CIS 721 - Real-Time Systems

Lecture 20: FreeRTOS

Mitch Neilsen
neilsen@ksu.edu

Outline

- **Real-Time Operating Systems**
  - **Commercial**
    - **VxWorks**
    - **ThreadX**
  - Open Source
    - BrickOS
    - FreeRTOS
      - For multiple platforms

VxWorks RTOS

- Preemptive, priority-based scheduling with round-robin at the same priority level
- 256 priority levels, 0 = highest, 255 = lowest
- Idle task at priority level 255
- Semaphores with priority inheritance
- Commercial RTOS with Tornado development environment

ThreadX RTOS

- Preemptive, priority-based scheduling with **preemption thresholds** and round-robin at the same level.
- 32 priority levels, 0 = highest, 31 = lowest
- Semaphores with priority inheritance
- Commercial RTOS
- http://www.rtos.com

Outline

- **Real-Time Operating Systems**
  - Commercial
    - VxWorks
    - ThreadX
  - **Open Source**
    - **BrickOS**
    - FreeRTOS
      - For multiple platforms

BrickOS Real-time Operating System

- Lego Mindstorms Robotics Invention System
  - Hardware: Lego **RCX 1.0** or RCX 2.0
  - Software: RIS 2.0, RoboLab 2.5.4, **BrickOS**, LeJOS, etc.
- BrickOS Real-Time Operating System
  - Preemptive, priority-based scheduling with round-robin at same priority level
  - 20 priority levels, 1 = lowest, 20 = highest
  - Simple semaphores – no priority inheritance
  - Open source: brickos.sourceforge.net

Lego Mindstorms Hardware
Robotics Command EXplorer (RCX 1.0)

- Hitachi H8/300 microcontroller
- 16 KB ROM, 16 KB RAM
- 16 MHz system clock
- 16 bit free-running timer
- 3 sensor ports (for light, touch, sound, temperature, etc.)
- 3 motor (actuator) ports
- IR transmitter/receiver
- LCD screen

Lego Mindstorms Software: Robotics Invention System (RIS 2.0)

- Lego Mindstorms:
  - Main site: http://mindstorms.lego.com
  - News site: http://news.lugnet.com/robotics/rcx/
- Real-time operating systems:
  - Erika: http://erika.sssup.it/index.shtml
  - BrickOS: http://brickos.sourceforge.net
  - Lejos: http://lejos.sourceforge.net
- Programming IDE:
  - Bricx Command Center: http://bricxcc.sourceforge.net
  - Not Quite C: http://bricxcc.sourceforge.net/nqc

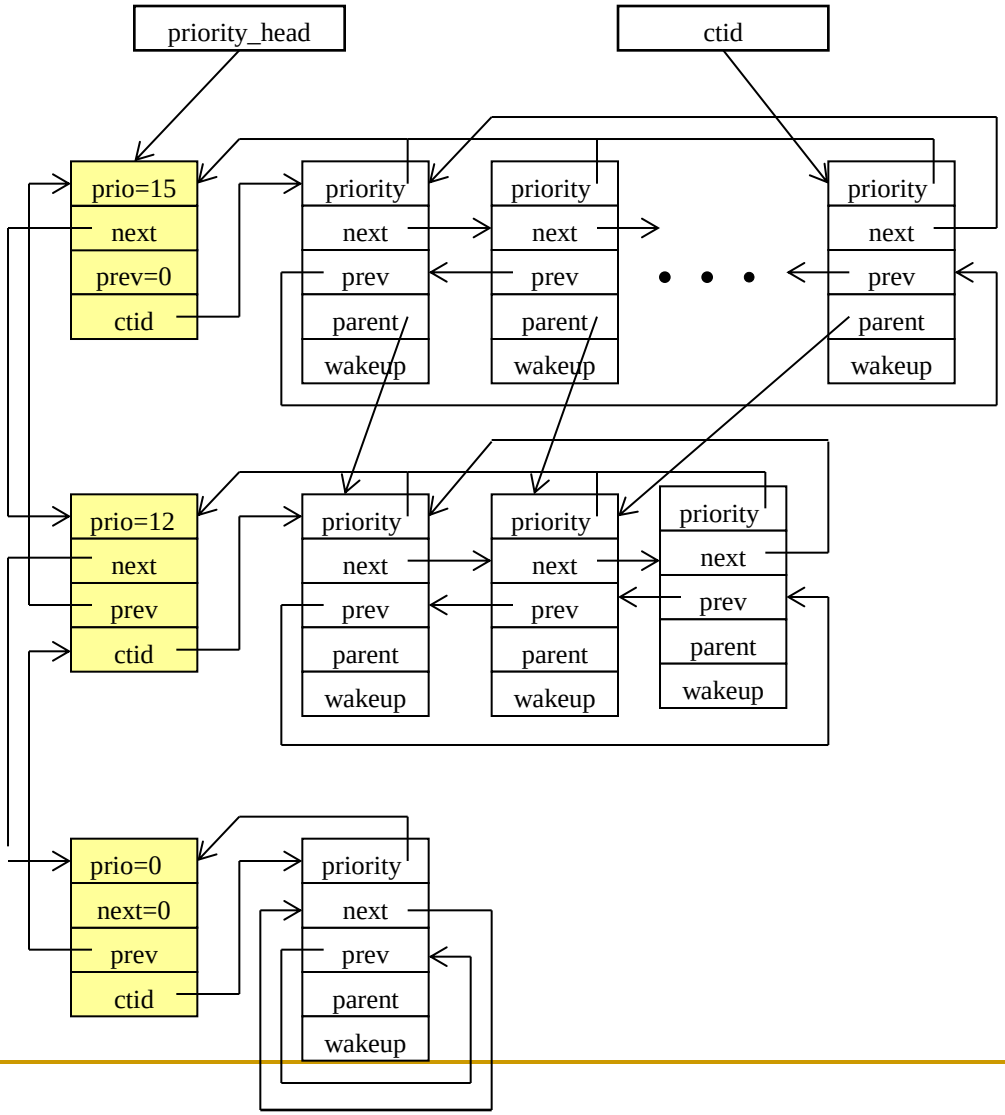
BrickOS Operating System

- Formerly, called LegOS.
- User-level and kernel-level programs are written in C/C++.
- Preemptive priority-based round-robin scheduling algorithm with a time quantum of 20 milliseconds.
- First fit dynamic memory allocation.
- Tasks (threads) in the BrickOS go through states sleeping, running, waiting, zombie, and dead.
- Source available at: http://brickos.sourceforge.net.

BrickOS Tasks

- Divided into kernel-level and user-level tasks.
- Tasks are sorted into circular queues based on their priorities. Tasks at the same priority level are put into the same circular doubly-linked queue.
- The head of each queue contains a pointer to the previous and next task, the priority level, and the first task in the chain.
- A **TCB (Task Control Block)** contains the stack pointer, state information, user and kernel flags, and a pointer to the priority chain.

Priority Queues for TCBs



Priority-Based Scheduling Algorithm

- Each task is given a default time slice (quanta) of 20ms to run before being interrupted by the OS to check if another task is ready to run.
- In choosing the next task to run, the OS searches through each task queue, beginning with the highest priority queue, and runs the first READY task found.
- Tasks at the same priority level are executed in a fair round-robin fashion.

Task Creation Functions

- (int)tid_t **execi (&PROCESS_NAME, int argc, char **argv, priority_t priority, size_t stack_size);**
  Place function **PROCESS_NAME** into the Process queue, returns the Process's assigned Thread ID or -1 if thread failed to start
- *Example:* **execi(&RunMotor, 0, NULL, 1, DEFAULT_STACK_SIZE);**
  Starts function RunMotor as a thread, with no parameters passed (0, NULL), at the lowest priority (1), with the DEFAULT_STACK_SIZE of 512 being used. The thread ID is passed back, but in this case it is not being stored, but it could have been assigned to a variable type tid_t to keep track of various thread.
- void **exit (int code);**
  Exits Process, returning **code**

Task Termination

- void **kill (tid_t TID);**
  Kill Thread associated with **(int)tid_t TID** as assigned when it was started by execi ( )
- void **killall (priority_t p);**
  Kill all Processes with a Priority less than **p**
- void **shutdown_task (tid_t TID);**
  Shutdown Thread associated with **tid_t TID** wakeup_t wait_event(wakeup_t(*wakeup) (wakeup_t), wakeup_t data);
  Suspend current Process until Event wakeup function is non-null:
  unistd.h, tm.c

Task Wakeup On Event

- void **yield ( );**
  -- Yield the rest of the current Task's timeslice
- int **sleep(int sec);**
  int **msleep(int msec);**
  -- Pause for an interval of time before executing next commands in current program thread, other program threads will continue to execute commands. Gives up CPU time for other threads.
- wakeup_t **wait_event (wakeup_t (* wakeup)(wakeup_t), wakeup_t data);**
  Suspend task until wakeup function returns a non-NULL
  Parameters:
  - **wakeup** the function to be called when woken up
  - **data** the wakeup_t structure to be passed to the called function

Task Priority Levels

Predefined Priority Levels:

- PRIO_LOWEST = 1 The Lowest Possible Task Priority
- PRIO_NORMAL = 10 The Normal Priority Level
- PRIO_HIGHEST = 20 The Highest Task Priority

Example

```
#include <unistd.h>
#include <dbutton.h>
#include <dmotor.h>
int mDirection = 0;  int cDirection = 0;
int RunMotor() {
        motor_a_speed(250); motor_a_dir(fwd);
        while (!shutdown_requested())
        {
                if(mDirection == cDirection)
                        cputs("SAME");
                else
                {
                        cputs("STOP"); motor_a_speed(0);
                        msleep(2000);
                        if (mDirection==1)
                                motor_a_dir(rev);
                        else
                                motor_a_dir(fwd);
                        motor_a_speed(255);
                        cDirection=mDirection;
                }
        }
        return 0;
}
```

Example (cont.)

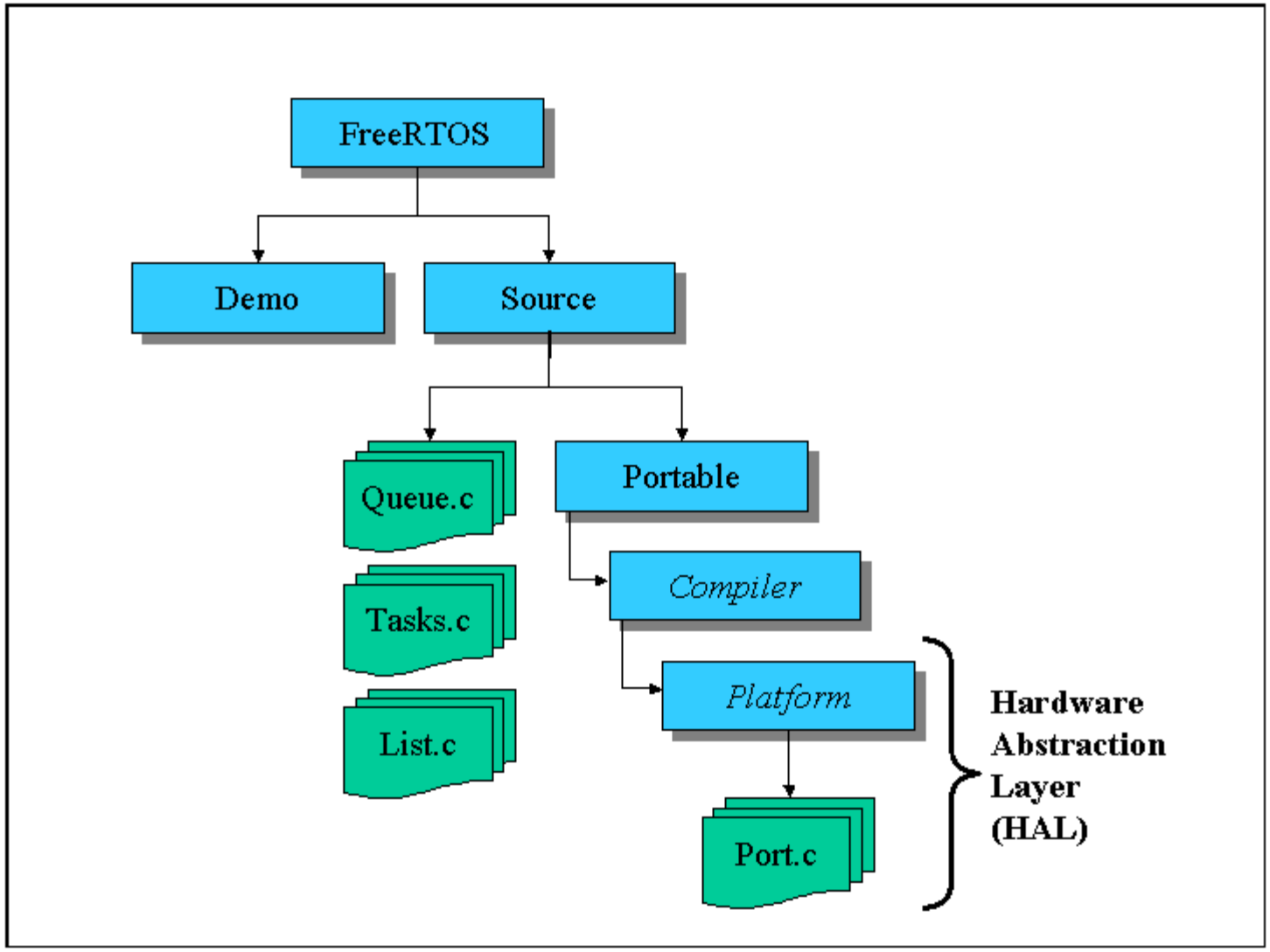
```
int CheckButton() {
        while (!shutdown_requested())
                if (PRESSED(dbutton(),BUTTON_PROGRAM))
                {
                        cputs("pressd");
                        mDirection = 1-mDirection;
                        msleep(1000);
                        if (mDirection==0)
                                cputs("Zero");
                        else
                                cputs("One");
                }
        return 0;
}

int main() {
        execi(CheckButton, 0, NULL, 10, DEFAULT_STACK_SIZE);
        execi(RunMotor, 0, NULL, 10, DEFAULT_STACK_SIZE);
        while(!shutdown_requested())
                msleep(1000);
        return 0;
}
```

Outline

- **Real-Time Operating Systems**
  - Commercial
    - VxWorks
    - ThreadX
  - **Open Source**
    - BrickOS
    - **FreeRTOS**
      - **For multiple platforms**

FreeRTOS File Structure



FreeRTOS Overview

Preemptive, priority-based scheduling or **cooperative scheduling:**

- **Cooperative scheduling** does not implement a timer based scheduler decision point – processes pass control to one another by yielding. The scheduler interrupts at regular frequency simply to increment the tick count. Useful for small, memory constrained systems.
- **Preemptive, priority-based scheduling algorithm** schedules the highest priority task first. Where more than one task exists at the highest priority, tasks are executed in round robin fashion.

Inter-Process Communication (IPC)

- **Inter-Process Communication:** Tasks within FreeRTOS can communicate with each other through the use of queuing and synchronization mechanisms:
  - **Queuing:** Inter-process communication is achieved via the creation of queues. Most information exchanged via queues is passed by value not by reference which should be a consideration for memory constrained applications. Queue reads or writes from within interrupt service routines (ISRs) are non-blocking. Queue reads or writes with zero timeout are non-blocking. All other queue reads or writes block with configurable timeouts.
  - **Synchronization:** FreeRTOS allows the creation and use of ***binary* semaphores**. The semaphores themselves are specialized instances of message queues with queue length of one and data size of zero. Because of this, taking and giving semaphores are atomic operations since interrupts are disabled and the scheduler is suspended in order to obtain a lock on the queue.

Blocking

- **Blocking and Deadlock Avoidance:** In FreeRTOS, tasks are either non-blocking or will block for a fixed period of time. Tasks that wake up at the end of timeout and still cannot get access to a resource must have made provisions for the fact that the API call to the resource may return an access failure notification. Timeouts on each block reduce the likelihood of resource deadlocks.
- **Critical Section Processing:** At the lowest level, critical section processing is handled by the disabling of interrupts. Critical sections within a task can be nested and each task tracks its own nesting count. However, it is possible to yield from within a critical section (in support of the cooperative scheduling) because software interrupts (SWI) are non-maskable and yield uses SWI to switch context. The state of interrupts are restored on each task context switch by the restoration of the interrupt (I) bit in the condition code register (CCR).

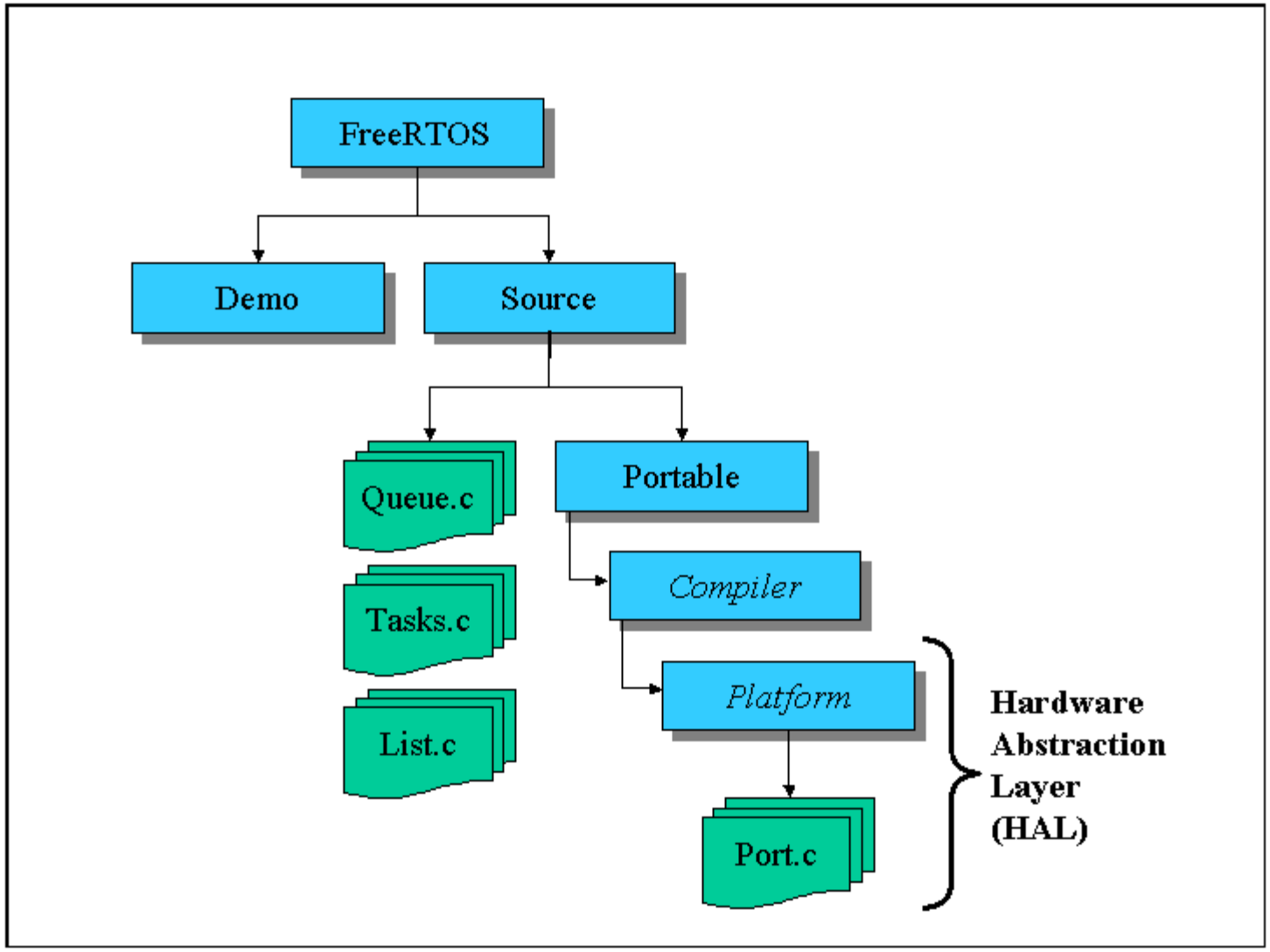
Suspending the Scheduler

- **Scheduler Suspension:** When exclusive access to the processor (MCU) is required without jeopardizing the operation of ISRs, the scheduler can be ***suspended***. Suspending the scheduler guarantees that the current process will not be pre-empted by a scheduling event while at the same time continuing to service interrupts.

Mutex vs. Binary Semaphore

- **Mutex** is only available from FreeRTOS V4.5.0 onwards.
  - **xSemaphoreHandle xSemaphoreCreateMutex(void)**
- A macro that creates a mutex semaphore by using the existing queue mechanism.
- Mutexes created using this macro can be accessed using the xSemaphoreTake() and xSemaphoreGive() macros.
- Mutexes and binary semaphores are very similar but have some subtle differences: **Mutexes include a priority inheritance mechanism**, binary semaphores do not. .

FreeRTOS File Structure



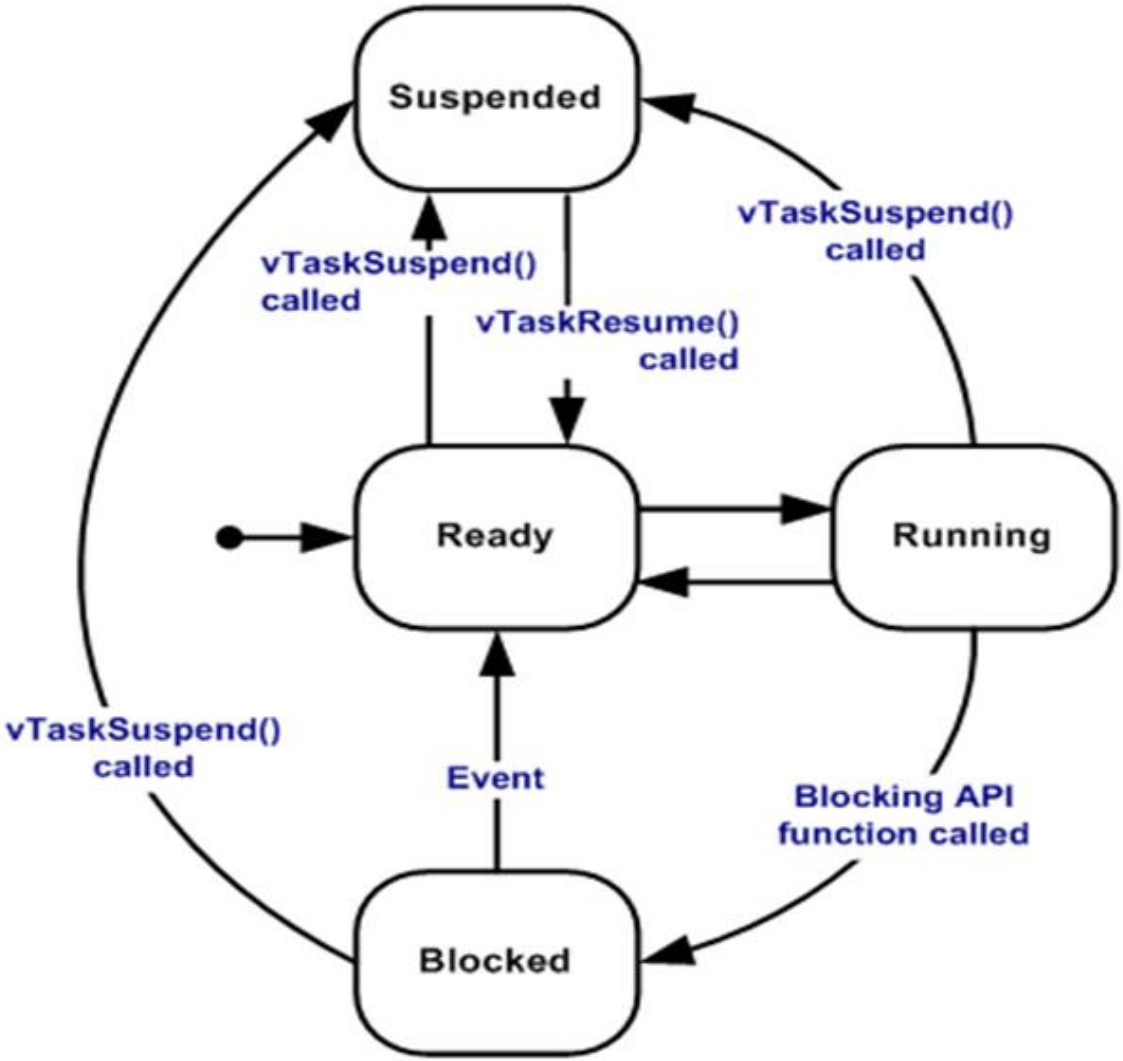
Hardware Independent Files

- Queue.c - queue/semaphore functions
- Tasks.c - scheduler and task functions
- List.c - list functions
- Timers.c - timer functions

Task Control Block (TCB)

Top of Stack	Pointer to last item placed on the stack for this task
Task State	List item that puts the TCB in the ready or blocked queues
Event List	List item used to place the TCB in event lists.
Priority	Task priority (0 = lowest)
Stack Start	Pointer to the start of the process stack
TCB Number	A debugging and tracing field.
Task Name	A task name
Stack Depth	Total depth of the stack in variables (not bytes)

Basic Process State Diagram

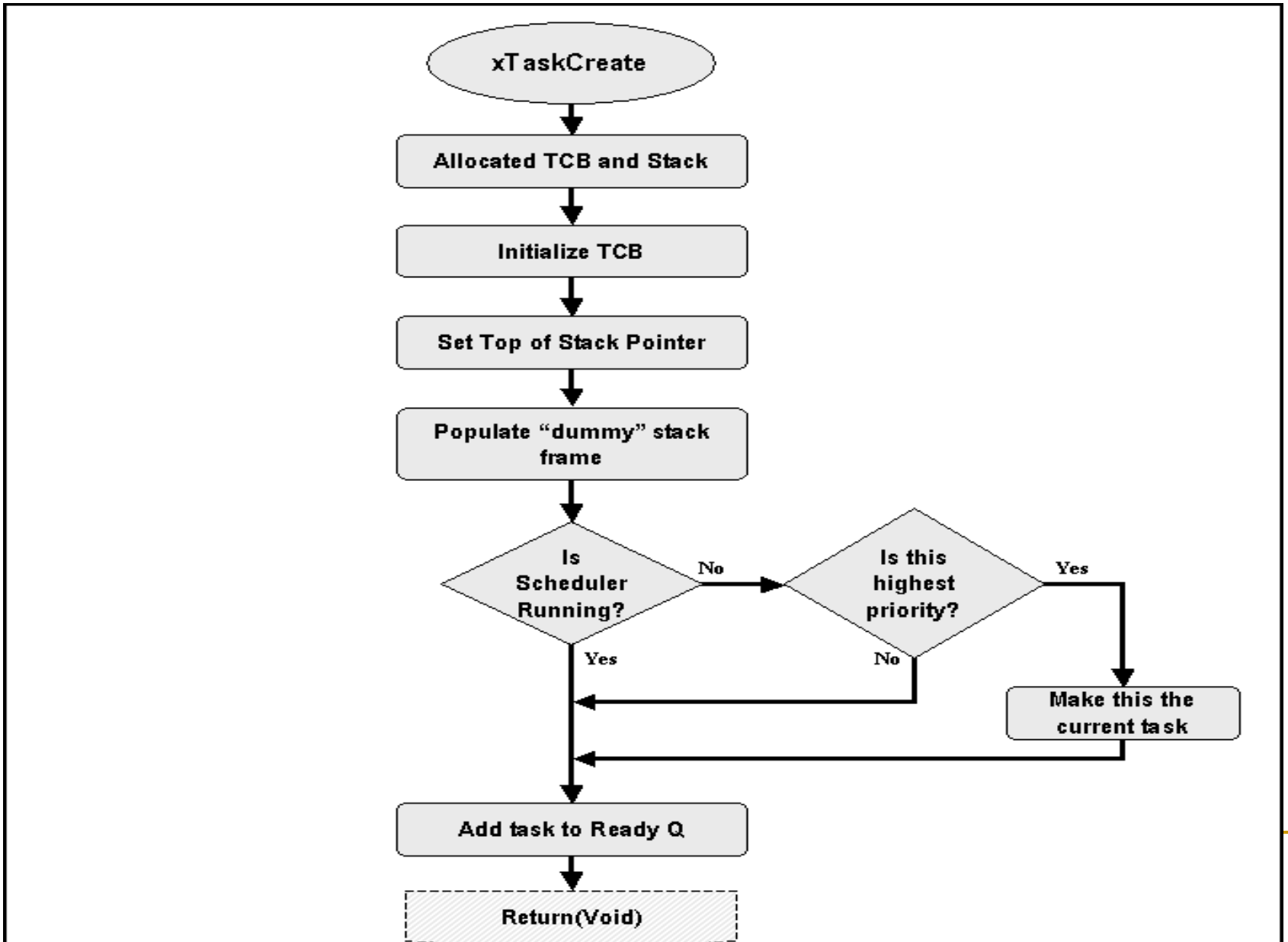


Valid task state transitions

FreeRTOS Task Creation

- The FreeRTOS kernel creates a task by instantiating and populating a TCB. New tasks are placed immediately in the Ready state by adding them to the Ready list.
- The Ready list is arranged **in order of priority** with tasks of equal priority being serviced on a round-robin basis. The implementation of FreeRTOS actually uses multiple Ready lists – one at each priority level. When choosing the next task to execute, the scheduler starts with the highest priority list and works its way progressively downward.
- The FreeRTOS kernel does not have an explicit “Running” list or state. Rather, the kernel maintains the variable pxCurrentTCB to identify the process in the Ready list that is currently running. pxCurrentTCB is therefore defined as a pointer to a TCB structure.

FreeRTOS Task Creation



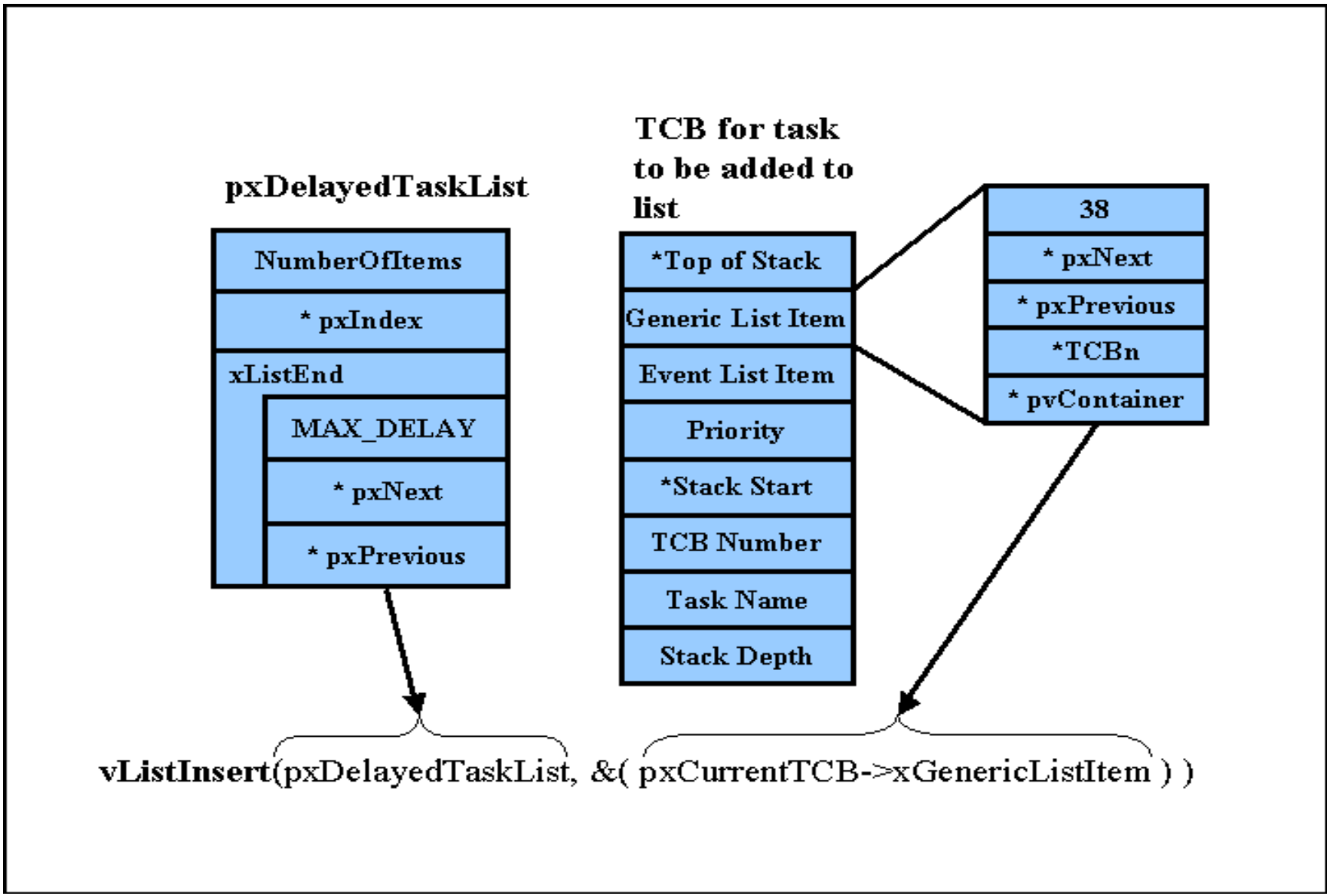
FreeRTOS Queues and Semaphores

- Tasks in FreeRTOS can be blocked when access to a resource is not currently available.
- The scheduler blocks tasks only when they attempt to read from or write to a queue that is either empty or full respectively. This includes attempts to obtain semaphores since these are special cases of queues.
- Access attempts against queues can be blocking or non-blocking. The distinction is made via the xTicksToWait variable which is passed into the queue access request as an argument. If xTicksToWait is 0, and the queue is empty/full, the task does not block. Otherwise, the task will block for a period of xTicksToWait scheduler ticks or until an event on the queue frees up the resource.

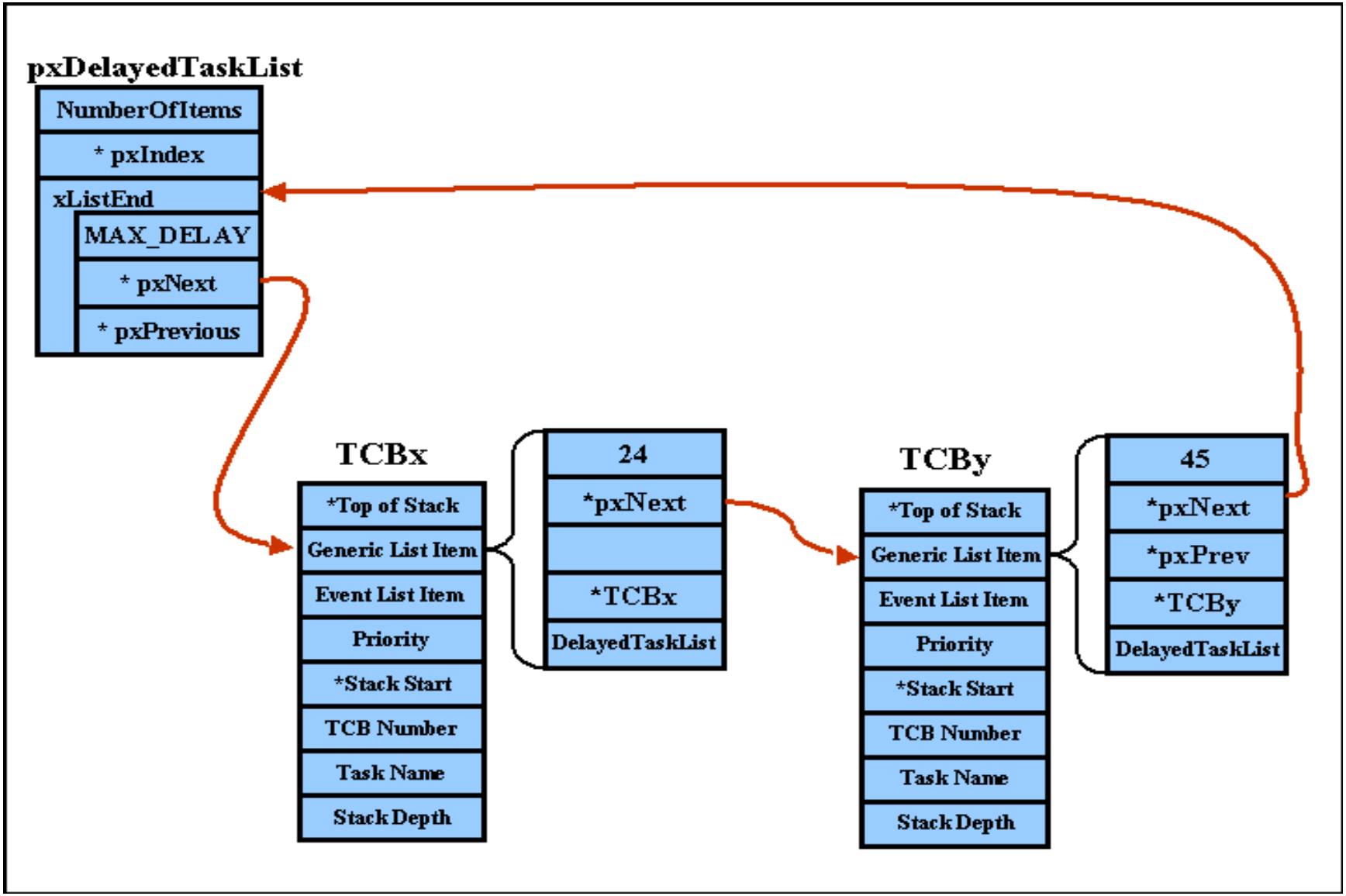
Lists Created By The Scheduler

FreeRTOSConfig.h	Lists Created
configMAX_PRIORITIES	ReadyTasksLists[0] : ReadyTasksLists[configMAX_PRIORITIES]
INCLUDE_vTaskDelete == 1	TasksWaitingTermination
INCLUDE_vTaskSuspend == 1	SuspendedTaskList
N/A	PendingReadyList
N/A	DelayedTaskList
N/A	OverflowDelayedTaskList

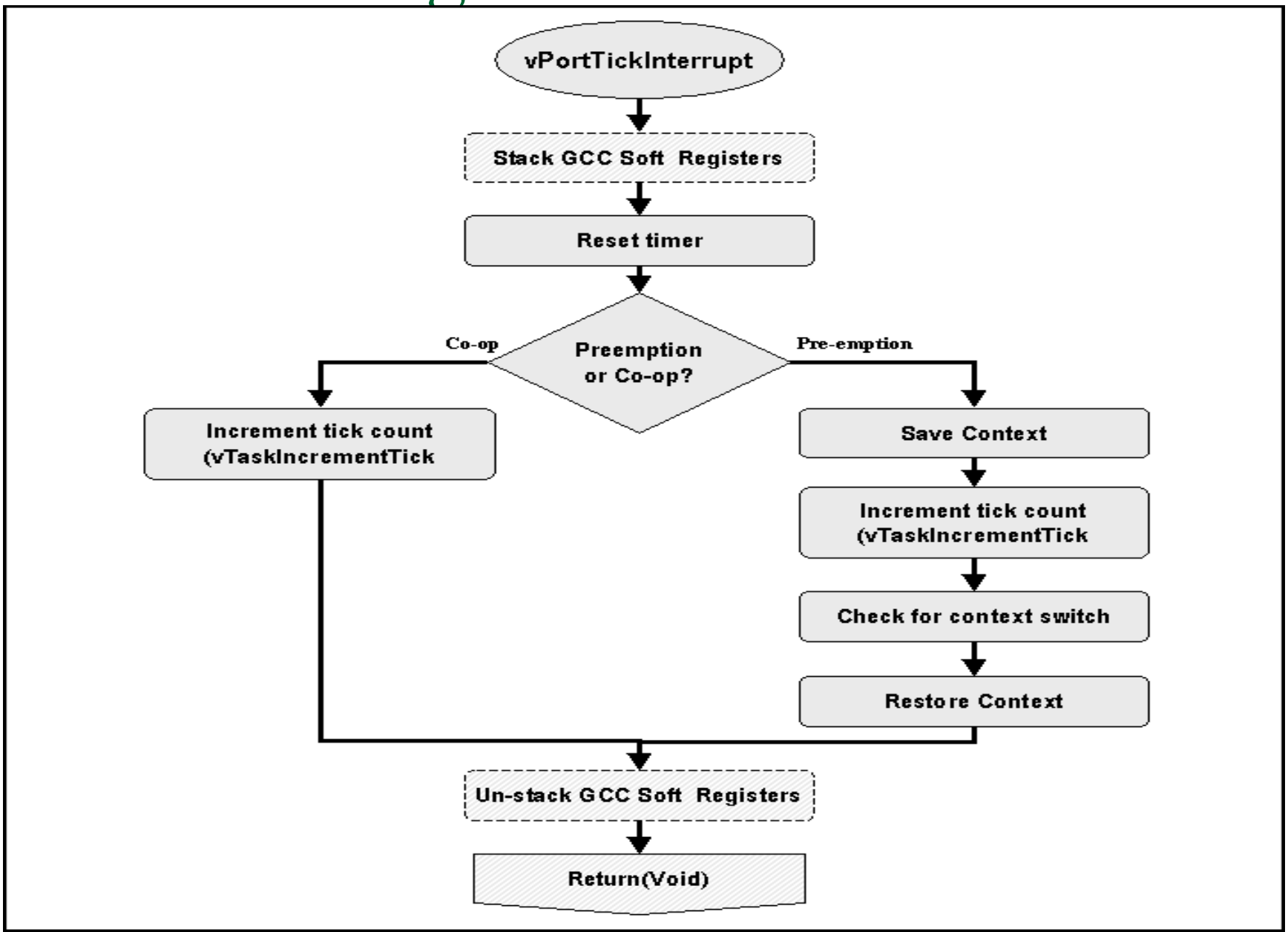
vListInsert



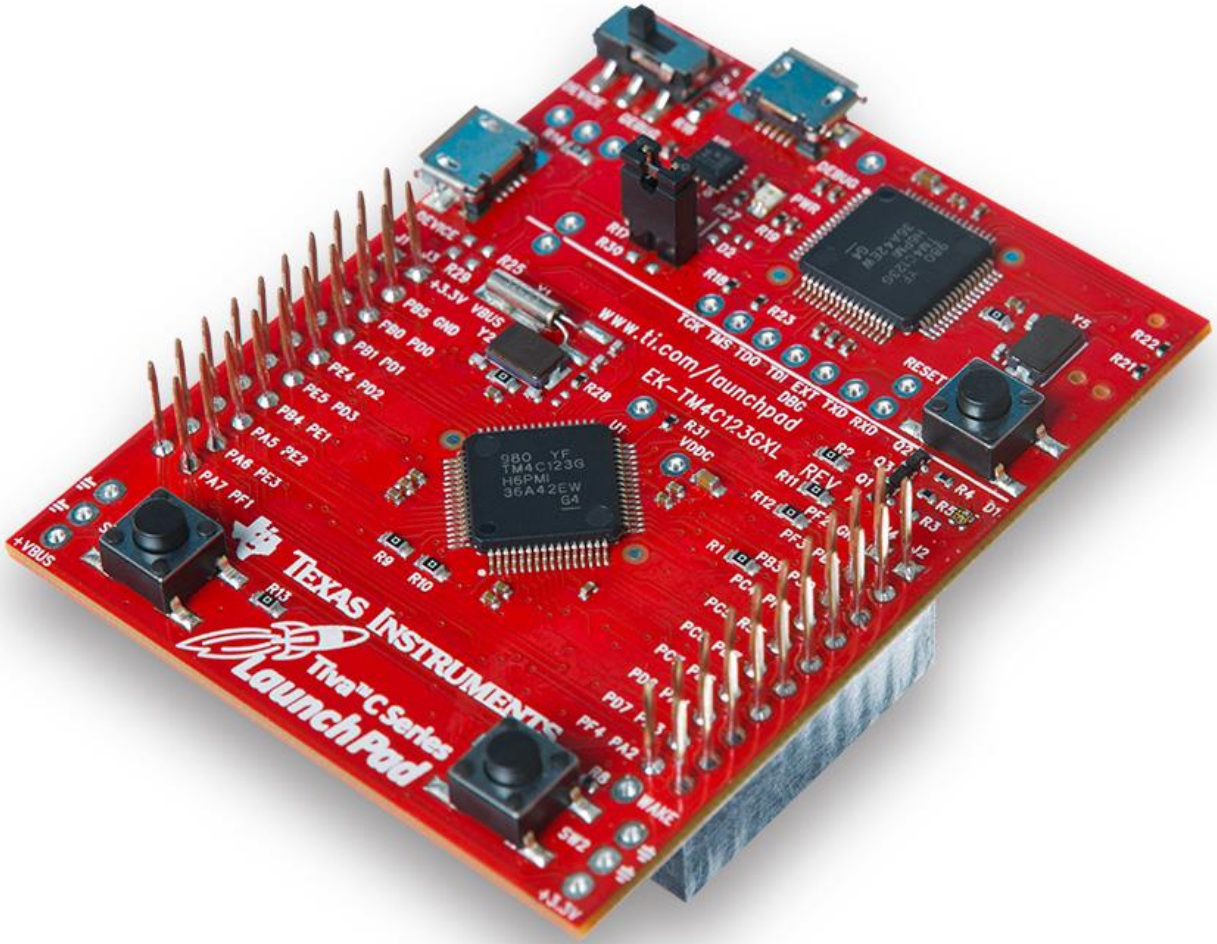
Example DelayedTaskList



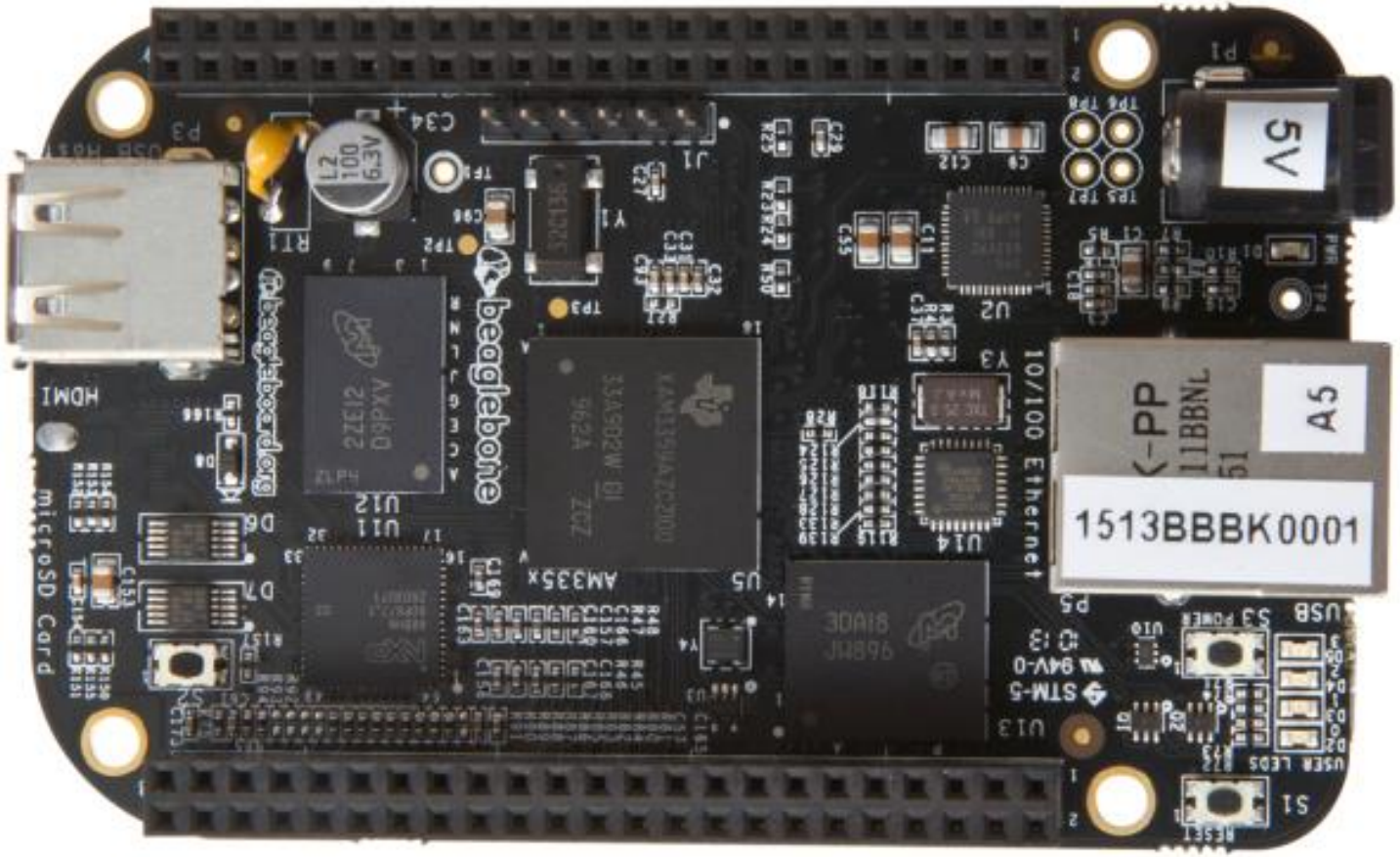
Scheduler Algorithm



TI Tiva C Series TM4C123G with FreeRTOS

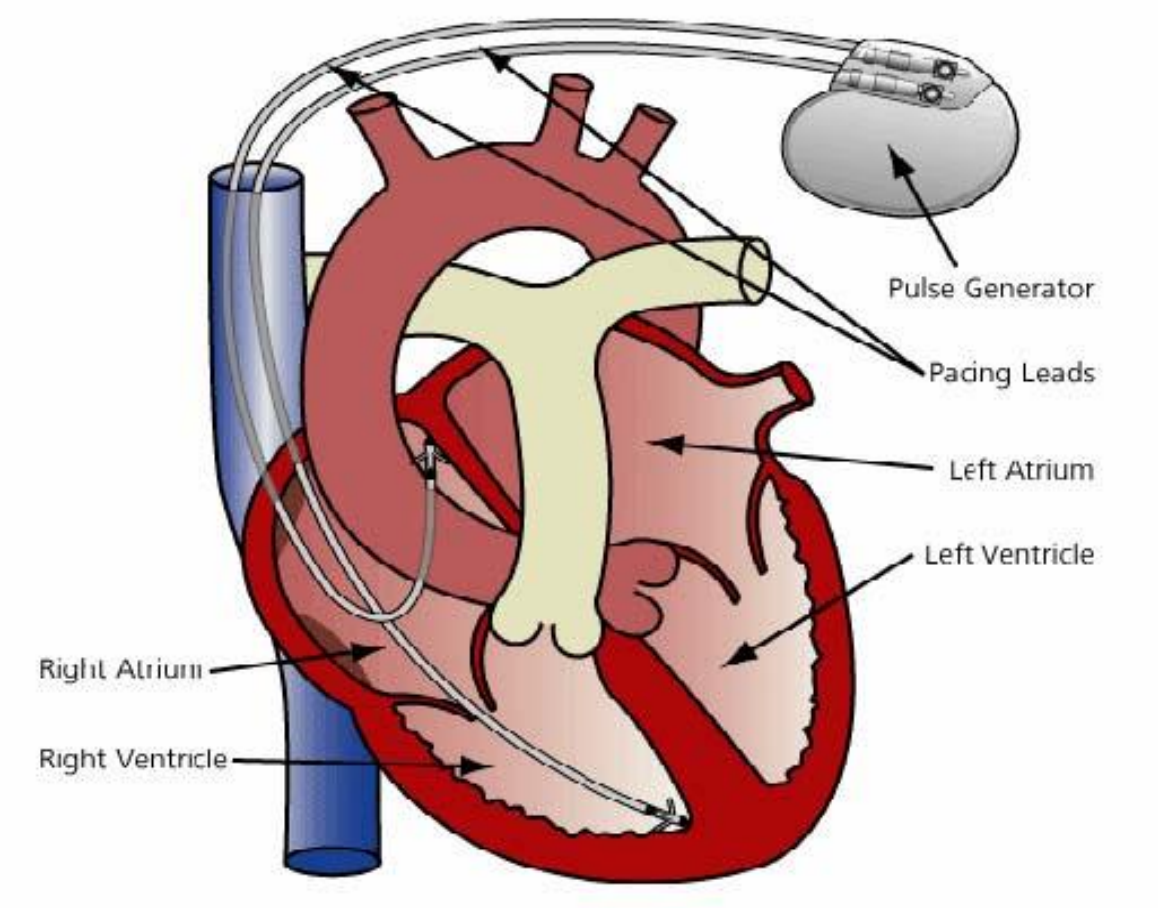


BeagleBone Black – beagleboard.org FreeRTOS or bonescript



PICKIT 3 Debug Express – Microchip FreeRTOS

Example: Academic Pacemaker



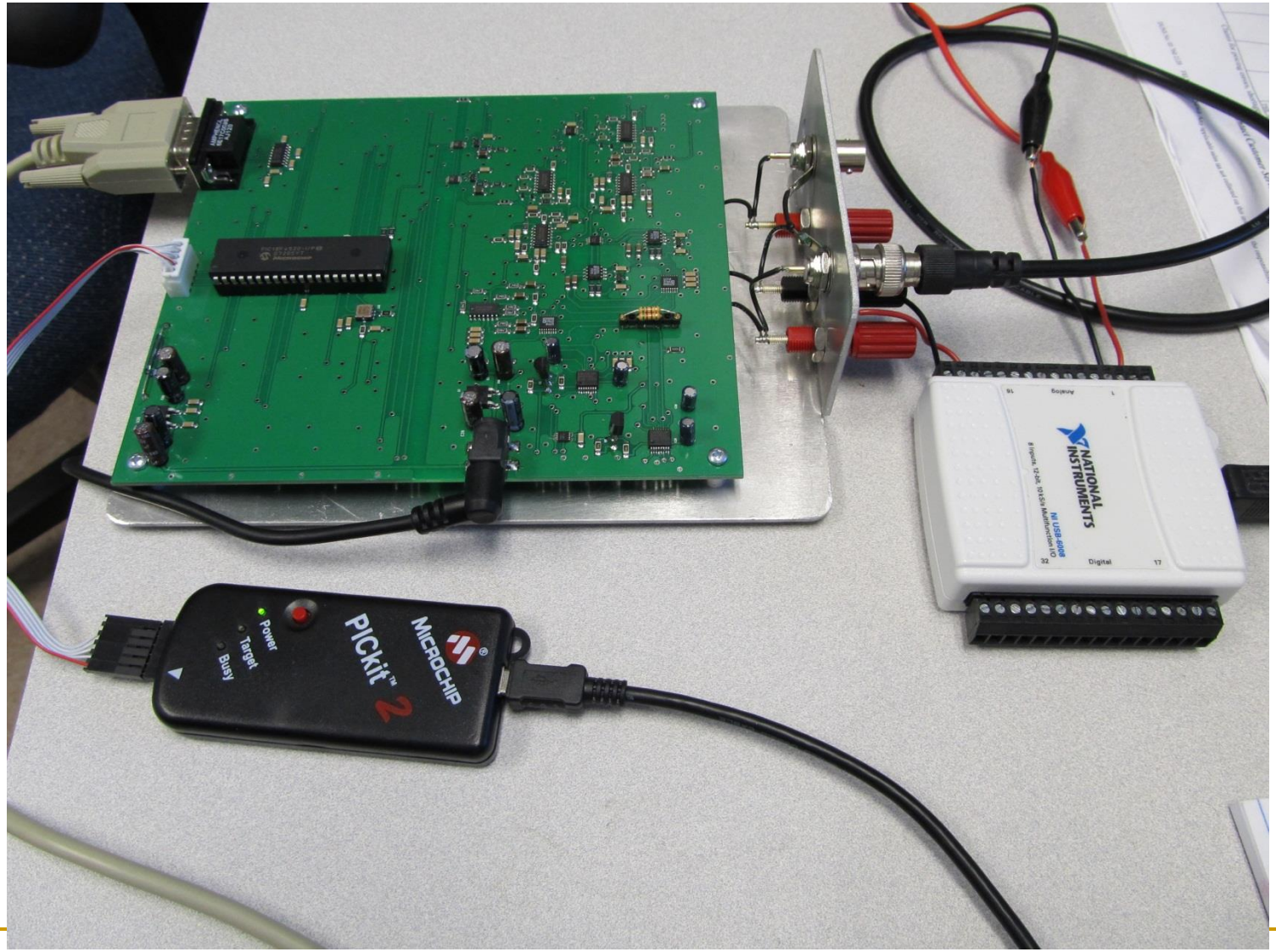
Pacemaker (Pulse Generator) [www.medtronic.com]

Pacemaker Pacing Modes *

Revise NASPE/BPEG generic code for antibradycardia pacing[3]				
I	**II**	**III**	**IV**	**V**
Chamber(s) paced	**Chamber(s) sensed**	**Response to sensing**	**Rate modulation**	**Multisite pacing**
O = None	**O = None**	**O = None**	**O = None**	**O = None**
A = Atrium	**A = Atrium**	**T = Triggered**	**R = Rate modulation**	**A = Atrium**
V = Ventricle	**V = Ventricle**	**I = Inhibited**		**V = Ventricle**
D = Dual (A+V)	**D = Dual (A+V)**	**D = Dual (T+I)**		**D = Dual (A+V)**

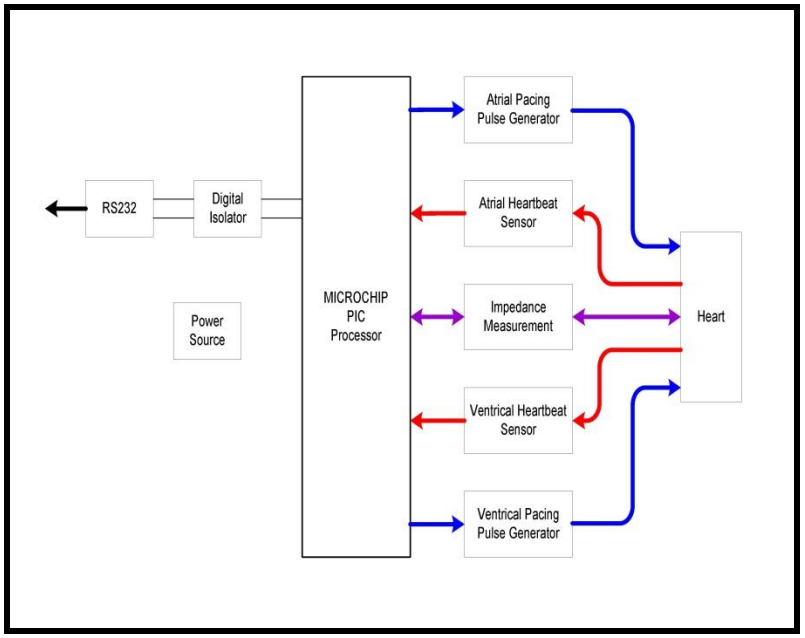
*** S. Serge Barold, Roland X. Stroobandt, and Alfons F. Sinnaeve, "Caridiac Pacemakers and Resynchronization Step-byStep: An Illustrated Guide", 2nd Edition, Blackwell Publ., Inc., 2010**

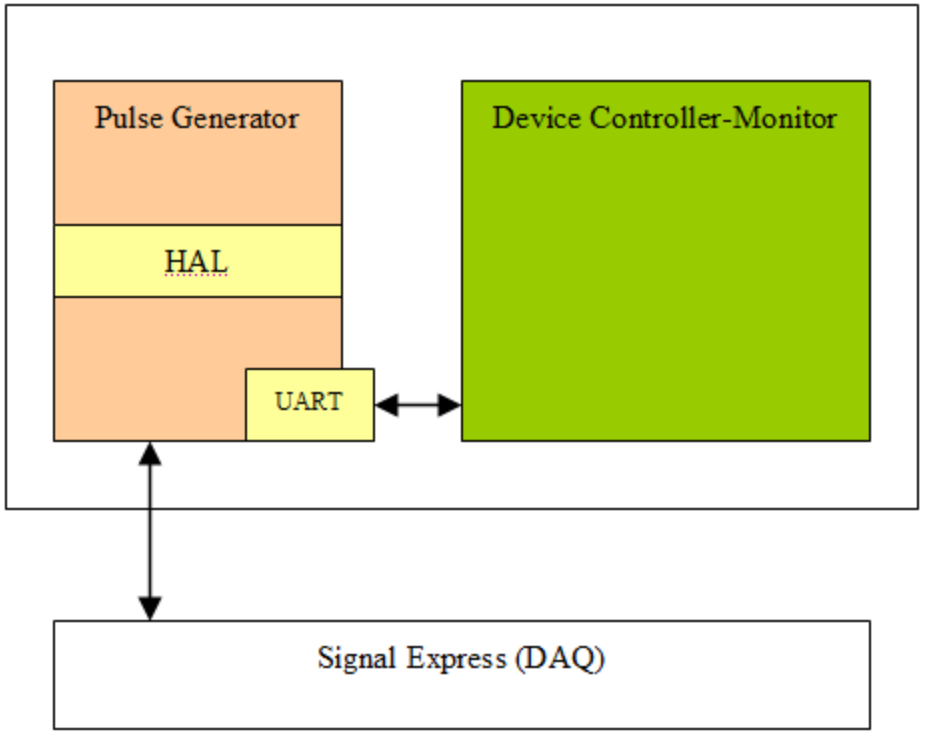
Example: Academic Pacemaker



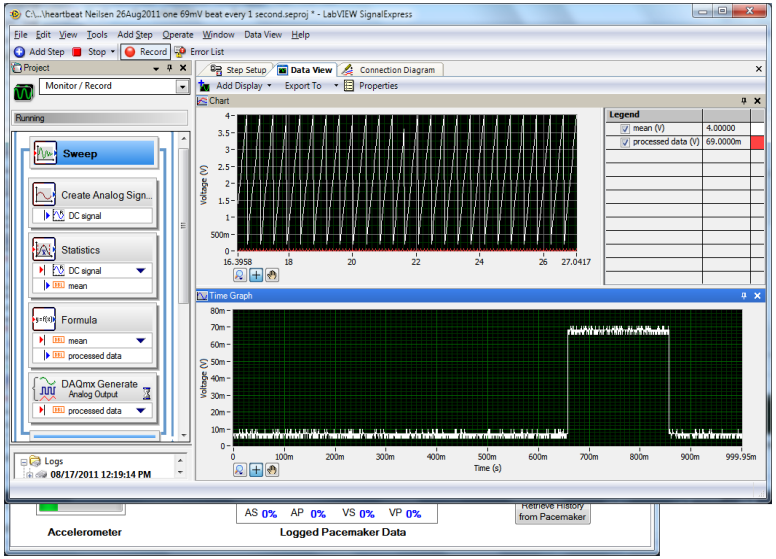
- Hardware Configuration

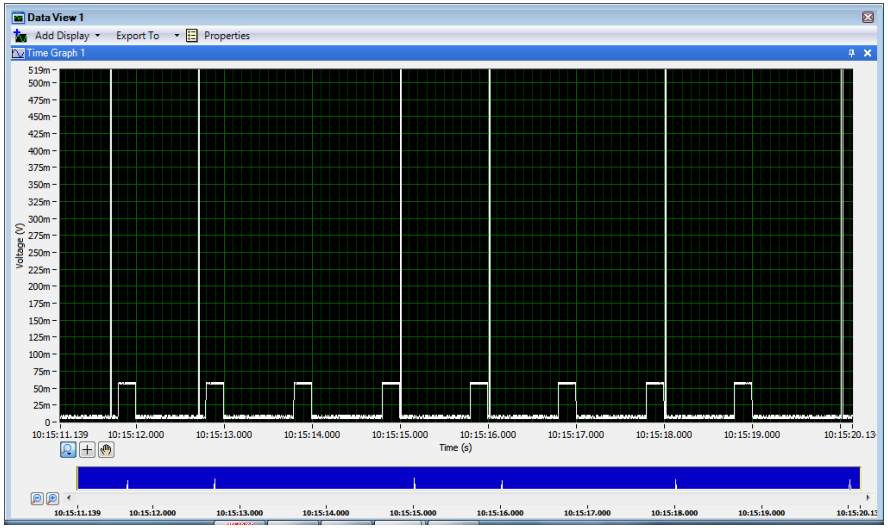
Pulse Generation/Sensing using PIC 18F4520



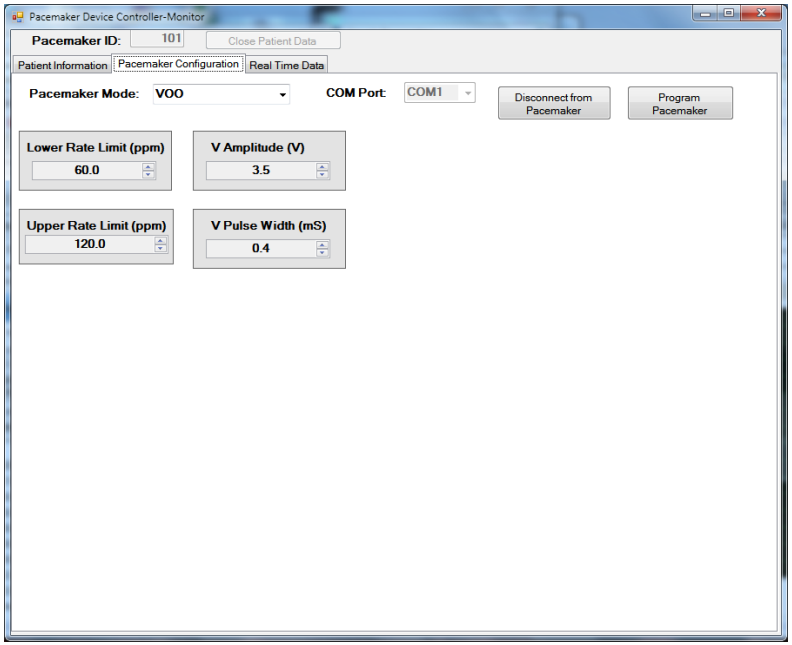


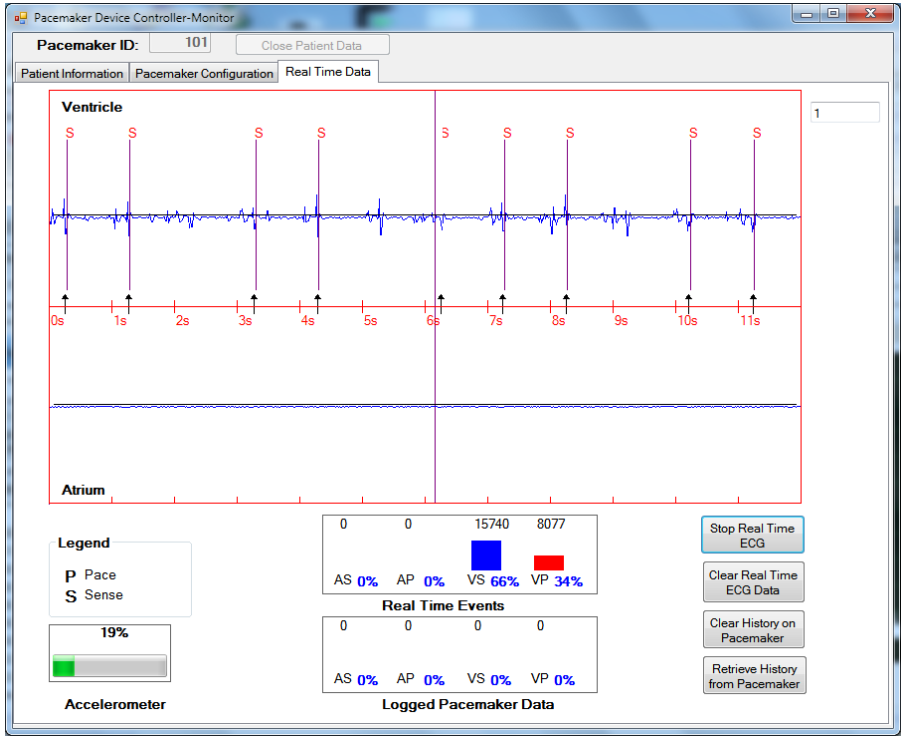
NI Signal Express - square wave pulse of 69mV or 59mV at 1Hz





Device Controller-Monitor (DCM)





Summary

- Read Ch. 8.
- Read Sha's paper on priority inheritance protocols.