Chapter 6 – Real-Time Operating Systems

Definition:

An operating system (OS) is a program that is loaded into the processor, along with application programs, at boot time. It then manages all the applications, determining which applications should run, in what order, and for what allotted time. Also, it manages the sharing of resources, namely CPU, memory, and peripherals.

Applications can access the Operating System by calling Application Programming Interface (API) functions.

Compare OS to No-OS

OS	No-OS (e.g. Super Loop)
helps manage complex system (when number of concurrent tasks is greater than "a few")	less modular
provides scheduling	does not
provides multi-tasking	does not
processes can run at non-integer multiple rates	cannot easily have arbitrary rates
scalable	less so
takes more CPU cycles (due to overhead)	can achieve fastest speed (due to no overhead)
uses more memory (due to kernel)	uses the least memory (since no kernel)
can easily add more tasks	harder to do so
can easily add more devices (drivers)	harder to do so
easier to maintain if an off-the-shelf OS (cf. custom OS)	
may have a license fee	
	less migratable (due to less h/w abstraction)
	easier to debug[1]

Windows XP ≈ 1.5 Gbyte
Windows Vista ≈ 20 Gbyte + 15 Gbyte free
Windows 7 ≈ 16-20 Gbyte
Windows 8 ≈ 16-20 Gbyte
Windows 10 ≈ 16-20 Gbyte

The "kernel" provides:
1. scheduling of tasks
2. synchronization of tasks
3. interrupt handling/pre-emption
   - context switching (save-restore)
4. multi-tasking

[1] Unless program is very complicated (then should have used an OS anyway).

Definition:

A real-time operating system (RTOS) is an operating system that facilitates application programs to meet real-world timing constraints.

It must include multi-tasking and pre-emption.

Compare RTOS to OS

RTOS	OS
small kernel (i.e., small memory footprint)	not important
extensible: memory footprint defined by only what the user needs	less so
modular	less so
must meet timing constraints	no
low and predictable interrupt latency	no
reduced time when interrupts disabled e.g. during context switching e.g. during high-priority interrupt service routine time has an upper bound	less so
multi-tasking	yes
pre-emptible	some are

Compare Off-the-Shelf RTOS to Custom RTOS

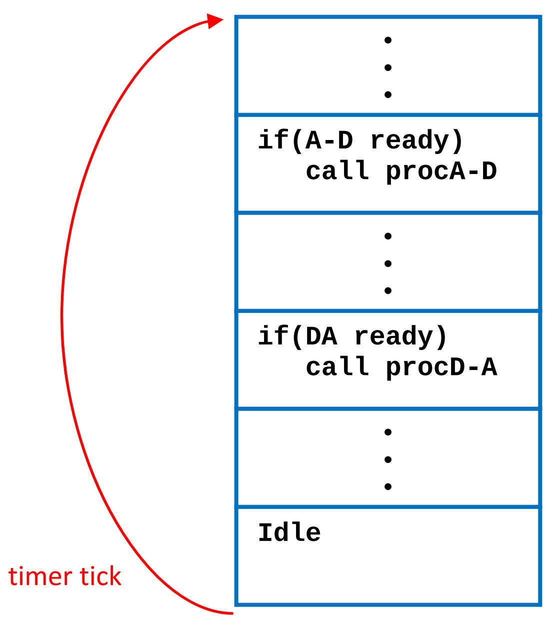
Off-the-Shelf RTOS	Custom RTOS
not as efficient	highest efficiency
technical support available	no
easier to maintain	less so, and lots of up-front work to make RTOS

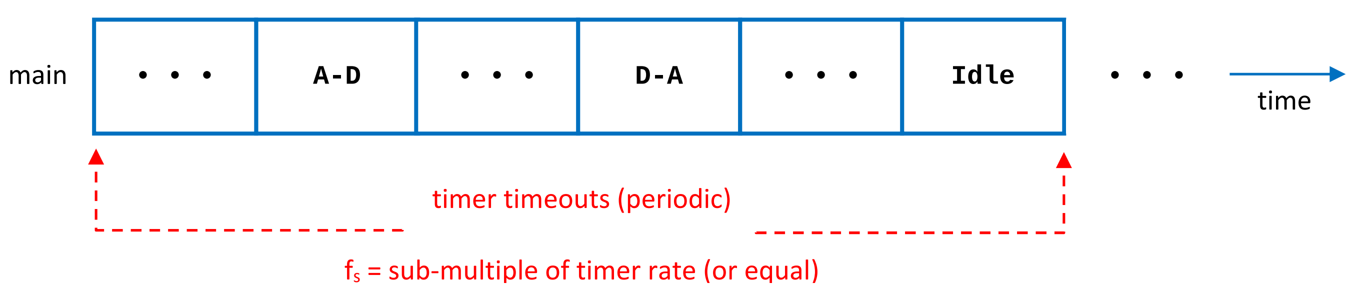
Firmware Architectures for Real-Time Embedded Systems

1. super loop: main task with polling and calls to functions
2. background task with event-based interrupts: interrupts cause their corresponding interrupts service routines to run
3. multi-tasking with RTOS

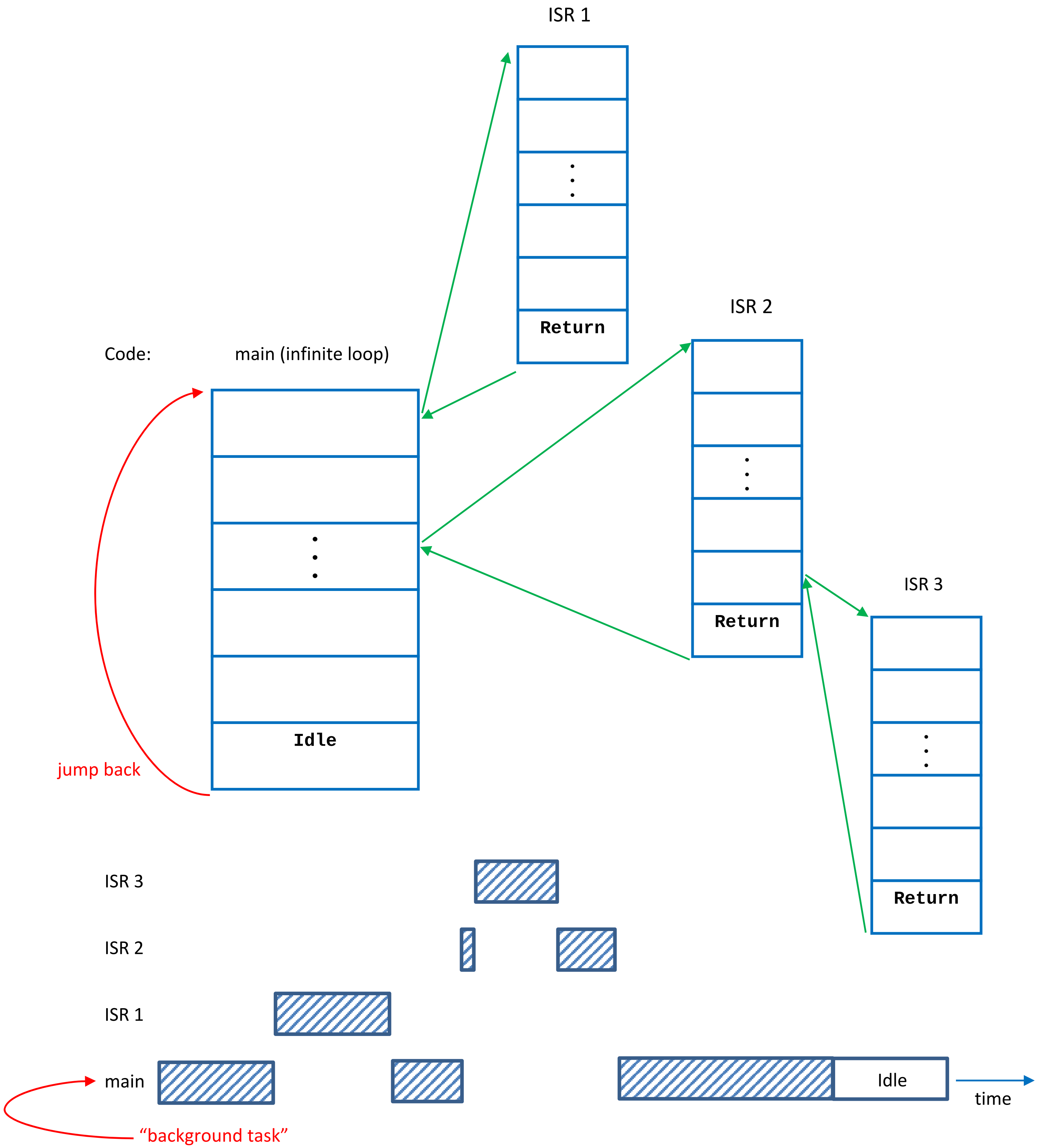
Super-Loop

Code: main (infinite loop)





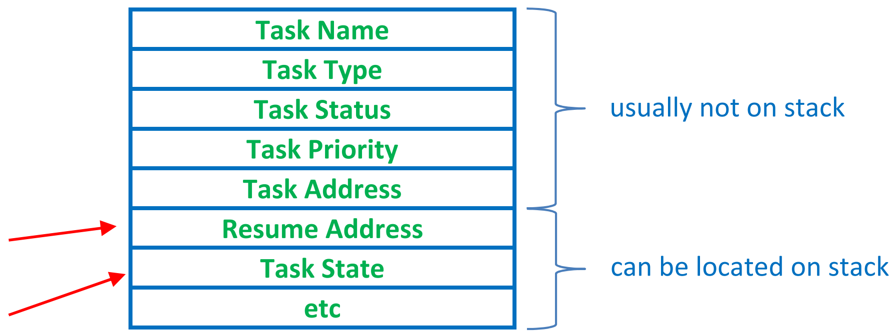
Background Task with Event-Based Interrupts



RTOS Concepts

Task

Basic unit of programming that is under control of an OS. A structure called a TCB (Task Control Block) is used to manage the task.



Multi-Tasking

In general, a CPU executes one instruction at a time. Via shared use of the CPU, multiple tasks can be effectively run "simultaneously"

.

Scheduling

Tasks in a multi-tasking program need to be scheduled in some fashion to share the CPU resource.

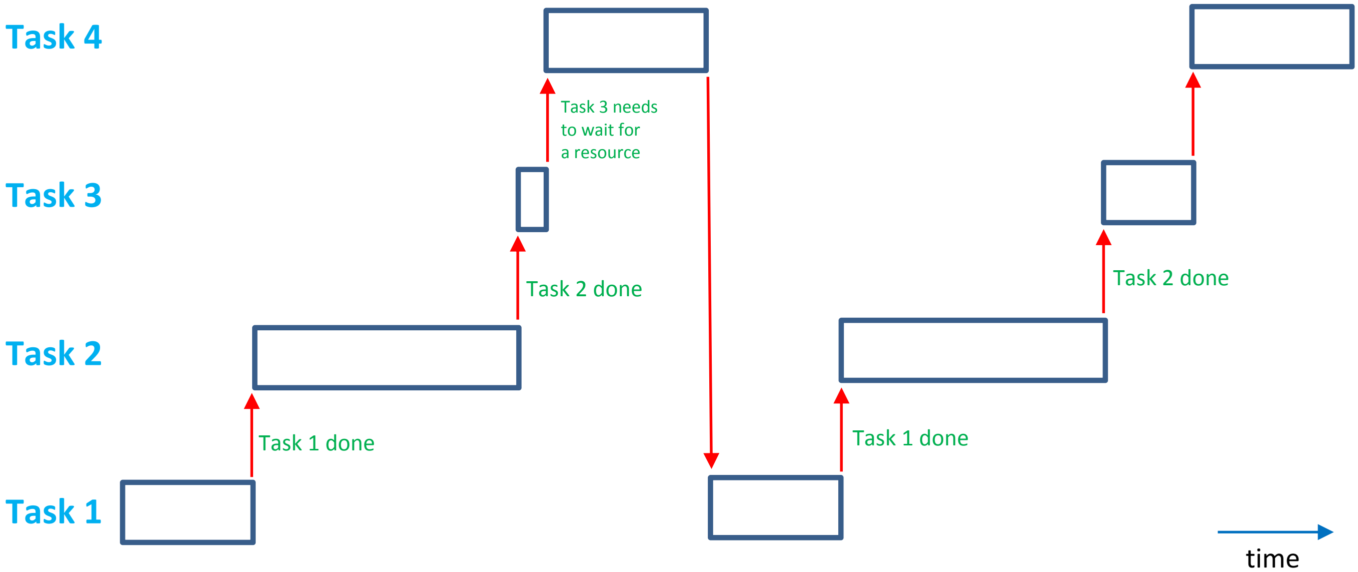
There are three basic methods of multi-task scheduling:

1. *cooperative* – In this method, if a task has finished running or is waiting for some event, it voluntarily relinquishes control of the processor so that other tasks can run. The tasks are, in general, "non-real-time" tasks.

2. *time-sharing* – In this method, each task receives a slice of time to use the CPU. A task is forced to relinquish control of the processor once it has run for its allotted time. The tasks share the CPU resource. The tasks are, in general, "non-real-time" tasks.

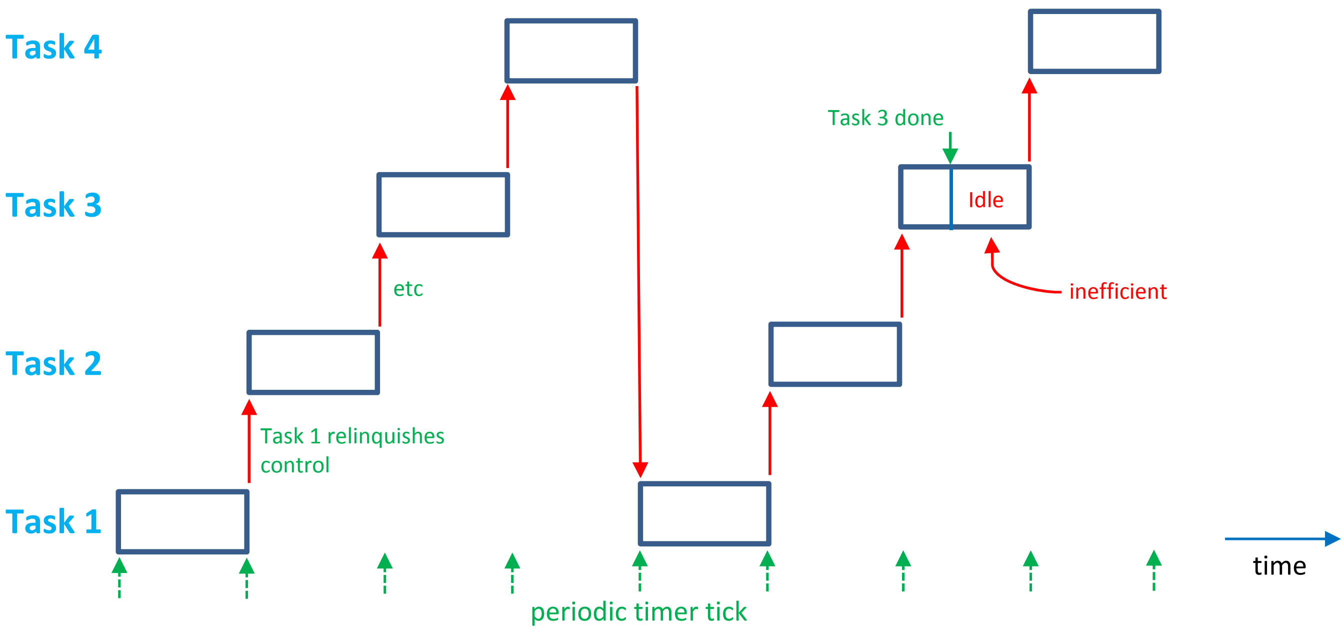
Co-Operative Scheduling

Task relinquishes control only when done or if waiting for a resource to free up or become ready. There is no idling.



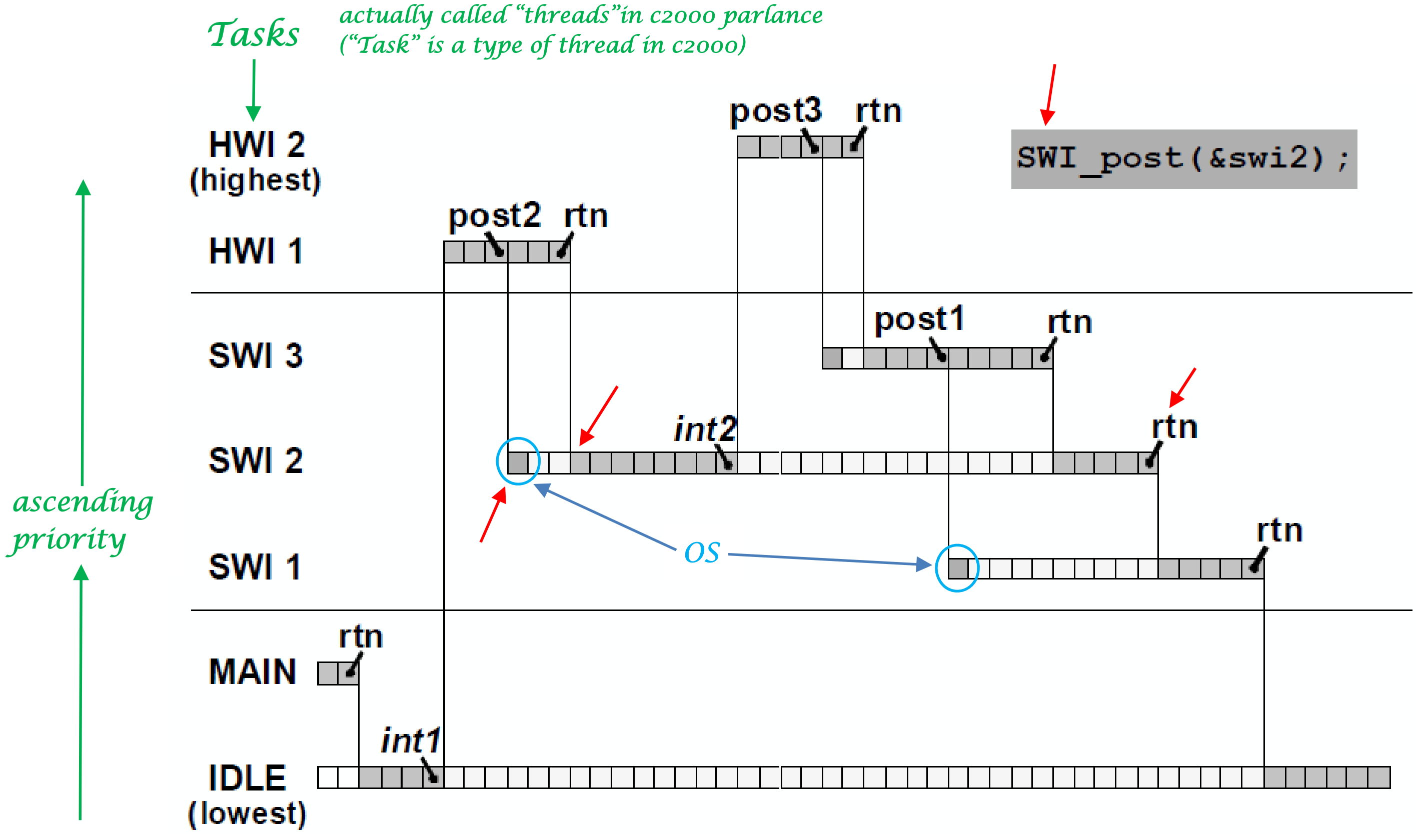
Time-Sharing Scheduling

Task relinquishes control even if not finished or keeps control to end of time-slice even if idle.



Period big enough so that context switch time not significant, but small enough to maintain some semblance of "simultaneous" tasks.

Pre-Emptive Scheduling *(c2000 nomenclature is shown)*



Texas Instruments RTOS

Called “**SYS/BIOS**” (formerly called “DSP/BIOS)

- no license fee
- fully-supported by TI

It is:

- a pre-emptive, multi-threading real-time operating system

It has:

- a scalable, real-time kernel
  - consumes only the memory space required to meet your use
- low interrupt latency

It can:

- schedule tasks where the user has set up their priorities
- set up timer-based periodic threads
- set up interrupt-driven threads

It includes:

- configuration control
  - e.g. helps user allocate memory sections
  - e.g. sets up interrupt table
  - e.g. sets up start-up sequence
- real-time scheduler
  - schedules your pre-emptive threads
- real-time communication
  - facilitates two-way communication between your tasks
    - synchronization:
      - control order of execution
      - control access to shared resource, e.g. data buffer
  - facilitates two-way communication between your application and PC host
- real-time analysis
  - your application can run unimpeded while debug data is displayed

Definition:

A thread (in TI parlance) is any independent stream of instructions executed by the processor.

Your application is a collection of threads each of which performs a modularized function.

Types of Threads in SYS/BIOS

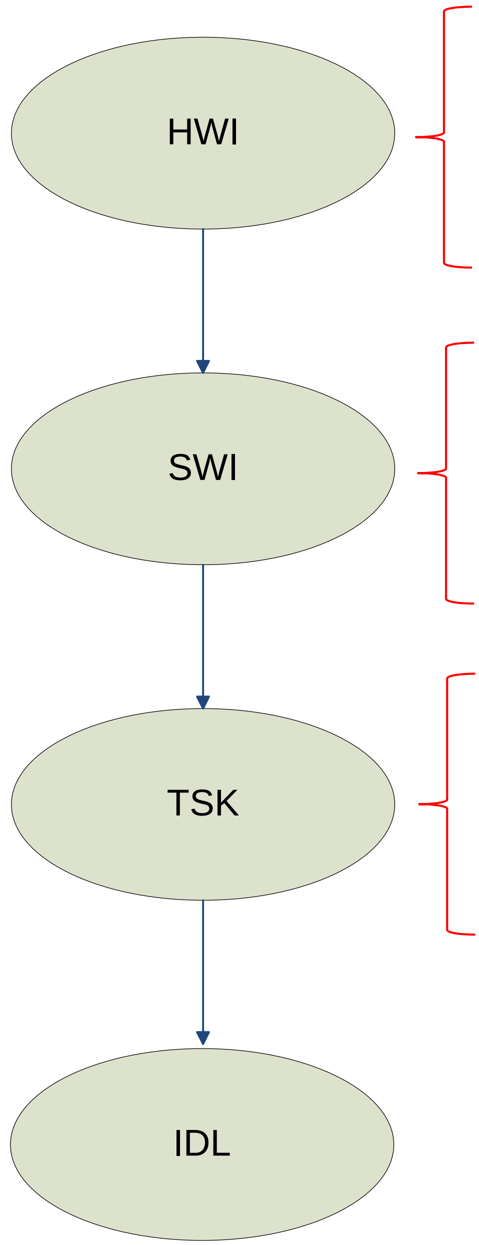
Thread Acronym	Thread	Description
HWI	Hardware Interrupt	• triggered by hardware interrupt: o e.g. A-D result ready, external event, etc • always runs to completion • context saved on system stack • could be interrupted by another HWI (only if interrupts were re-enabled by original HWI)

SWI	Software Interrupt	• triggered programmatically:
		    o i.e., by calling API such as “post”
		• always runs to completion
		    o but can be interrupted by HWI or pre-empted by another SWI
		• context saved on system stack

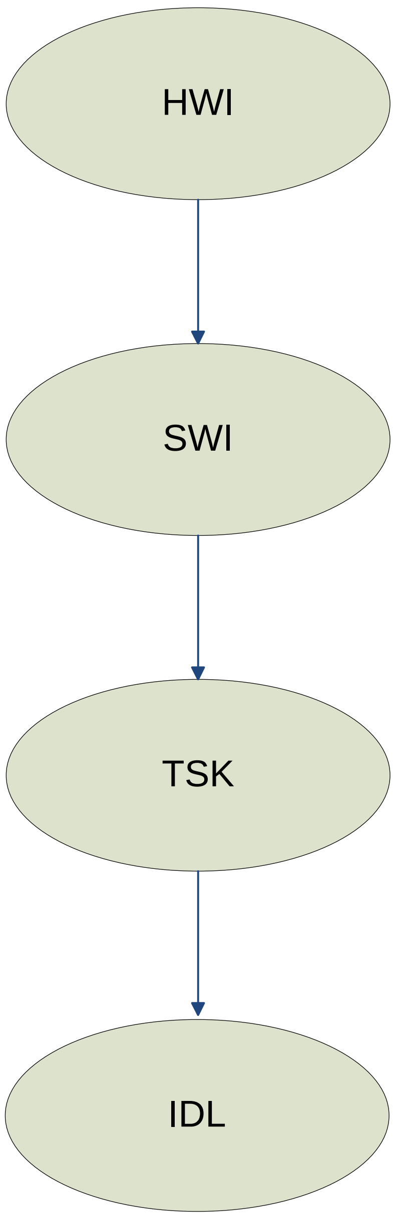
TSK	Task	• triggered programmatically
		• does not have to run to completion
		    o can be blocked until resource available
		    o can be interrupted by HWI or pre-empted by SWI
		• context saved on separate stack (one per task)
		• inter-task communication and synchronization available via:
		    o semaphores
		    o events
		    o mailboxes

IDL	Idle (Background Task)	• one continuous loop • can be interrupted by HWI or pre-empted by SWI or TSK

Priorities of Threads in SYS/BIOS



Choosing Which Type of Thread to Use for What Part of Your Application



HWI

- ➔ use for most critical response time to real-time events

SWI

- ➔ use to perform "follow-up" activity to HWI
- ➔ SWI can be started by HWI calling "post" API

TSK

- ➔ use if there are complex interdependencies and data sharing requirements
  - o TSKs have synchronization APIs available to use
- ➔ "pend"ing TSK can be *unblocked* by calling an API to "post" a "semaphore"

IDL

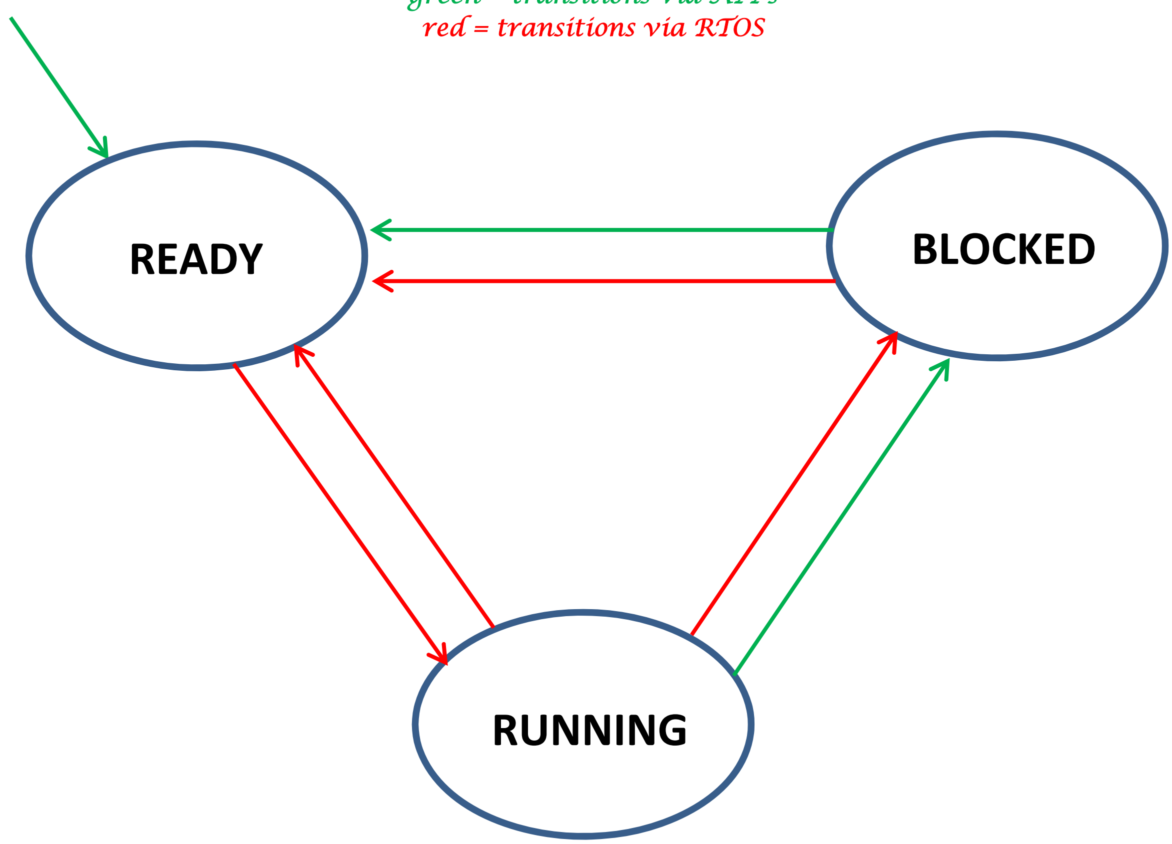
- ➔ infinite loop – use for background, low-priority, non-real-time work
  - o e.g. housekeeping
- ➔ use for transferring data to host, e.g. for debug and development
- ➔ use for low power modes if desired

Task States[2]

State	Description	C Code Name	enum#:
READY	The task is scheduled for execution, but not yet running.	Task_Mode_READY	1
RUNNING	The task is executing.	Task_Mode_RUNNING	0
BLOCKED	The task is not allowed to execute until particular event occurs.	Task_Mode_BLOCKED	2
TERMINATED	The task has been ended and does not execute again.	Task_Mode_TERMINATED	3

Task State Transitions[3]

green = transitions via API's
red = transitions via RTOS



[2] There is also an INACTIVE state, but we will not consider it.
[3] Not shown: API's Task_yield() (RUNNING-to-READY transition) and Task_sleep() (RUNNING-to-BLOCKED transition)

Hypothetical Example for Choosing Which Type and Priority of Threads to Use

Assumptions:

- on-board A-D is triggered once every 10 μsec by on-board timer
- when there are 128 samples collected, a 128-point FFT is computed, then the magnitude-squared of each bin is computed
- each bin is searched for a signal that exceeds some threshold and if so turns on green LED
- on-board temperature sensor is checked once in a while to ensure the processor is not overheated; if it is, turn on red LED
- there is a numerical keypad which generates an external interrupt each time a key is pressed – the digit pressed indicates to which radio frequency band to tune the analog front-end