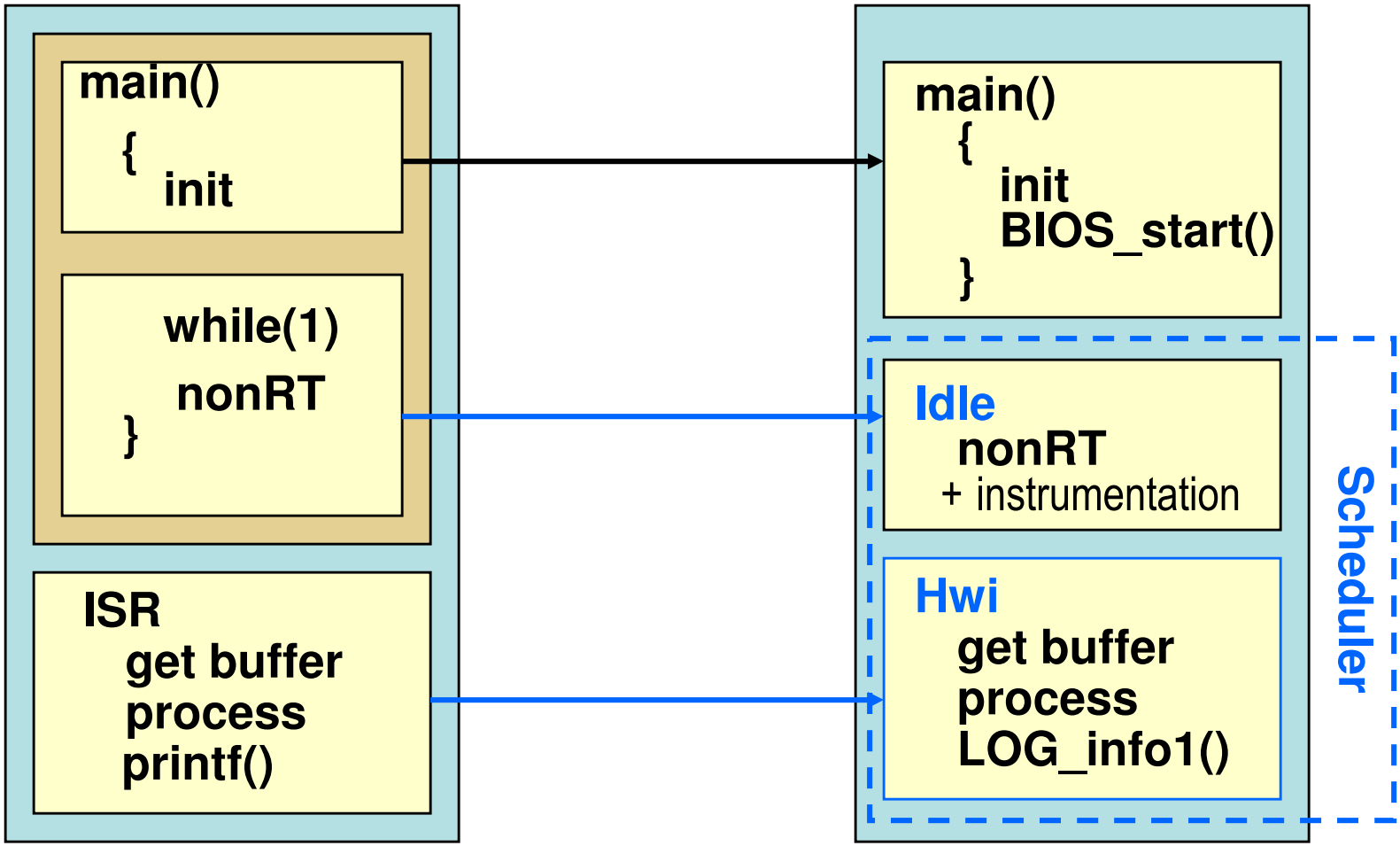
Soft R/T

Idle
Background

- Continuous loop
- Non-realtime in nature

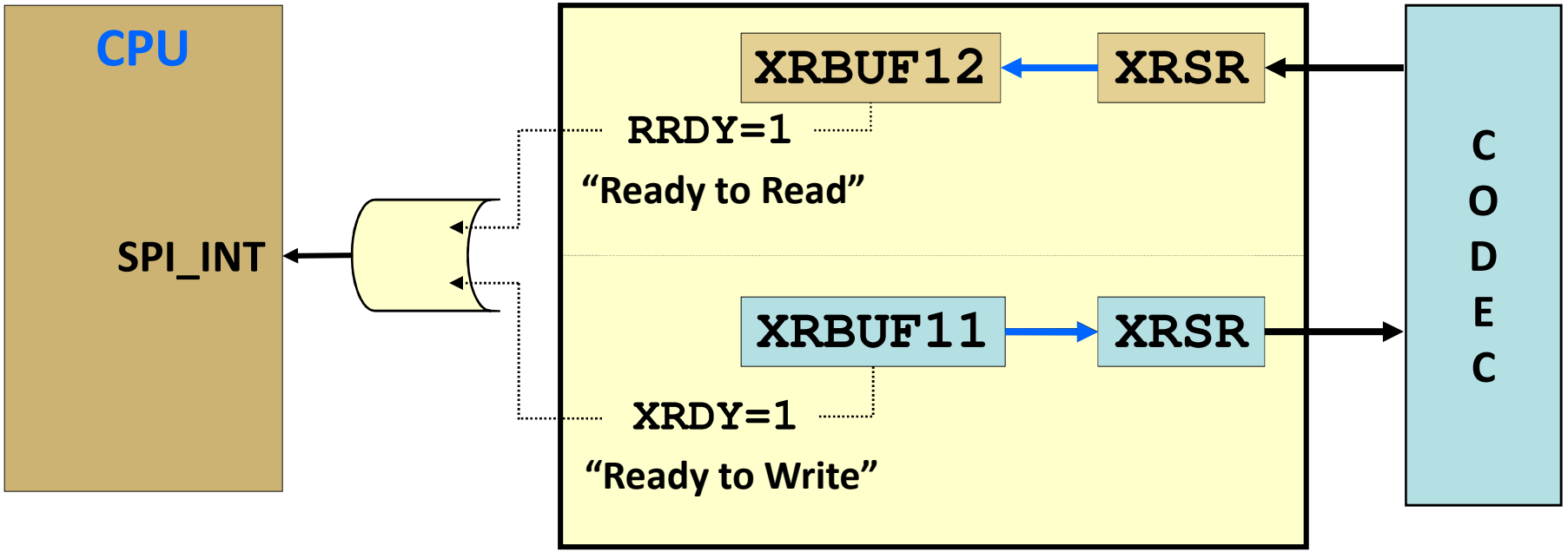
- Idle events run in sequence when no Hwi are posted.
- Hwi is ISR with automatic vector table generation + context save/restore.
- Any Hwi preempts Idle, Hwi may preempt other Hwi if desired.

Foreground / Background Scheduling



- Idle events run in sequence when no Hwi 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 (SPI)



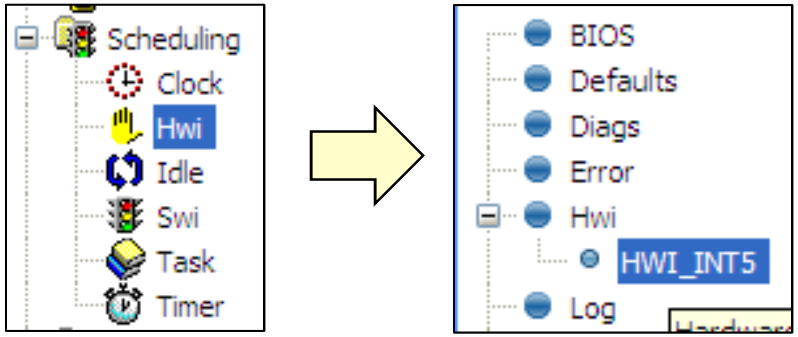
- A peripheral (e.g., SPI on C6678) causes an interrupt to the CPU to indicate "service required."
- This "event" will have an ID (datasheet) and can be tied to a specific CPU interrupt (target specific).

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

Configuring an Hwi: Statically via GUI

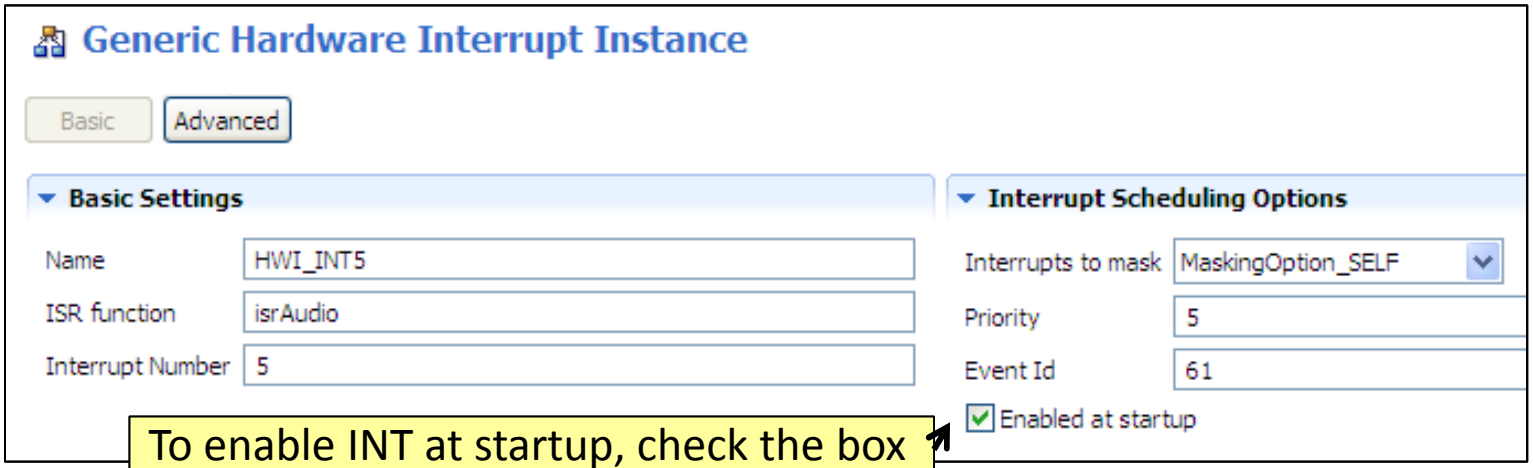
Example: Tie SPI_INT to the CPU HWI_5

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



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

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



To enable INT at startup, check the box

Where do you find the Event Id #?

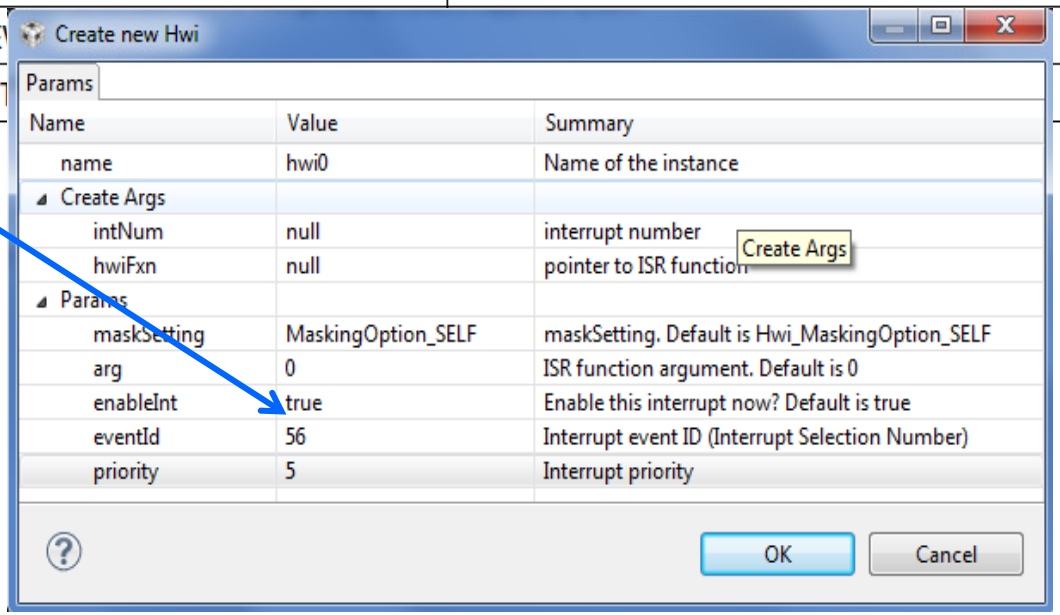
Hardware Event IDs

- How do you know the names of the interrupt events and their corresponding event numbers?

  Look it up in the datasheet.

Source: TMS320C6678 datasheet

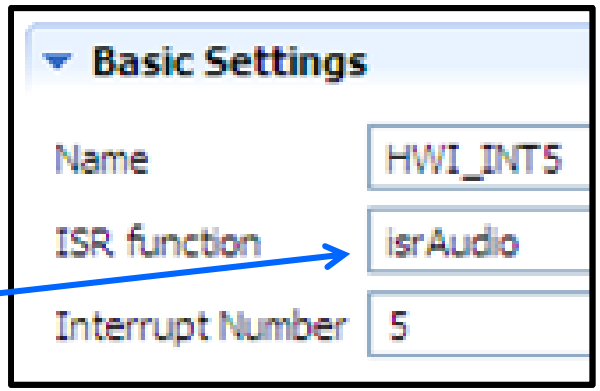
52	PCIEXpress_Legacy_INTC	Legacy interrupt mode
53	PCIEXpress_Legacy_INTD	Legacy interrupt mode
54	SPIINT0	SPI interrupt0
55	SPIINT1	SPI interrupt1
56	SPIXEVT	Transmit event
57	SPIRE	
58	I2CINT	



- As appropriate, refer to the datasheet for your target platform.

Example ISR (SPI)

Example ISR for SPIXEVT_INT interrupt

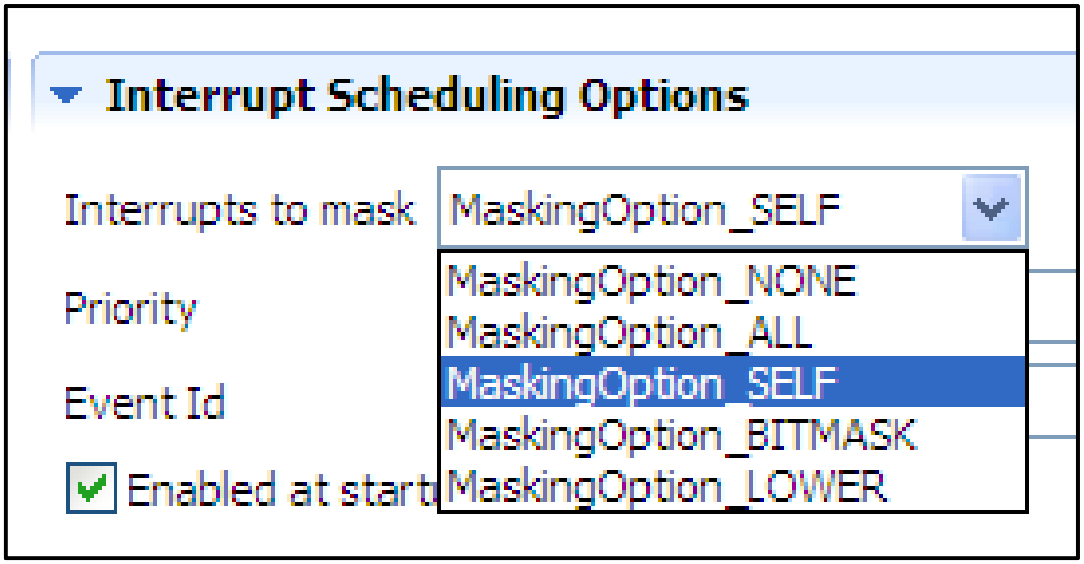


isrAudio:

```
pInBuf[blkCnt] = SPI->RCV;        // READ audio sample from SPI
SPI->XMT = pOutBuf[blkCnt]        // WRITE audio sample to SPI
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 activities 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)

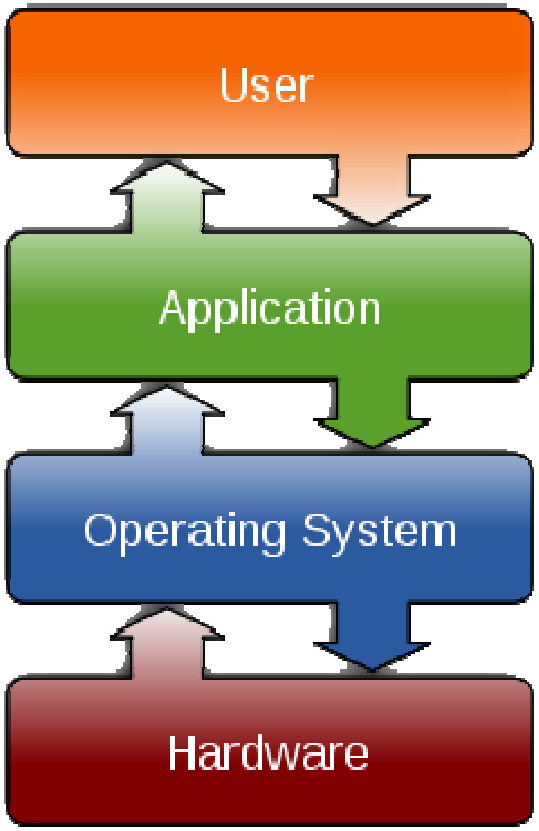
SYS/BIOS Hwi APIs

Other useful Hwi APIs:

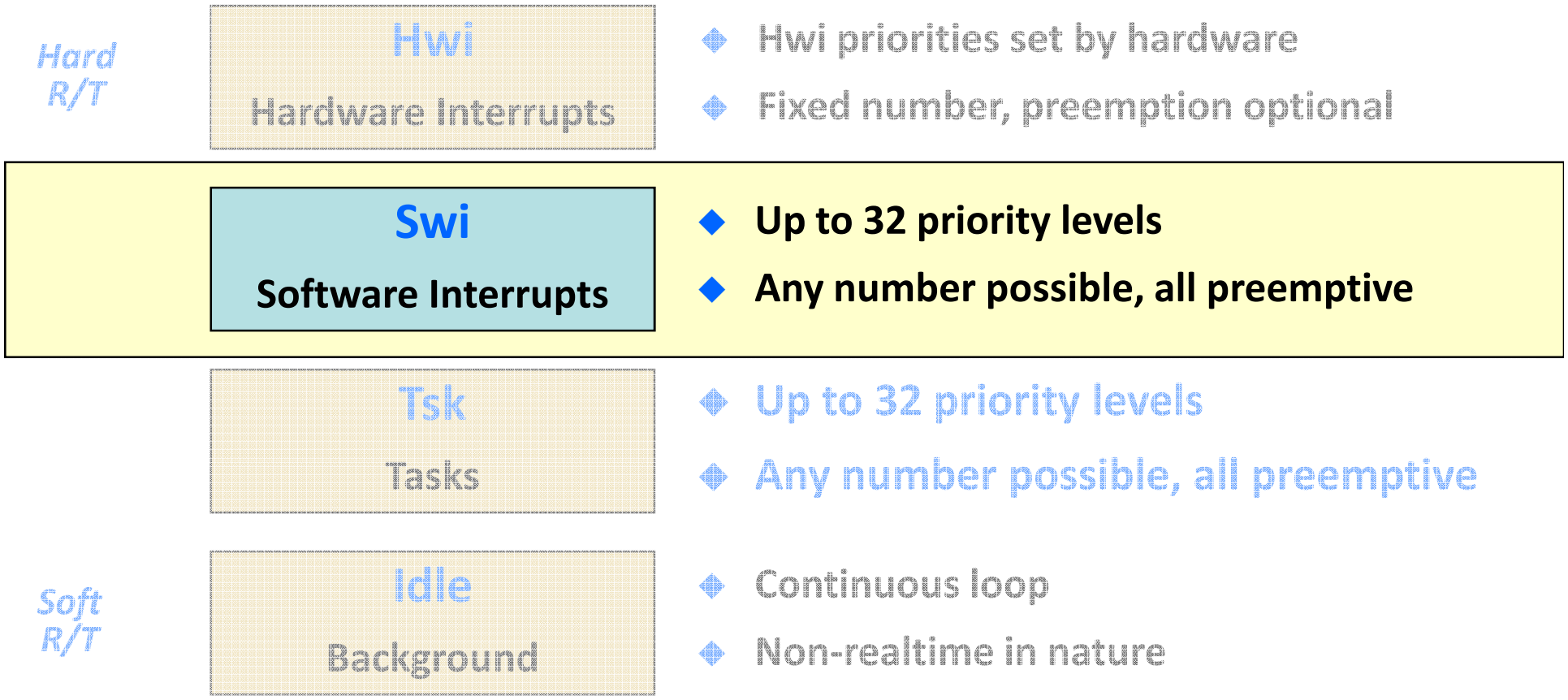
`Hwi_disableInterrupt()`	Set enable bit = 0
`Hwi_enableInterrupt()`	Set enable bit = 1
`Hwi_clearInterrupt()`	Clear INT flag bit = 0
`Hwi_post()` **New in SYS/BIOS**	Post INT # (in code)
`Hwi_disable()`	Global INTs disable
`Hwi_enable()`	Global INTs enable
`Hwi_restore()`	Global INTs restore

Outline

- Intro to SYS/BIOS
- BIOS Threads
  - Hardware Interrupts (Hwi)
  - **Software Interrupts (Swi)**
  - Tasks (Tsk)
  - Semaphores (Sem)



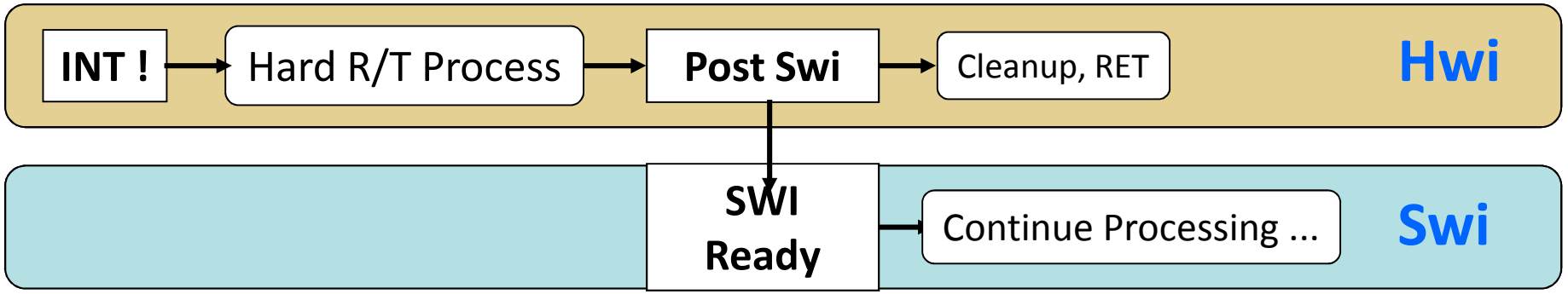
Swi Scheduling



- 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:

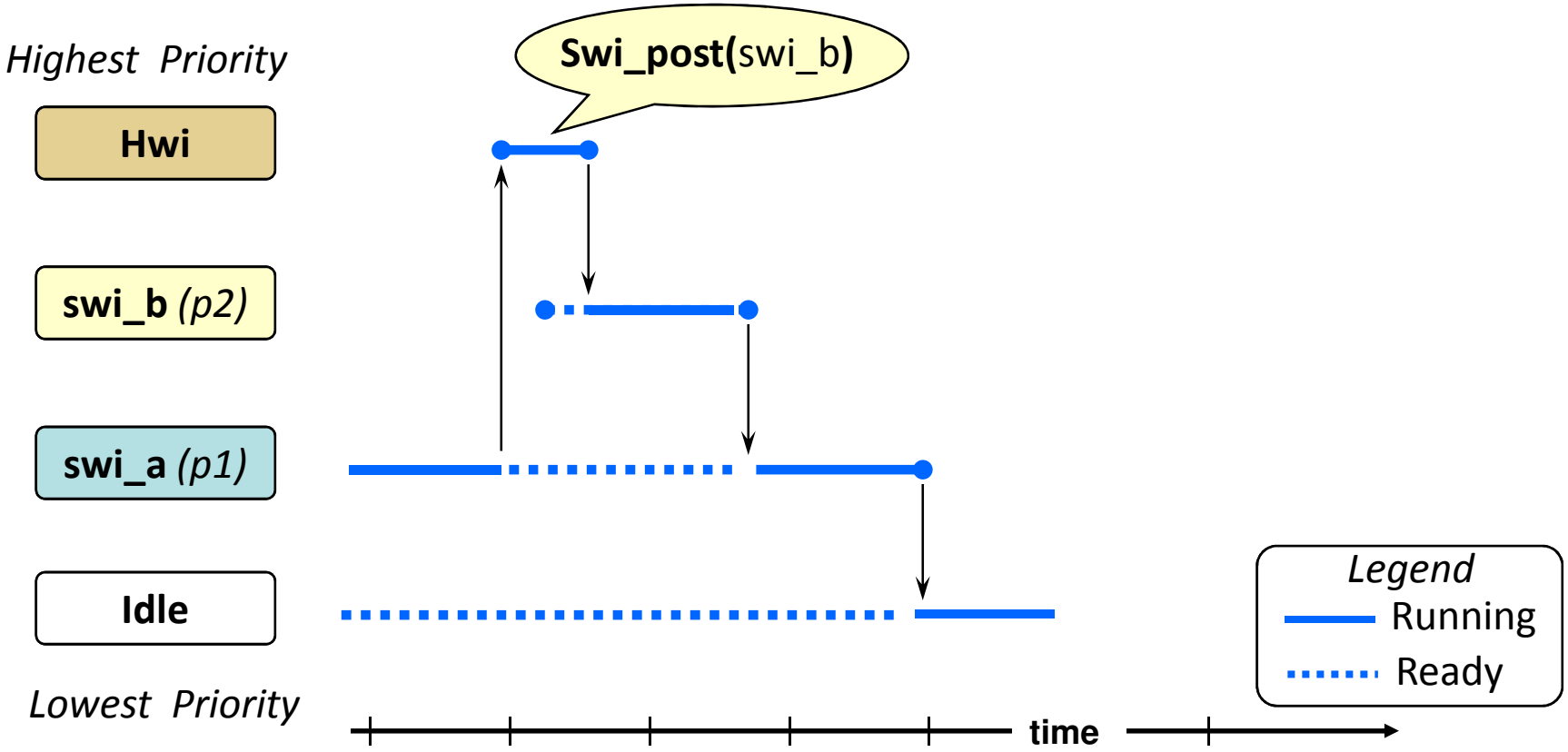
```
*buf++ = *XBUF;
cnt++;
if (cnt >= BLKSZ) {
   Swi_post(swiFir);
   count = 0;
   pingPong ^= 1;
}
```

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.

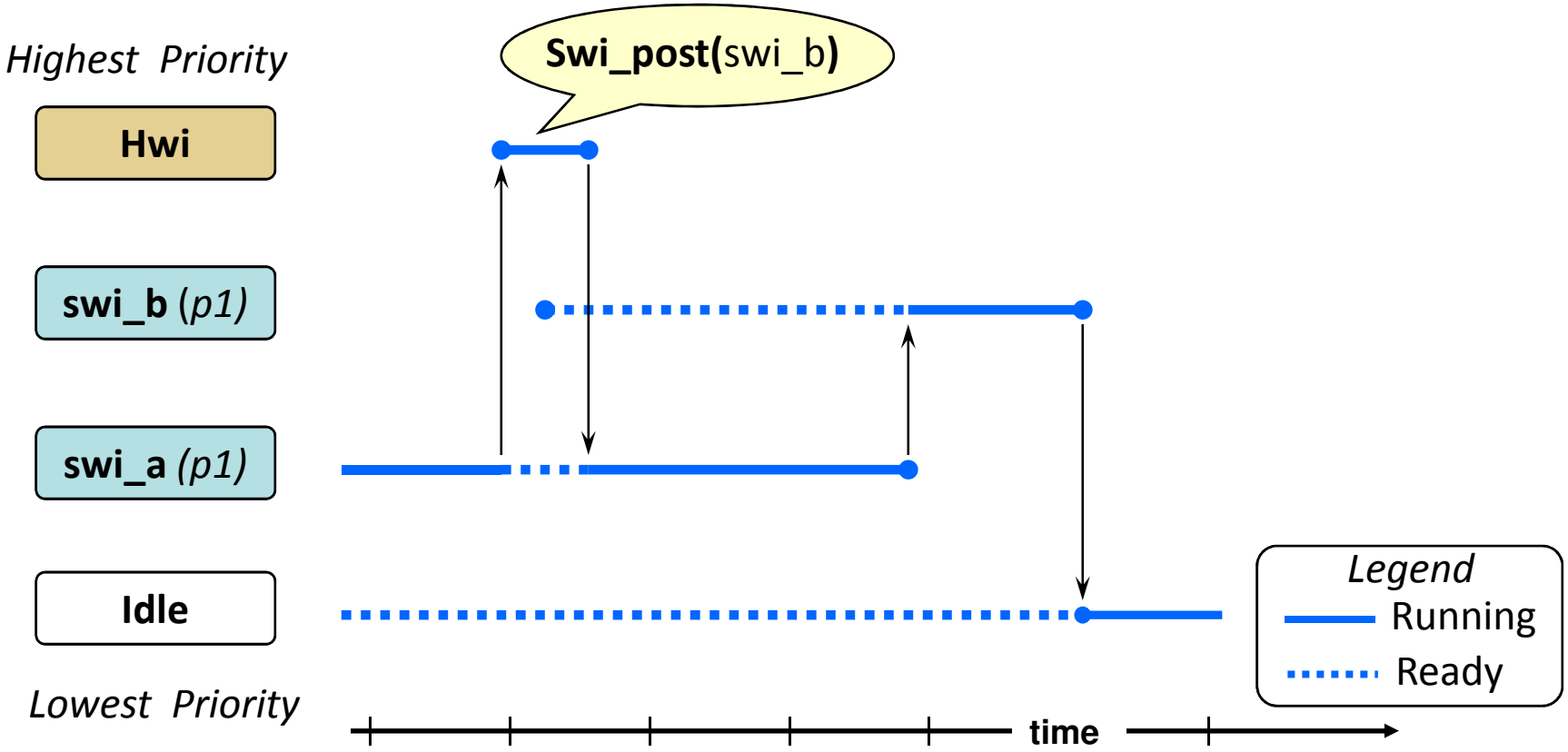
Scheduling Rules



- ***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 range from 1 to 32.
- Automatic context switch (uses system stack)

What if more than one Swi process is set to the same priority?

Scheduling Rules

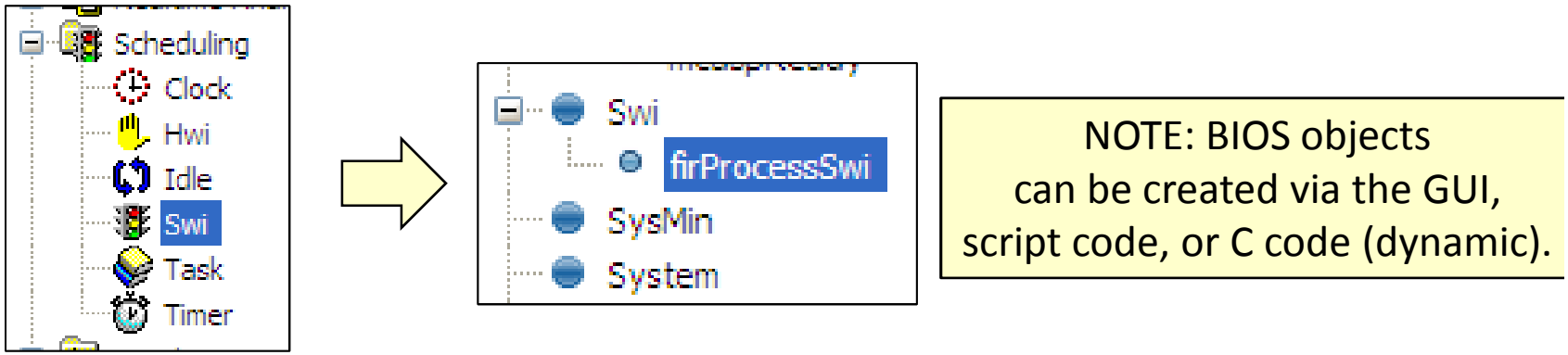


- Processes of same priority are scheduled first-in first-out (FIFO).
- Having threads at the SAME priority offers certain advantages, such as resource sharing (without conflicts).

Configuring a Swi: Statically via GUI

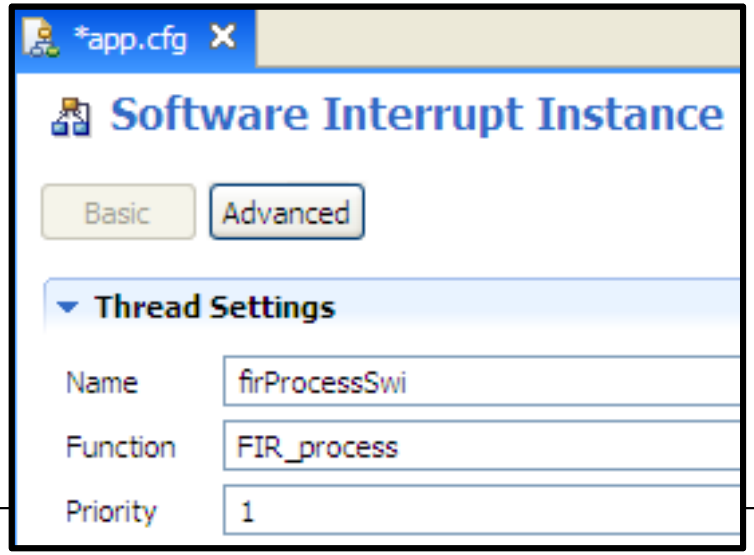
Example: **Tie isrAudio() fxn to Swi; Use priority 1**

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



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

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



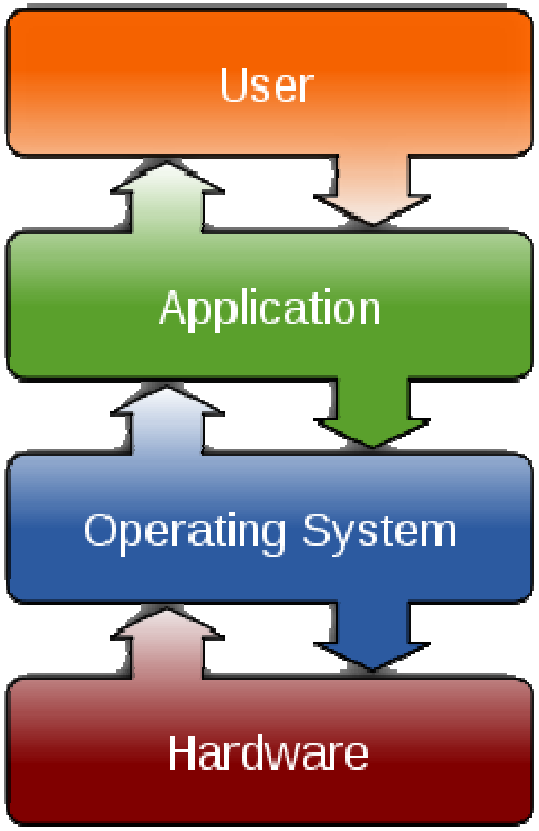
SYS/BIOS Swi APIs

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

Outline

- Intro to SYS/BIOS
- BIOS Threads
  - Hardware Interrupts (Hwi)
  - Software Interrupts (Swi)
  - Tasks (Tsk)
  - Semaphores (Sem)



Task Scheduling

Hard R/T

Hwi
Hardware Interrupts

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

Swi
Software Interrupts

- Up to 32 priority levels
- Any number possible, all preemptive

Tsk
Tasks

- **Up to 32 priority levels**
- **Any number possible, all preemptive**

Soft R/T

Idle
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 Tsk, 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.

How is a Task is different from a Swi?.

Comparing Swi and 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** share system stack w/ Hwi
- Usage: As follow-up to Hwi and/or when memory size is an absolute premium

Task

_create →

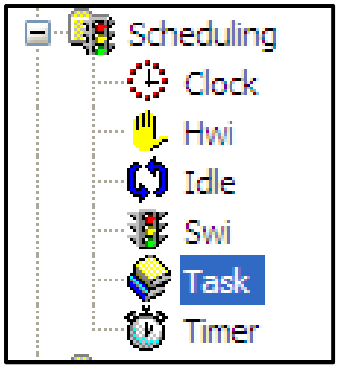
```
void myTask () {
   // Prologue (set Task env)
   while(cond){
      Semaphore_pend();
      *** RUN ***
   }
   // Epilogue (free env)
}
```

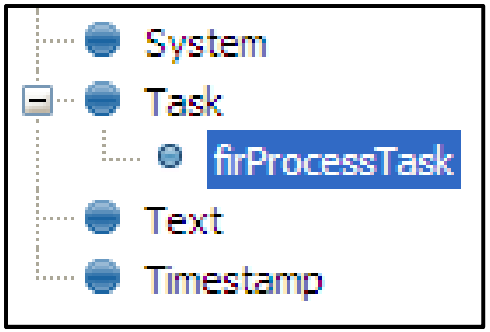
- "Ready" when *CREATED* (BIOS_start or dynamic)
- P-L-E structure handy for resource creation (P) and deletion (E), initial state preserved
- Can block/suspend on semaphore (flag)
- Context switch speed (~160c)
- Uses its OWN stack to store context
- Usage: Full-featured sys, CPU w/more speed/mem

Configuring a Task: Statically via the GUI

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

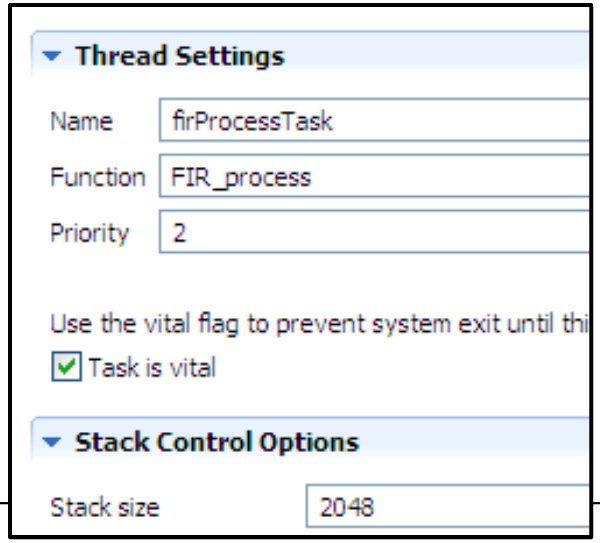
1 **Use Task module** *(Available Products)* **, insert new Task** *(Outline View)*





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

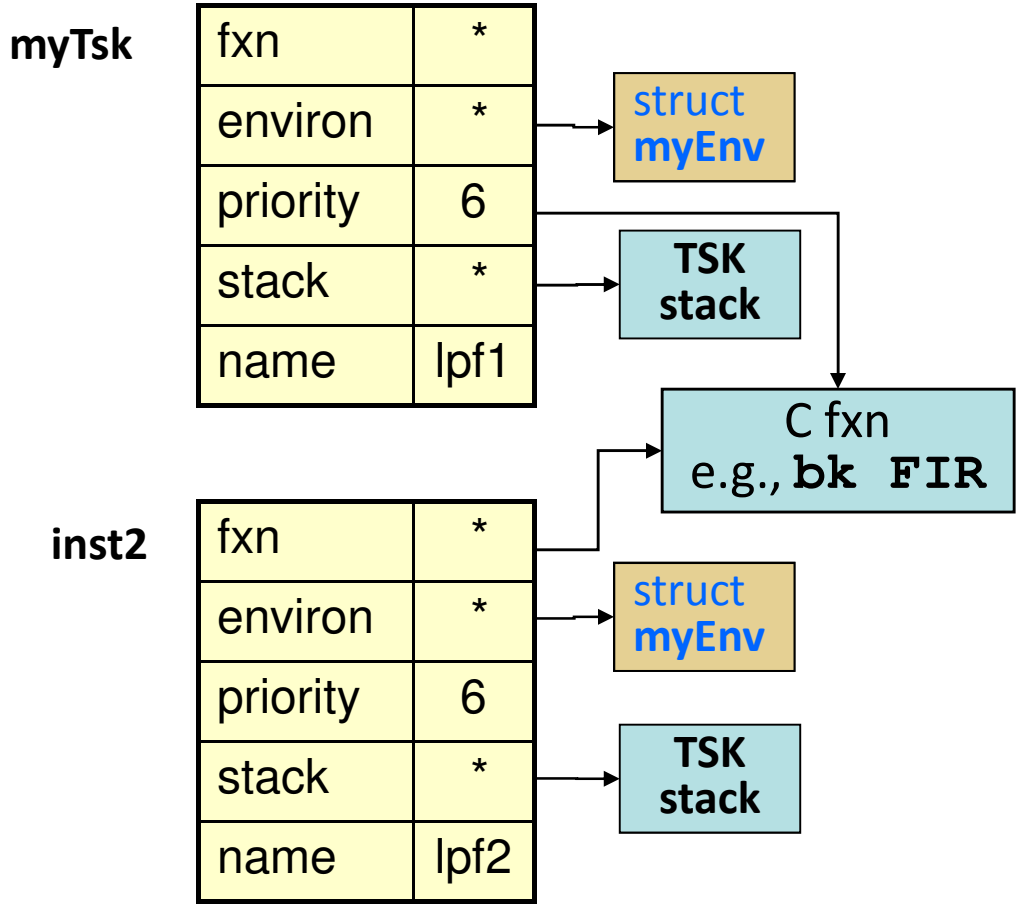
2 **Configure Task: Object name, function, priority, stack size**



Task Object Concepts

Task object:

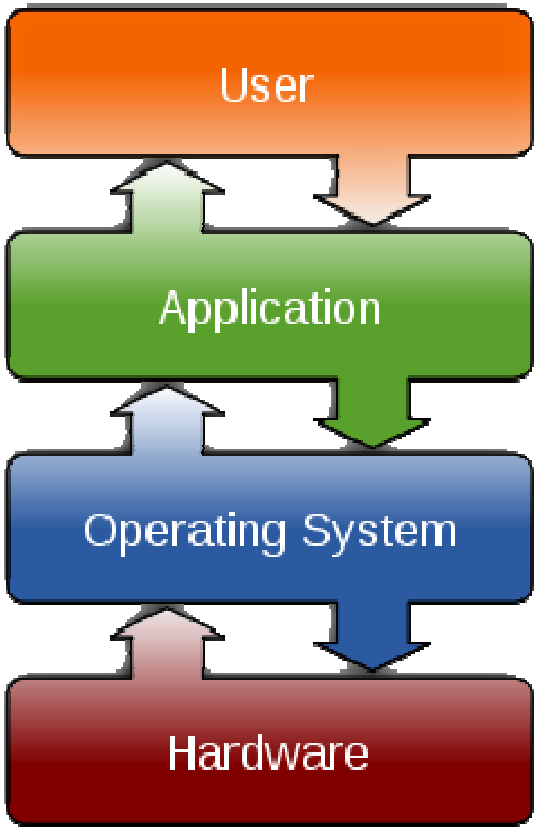
- Pointer to task function
- Priority: changable
- Pointer to task's stack
  - Stores local variables
  - Nested function calls
  - makes blocking possible
  - Interrupts run on the system stack
- Pointer to text name of TSK
- Environment: pointer to *user defined* structure:



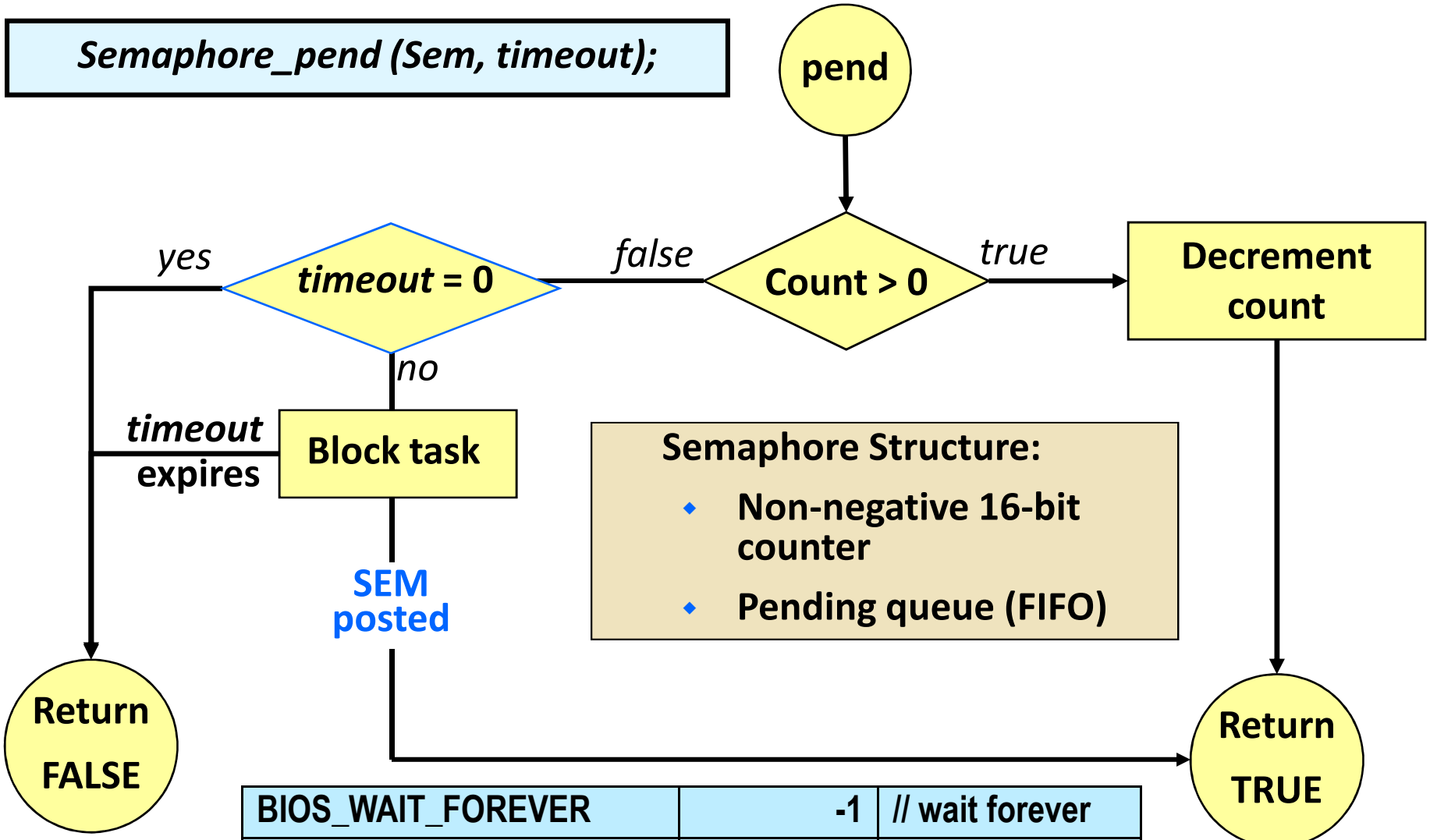

```
Task_setenv(Task_self(),&myEnv);

hMyEnv = Task_getenv(&myTsk);
```

Outline

- Intro to SYS/BIOS
- BIOS Threads
  - Hardware Interrupts (Hwi)
  - Software Interrupts (Swi)
  - Tasks (Tsk)
  - Semaphores (Sem)

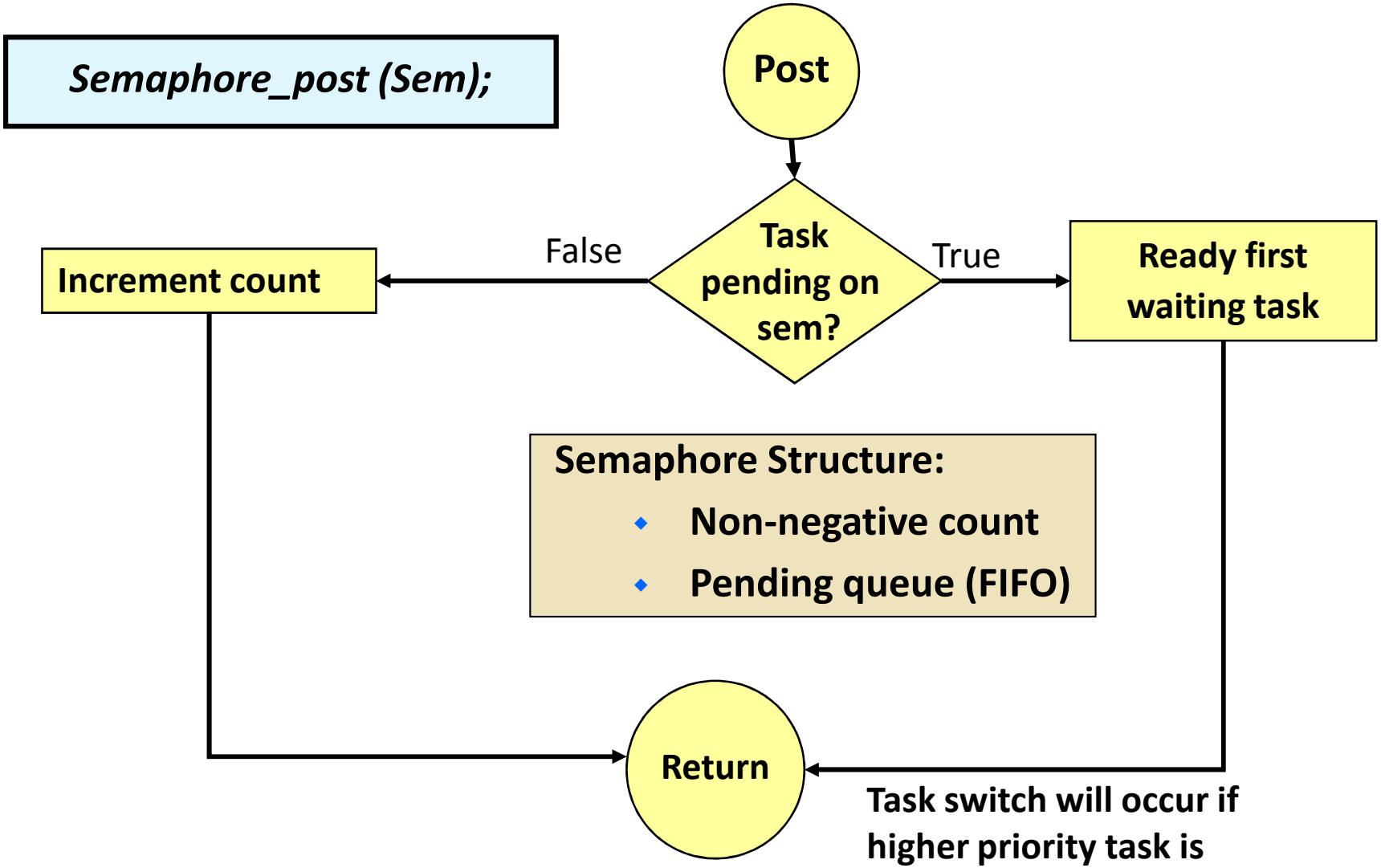


Semaphore Pend



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

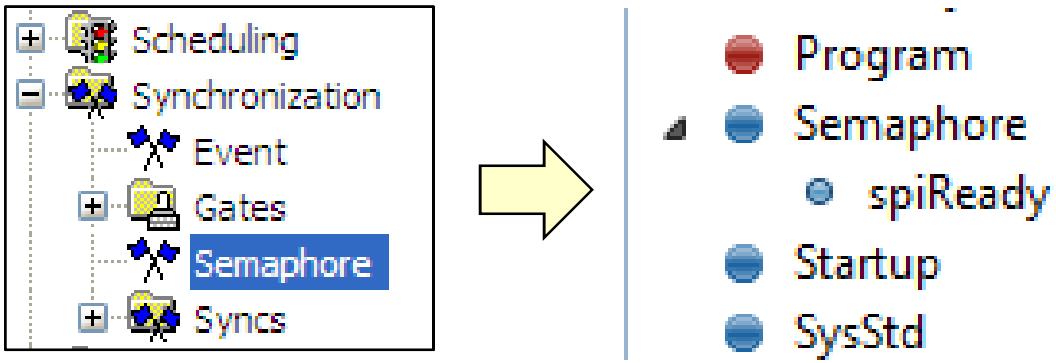
Semaphore Post



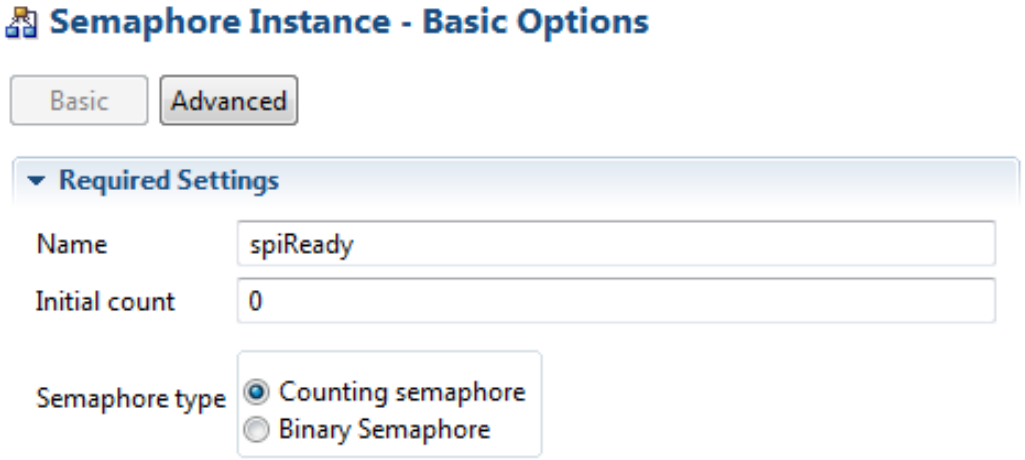
Configuring a Semaphore: Statically via GUI

Example: Create *spiReady*, 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?