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?
  - 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

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**

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

**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**
- Hardware Interrupts
  - Implements ‘urgent’ part of real-time event
  - Hardware interrupt 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**

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

**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

- System Stack (Hwi/Swi)
- Swi_post (Swi);
- “run to completion”
- end

Task

- Semaphore_post (Sem);
- Semaphore_pend
- Pause (blocked state)
- start
- end

- 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)**

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

**Static (GUI or Script)**

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

---

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

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
<th>Provided by TI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Reset</td>
<td>BIOS_init()</td>
<td>BIOS_start()</td>
</tr>
<tr>
<td>Boot Loader</td>
<td>BIOS_init()</td>
<td>BIOS_start()</td>
</tr>
<tr>
<td></td>
<td>BIOS_init()</td>
<td>BIOS_start()</td>
</tr>
<tr>
<td></td>
<td>_c_int00</td>
<td>_c_int00</td>
</tr>
</tbody>
</table>

- **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()

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

Execution Graph

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

Multicore Training
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”
.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

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

Configuration Build Flow (CFG)

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

**USER**

- BIOS pkgs (.cfg)
- Platform/Target
- Build Options

**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
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 Real-Time Operating System (RTOS)

Order Now

<table>
<thead>
<tr>
<th>Part Number</th>
<th>Texas Instruments</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSBIOS: SYS/BIOS 6.x Real-Time Operating System (previously DSP/BIOS v6)</td>
<td>Get Software</td>
<td>ACTIVE</td>
</tr>
</tbody>
</table>

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

Target Content Infrastructure Product Downloads

- DSP/BIOS and SYS/BIOS
- DSP/BIOS BIOS USB Product
- DSP/BIOS Utilities
- Digital Video Software 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

<table>
<thead>
<tr>
<th>Hard R/T</th>
<th>Hwi (hi)</th>
<th>Hwi priorities set by hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fixed number, preemption optional</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Soft R/T</th>
<th>Swi</th>
<th>Software Interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Task</td>
<td>Tasks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Idle (lo)</th>
<th>Background</th>
</tr>
</thead>
</table>

- 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: SPI)**

- 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’s HWI5

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

![Image]

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

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

![Image]

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: TMS320C6678 datasheet (exerpt):**

<table>
<thead>
<tr>
<th>Event ID</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>PCEO Spurs_Legacy_INT</td>
<td>Legacy interrupt mode</td>
</tr>
<tr>
<td>53</td>
<td>PCEO Spurs_Legacy_INT</td>
<td>Legacy interrupt mode</td>
</tr>
<tr>
<td>54</td>
<td>SPIINT0</td>
<td>SPI interrupt</td>
</tr>
<tr>
<td>55</td>
<td>SPIINT1</td>
<td>SPI interrupt</td>
</tr>
<tr>
<td>56</td>
<td>SPIXINT</td>
<td>Transmit event</td>
</tr>
<tr>
<td>57</td>
<td>SPIRESET</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>HINT</td>
<td></td>
</tr>
</tbody>
</table>

- This example is target-specific for the C6678 DSP. Simply refer to your target’s datasheet for similar info.

What happens in the ISR?
Example ISR (SPI)

Example ISR for SPI_EVT_INT interrupt:

```c
ISR function: isrAudio
```

```c
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
    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:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Hwi_disableInterrupt()</code></td>
<td>Set enable bit = 0</td>
</tr>
<tr>
<td><code>Hwi_enableInterrupt()</code></td>
<td>Set enable bit = 1</td>
</tr>
<tr>
<td><code>Hwi_clearInterrupt()</code></td>
<td>Clear INT flag bit = 0</td>
</tr>
<tr>
<td><code>Hwi_post()</code></td>
<td>Post INT # (in code)</td>
</tr>
<tr>
<td><code>Hwi_disable()</code></td>
<td>Global INTs disable</td>
</tr>
<tr>
<td><code>Hwi_enable()</code></td>
<td>Global INTs enable</td>
</tr>
<tr>
<td><code>Hwi_restore()</code></td>
<td>Global INTs restore</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hwi (Hi)</strong></td>
<td>Hwi (Hi) Hardware Interrupts&lt;br&gt;- Hwi priorities set by hardware&lt;br&gt;- Fixed number, preemption optional</td>
</tr>
<tr>
<td><strong>Swi</strong></td>
<td>Swi Software Interrupts&lt;br&gt;- Up to 32 priority levels (16 on C28x)&lt;br&gt;- Any number possible, all preemptive</td>
</tr>
<tr>
<td><strong>Task</strong></td>
<td>Task Tasks&lt;br&gt;- Up to 32 priority levels (16 on C28x)&lt;br&gt;- Any number possible, all preemptive</td>
</tr>
<tr>
<td><strong>Idle (Io)</strong></td>
<td>Idle (Io) Background&lt;br&gt;- Continuous loop&lt;br&gt;- Non-realtime in nature</td>
</tr>
</tbody>
</table>

- 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:**

1. INT → Hard R/T Process
2. Post Swi → Cleanup, RET
3. **Hwi**
4. ISR Audio
   ```
   *buf++ = *XBUF;
   cnt++;
   if (cnt >= BLKSZ) {
     Swi_post(swiFir);
     count = 0;
     pingPong ^= 1;
   }
   **Swi**

- **Hwi**<br>  - Fast response to INTs<br>  - Min context switching<br>  - High priority for CPU<br>  - Limited # of Hwi possible
- **Swi**<br>  - Latency in response time<br>  - Context switch<br>  - Selectable priority levels<br>  - 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

Highest Priority

Swi_post(swi_b)

swi_b (p2)

swi_a (p1)

Idle

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)

Legend

Running

Ready

What if the SWIs are at the same priority?

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)

Swi priorties from 1 to 32 (C28x has 16)

Automatic context switch (uses system stack)

Legend

Running

Ready

How do you configure a SWI?
Configuring a **Swi** – Statically via GUI

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

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

   ![Swi module GUI](image)

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

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

   ![Swi configuration GUI](image)

   Let's move on to Tasks...

---

**SYS/BIOS Swi APIs**

Other useful Swi APIs:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swi_inc</td>
<td>Post, increment count</td>
</tr>
<tr>
<td>Swi_dec</td>
<td>Decrement count, post if 0</td>
</tr>
<tr>
<td>Swi_or</td>
<td>Post, OR bit (signature)</td>
</tr>
<tr>
<td>Swi_andn</td>
<td>ANDn bit, post if all posted</td>
</tr>
<tr>
<td>Swi_getPri</td>
<td>Get any Swi Priority</td>
</tr>
<tr>
<td>Swi_enable</td>
<td>Global Swi enable</td>
</tr>
<tr>
<td>Swi_disable</td>
<td>Global Swi disable</td>
</tr>
<tr>
<td>Swi_restore</td>
<td>Global Swi restore</td>
</tr>
</tbody>
</table>

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

**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

```c
void taskFunction(...) {
    /* Prolog */
    while ('condition'){
        Semaphore_pend()
        /* Process */
    }
    /* Epilog */
}
```

- 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**

```c
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**

```c
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)

   ![Insert new Task](image)

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

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

   ![Configure Task](image)

Outline

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

Semaphore_pend (Sem, timeout);

- **Decrement count**
  - **timeout = 0**
  - **Count > 0**

- **Block task**
  - **timeout expires**
  - **SEM posted**

- **Return FALSE**

- **Return TRUE**

**Semaphore Structure:**
- Non-negative 16-bit counter
- Pending queue (FIFO)

**How does _post work?**

Semaphore Post

Semaphore_post (Sem);

- **Post**
  - **Increment count**
  - **Task pending on sem?**
    - **False**
    - **True**
      - **Ready first waiting task**

- **Return**
  - **Task switch will occur if higher priority task is made ready**

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

**How do you configure a Semaphore?**
Configuring a **Semaphore** – Statically via GUI

**Example:** Create spiReady, counting

1. **Use Semaphore** *(Available Products)*, insert new Semaphore *(Outline View)*
   - Program
   - Semaphore
     - spiReady
   - Startup
   - SysStd

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

   ![Semaphore Instance - Basic Options](image)

   - Required Settings
     - Name: spiReady
     - Initial count: 0
     - Semaphore type: Counting semaphore

**SYS/BIOS Semaphore/Task APIs**

Other useful Semaphore APIs:

- `Semaphore_getCount()` – Get semaphore count

Other useful Task APIs:

<table>
<thead>
<tr>
<th>Task Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Task_sleep()</code></td>
<td>Sleep for N system ticks</td>
</tr>
<tr>
<td><code>Task_yield()</code></td>
<td>Yield to same pri Task</td>
</tr>
<tr>
<td><code>Task_setPri()</code></td>
<td>Set Task priority</td>
</tr>
<tr>
<td><code>Task_getPri()</code></td>
<td>Get Task priority</td>
</tr>
<tr>
<td><code>Task_get/setEnv()</code></td>
<td>Get/set Task Env</td>
</tr>
<tr>
<td><code>Task_enable()</code></td>
<td>Enable Task Mgr</td>
</tr>
<tr>
<td><code>Task_disable()</code></td>
<td>Disable Task Mgr</td>
</tr>
<tr>
<td><code>Task_restore()</code></td>
<td>Restore Task Mgr</td>
</tr>
</tbody>
</table>
Questions?