**EE445M – Lab 3: Performance Measures of an RTOS**

**with Blocking and Priority**

Sourabh Shirhatti and Nelson Wu, Spring 2015

**Objectives**

The objective of this lab was to extend our RTOS from Lab 2 to include blocking and priority. We also provided support for two real-time periodic tasks and two switches. A priority scheduler and blocking semaphores will allow us to implement bounded waiting, as well as execute more time critical tasks first at any point while the OS is running. This may improve the efficiency of our robots. In addition, we developed minimally invasive tools to determine performance measures such as latency, utilization, and total time running with interrupts disabled. The debugging and performance data was recorded and downloaded to the PC.

**Hardware Design**

None

**Software Design**

long MaxJitter;

unsigned static long LastTime; // time at previous ADC sample

unsigned long thisTime; // time at current ADC sample

long jitter; // time between measured and expected, in us

unsigned long diff;

void Jitter(void) {

unsigned long myId = OS\_Id();

ST7735\_Message(1,3,"Jitter 0.1us=",MaxJitter);

ST7735\_Message(1,4,"Difference =",MaxDiff);

ST7735\_Message(1,5,"StartTime =",StartTime);

ST7735\_Message(1,6,"StopTime =",StopTime);

OS\_Kill(); // done, OS does not return from a Kill

}

void Thread7(void){ // foreground thread

OS\_Sleep(5000); // 10 seconds

Jitter(); // print jitter information

OS\_Sleep(5000); // 10 seconds

Jitter(); // print jitter information

OS\_Kill();

}

#define workA 500 // {5,50,500 us} work in Task A

#define counts1us 10 // number of OS\_Time counts per 1us

void TaskA(void){ // called every {1000, 2990us} in background

static long Count = 0;

PE1 = 0x02; // debugging profile

CountA++;

thisTime = OS\_Time(); // current time, 12.5 ns

PseudoWork(workA\*counts1us); // do work (100ns time resolution)

Count++; // calculation finished

if(Count>1){ // ignore timing of first interrupt

diff = OS\_TimeDifference(LastTime,thisTime);

StartTime = LastTime;

StopTime = thisTime;

MaxDiff = diff;

if(diff>PERIOD){

jitter = (diff-TIME\_1MS+4)/8; // in 0.1 usec

}else{

jitter = (TIME\_1MS-diff+4)/8; // in 0.1 usec

}

if(jitter > MaxJitter){

MaxJitter = jitter; // in usec

MaxDiff = diff;

StartTime = LastTime;

StopTime = thisTime;

} // jitter should be 0

if(jitter >= JitterSize){

jitter = JITTERSIZE-1;

}

JitterHistogram[jitter]++;

}

LastTime = thisTime;

PE1 = 0x00; // debugging profile

}

Sema4Type s; // test of this counting semaphore

unsigned long SignalCount1; // number of times s is signaled

unsigned long SignalCount2; // number of times s is signaled

unsigned long SignalCount3; // number of times s is signaled

unsigned long WaitCount1; // number of times s is successfully waited on

unsigned long WaitCount2; // number of times s is successfully waited on

unsigned long WaitCount3; // number of times s is successfully waited on

#define MAXCOUNT 2000 //20000

void OutputThread(void){ // foreground thread

UART\_OutString("\n\rEE345M/EE380L, Lab 3 Preparation 4\n\r");

while(SignalCount1+SignalCount2+SignalCount3<100\*MAXCOUNT){

OS\_Sleep(1000); // 1 second

UART\_OutString(".");

}

UART\_OutString(" done\n\r");

UART\_OutString("Signalled="); UART\_OutUDec(SignalCount1+SignalCount2+SignalCount3);

UART\_OutString(", Waited="); UART\_OutUDec(WaitCount1+WaitCount2+WaitCount3);

UART\_OutString("\n\r");

OS\_Kill();

}

void Wait1(void){ // foreground thread

for(;;){

OS\_Wait(&s); // three threads waiting

WaitCount1++;

}

}

void Wait2(void){ // foreground thread

for(;;){

OS\_Wait(&s); // three threads waiting

WaitCount2++;

}

}

void Wait3(void){ // foreground thread

for(;;){

OS\_Wait(&s); // three threads waiting

WaitCount3++;

}

}

void Signal1(void){ // called every 799us in background

if(SignalCount1<MAXCOUNT){

OS\_Signal(&s);

SignalCount1++;

}

}

// edit this so it changes the periodic rate

void Signal2(void){ // called every 1111us in background

if(SignalCount2<MAXCOUNT){

OS\_Signal(&s);

SignalCount2++;

}

}

void Signal3(void){ // foreground

while(SignalCount3<98\*MAXCOUNT){

OS\_Signal(&s);

SignalCount3++;

}

OS\_Kill();

}

long add(const long n, const long m){

static long result;

result = m+n;

return result;

}

int Testmain6(void){ // Testmain6 Lab 3

volatile unsigned long delay;

OS\_Init(); // initialize, disable interrupts

// SYSCTL\_RCGCGPIO\_R |= 8;

// delay = SYSCTL\_RCGCGPIO\_R;

// GPIO\_PORTD\_DIR\_R = 0;

// GPIO\_PORTD\_DEN\_R = 0xFF;

// GPIO\_PORTD\_AMSEL\_R = 0;

// GPIO\_PORTD\_AFSEL\_R = 0;

// GPIO\_PORTD\_DATA\_R = 0;

delay = add(3,4);

PortE\_Init();

SignalCount1 = 0; // number of times s is signaled

SignalCount2 = 0; // number of times s is signaled

SignalCount3 = 0; // number of times s is signaled

WaitCount1 = 0; // number of times s is successfully waited on

WaitCount2 = 0; // number of times s is successfully waited on

WaitCount3 = 0; // number of times s is successfully waited on

OS\_InitSemaphore(&s,0); // this is the test semaphore

OS\_AddPeriodicThread(&Signal1,(799\*TIME\_1MS)/1000,0); // 0.799 ms, higher priority

OS\_AddPeriodicThread(&Signal2,(1111\*TIME\_1MS)/1000,1); // 1.111 ms, lower priority

NumCreated = 0 ;

NumCreated += OS\_AddThread(&Thread6,128,5); // idle thread to keep from crashing

NumCreated += OS\_AddThread(&OutputThread,128,2); // results output thread 2

NumCreated += OS\_AddThread(&Signal3,128,2); // signalling thread

NumCreated += OS\_AddThread(&Wait1,128,2); // waiting thread

NumCreated += OS\_AddThread(&Wait2,128,2); // waiting thread

NumCreated += OS\_AddThread(&Wait3,128,2); // waiting thread

OS\_Launch(TIME\_1MS); // 1ms, doesn't return, interrupts enabled in here

return 0; // this never executes

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Modified by Sourabh Shirhatti and Nelson Wu for EE 445M, Spring 2015

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Used with Testmain2; comment out when running Testmain1

#define WITH\_SYSTICK

#define TESTMAIN5

//#define RROBIN

//#define SPINLOCK

// os.c

// Runs on LM4F120/TM4C123

// A very simple real time operating system with minimal features.

// Daniel Valvano

// January 29, 2015

/\* This example accompanies the book

"Embedded Systems: Real Time Interfacing to ARM Cortex M Microcontrollers",

ISBN: 978-1463590154, Jonathan Valvano, copyright (c) 2015

Programs 4.4 through 4.12, section 4.2

Copyright 2015 by Jonathan W. Valvano, valvano@mail.utexas.edu

You may use, edit, run or distribute this file

as long as the above copyright notice remains

THIS SOFTWARE IS PROVIDED "AS IS". NO WARRANTIES, WHETHER EXPRESS, IMPLIED

OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF

MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE.

VALVANO SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL,

OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

For more information about my classes, my research, and my books, see

http://users.ece.utexas.edu/~valvano/

\*/

#include <stdint.h>

#include "os.h"

#include "PLL.h"

#include "ST7735.h"

// Additional includes for Lab 2

#include "inc/tm4c123gh6pm.h"

#include "UART.h"

#define NVIC\_ST\_CTRL\_R (\*((volatile uint32\_t \*)0xE000E010))

#define NVIC\_ST\_CTRL\_CLK\_SRC 0x00000004 // Clock Source

#define NVIC\_ST\_CTRL\_INTEN 0x00000002 // Interrupt enable

#define NVIC\_ST\_CTRL\_ENABLE 0x00000001 // Counter mode

#define NVIC\_ST\_RELOAD\_R (\*((volatile uint32\_t \*)0xE000E014))

#define NVIC\_ST\_CURRENT\_R (\*((volatile uint32\_t \*)0xE000E018))

#define NVIC\_INT\_CTRL\_R (\*((volatile uint32\_t \*)0xE000ED04))

#define NVIC\_INT\_CTRL\_PENDSTSET 0x04000000 // Set pending SysTick interrupt

#define NVIC\_SYS\_PRI3\_R (\*((volatile uint32\_t \*)0xE000ED20)) // Sys. Handlers 12 to 15 Priority

// Additional defines for Lab 2

#define NVIC\_EN0\_INT21 0x00200000 // Interrupt 21 enable

#define NVIC\_EN1\_INT35 0x00000008

#define NVIC\_EN2\_INT70 0x00000040

#define TIMER\_CFG\_32\_BIT\_TIMER 0x00000000 // 32-bit timer configuration

#define TIMER\_TAMR\_TACDIR 0x00000010 // GPTM Timer A Count Direction

#define TIMER\_TAMR\_TAMR\_PERIOD 0x00000002 // Periodic Timer mode

#define TIMER\_CTL\_TAEN 0x00000001 // GPTM TimerA Enable

#define TIMER\_IMR\_TATOIM 0x00000001 // GPTM TimerA Time-Out Interrupt

// Mask

#define TIMER\_ICR\_TATOCINT 0x00000001 // GPTM TimerA Time-Out Raw

// Interrupt

#define TIMER\_TAILR\_M 0xFFFFFFFF // GPTM Timer A Interval Load

// Register

#define GPIO\_PORTF2 (\*((volatile uint32\_t \*)0x40025010))

// function definitions in osasm.s

void OS\_DisableInterrupts(void); // Disable interrupts

void OS\_EnableInterrupts(void); // Enable interrupts

int32\_t StartCritical(void);

void EndCritical(int32\_t primask);

void StartOS(void);

#define NUMTHREADS 50 // maximum number of threads

#define STACKSIZE 100 // number of 32-bit words in stack

#define OSFIFOSIZE 32

#define NUMPRI 6 // maximum number of priorities

struct tcb{

int32\_t \*sp; // pointer to stack (valid for threads not running

struct tcb \*next; // linked-list pointer

Sema4Type \*status; // pointer to resource thread is blocked on (0 if not)

uint32\_t sleepCt; // sleep counter in MS

uint32\_t age; // how long the thread has been active

uint32\_t id; // thread #

struct tcb \*prev; // previous thread

uint32\_t priority; // used in priority scheduling

};

tcbType tcbs[NUMTHREADS];

tcbType \*RunPt;

tcbType \*NextRunPt;

int32\_t Stacks[NUMTHREADS][STACKSIZE];

static int i;

tcbType\* RunPtArray[NUMPRI];

uint32\_t PriCount[NUMPRI];

int32\_t available[NUMTHREADS];

uint32\_t PriLevel;

uint32\_t Launched;

void InitAvailable(void) {

for (i = 0; i < NUMTHREADS; i++) {

available[i] = 1;

}

}

void InitPriorities(void) {

int j;

PriLevel = 0;

InitAvailable();

for (j = 0; j < NUMPRI; j++) {

RunPtArray[j] = 0;

PriCount[j] = 0;

}

}

int add\_thread(void) {

int ret;

for (i=0; i < NUMTHREADS; i++) {

if (available[i]) {

ret = i;

available[i] = 0;

return ret;

}

}

return -1;

}

int add\_thread\_pri(uint32\_t thread, uint32\_t priority) {

tcbType \*next, \*prev;

if(PriCount[priority] == 0) {

tcbs[thread].prev = &tcbs[thread];

tcbs[thread].next = &tcbs[thread];

RunPtArray[priority] = &tcbs[thread];

}

else {

prev = RunPtArray[priority];

next = prev->next;

tcbs[thread].prev = prev;

tcbs[thread].next = next;

prev->next = &tcbs[thread];

next->prev = &tcbs[thread];

}

PriCount[priority]++;

return 1;

}

int add\_thread\_sema4(Sema4Type \*semaPt, uint32\_t priority) {

tcbType \*priList = semaPt->TcbPri[priority];

RunPt->status = semaPt;

RunPt->next = 0;

if(priList == 0) { // first node

semaPt->TcbPri[priority] = RunPt;

RunPt->prev = 0;

}

else {

while(priList->next) {

priList = priList->next;

}

priList->next = RunPt;

RunPt->prev = priList;

}

return 1;

}

int delete\_thread\_sema4(Sema4Type \*semaPt) {

uint32\_t del\_priority = 0;

tcbType \*priNode;

for(del\_priority = 0; del\_priority < NUMPRI; del\_priority++) {

if(semaPt->TcbPri[del\_priority] != 0) { break; }

}

if(del\_priority == NUMPRI) {} // Shouldn't happen, ERROR

priNode = semaPt->TcbPri[del\_priority];

semaPt->TcbPri[del\_priority] = priNode->next;

// Don't bother replacing priNode->next->prev

return priNode->id;

}

int delete\_thread\_pri(uint32\_t priority) {

tcbType \*next, \*prev;

PriCount[priority]--;

prev = RunPt->prev;

next = RunPt->next;

if(PriCount[priority] > 0) {

prev->next = next;

next->prev = prev;

}

else {

prev = 0;

next = 0;

}

RunPtArray[priority] = prev;

return 1;

}

int delete\_thread(int thread) {

if (available[thread]) {

return -1;

// Cannot release thread which is already available

}

available[thread] = 1;

return 1;

}

int find\_prev(int thread) {

int ret;

for (i = (thread+NUMTHREADS-1)%NUMTHREADS; i != thread; i = (i+NUMTHREADS-1)%NUMTHREADS ) {

if (!available[i]) {

ret = i;

return ret;

}

}

return -1;

}

void SetInitialStack(int i){

tcbs[i].sp = &Stacks[i][STACKSIZE-16]; // thread stack pointer

Stacks[i][STACKSIZE-1] = 0x01000000; // thumb bit

Stacks[i][STACKSIZE-3] = 0x14141414; // R14

Stacks[i][STACKSIZE-4] = 0x12121212; // R12

Stacks[i][STACKSIZE-5] = 0x03030303; // R3

Stacks[i][STACKSIZE-6] = 0x02020202; // R2

Stacks[i][STACKSIZE-7] = 0x01010101; // R1

Stacks[i][STACKSIZE-8] = 0x00000000; // R0

Stacks[i][STACKSIZE-9] = 0x11111111; // R11

Stacks[i][STACKSIZE-10] = 0x10101010; // R10

Stacks[i][STACKSIZE-11] = 0x09090909; // R9

Stacks[i][STACKSIZE-12] = 0x08080808; // R8

Stacks[i][STACKSIZE-13] = 0x07070707; // R7

Stacks[i][STACKSIZE-14] = 0x06060606; // R6

Stacks[i][STACKSIZE-15] = 0x05050505; // R5

Stacks[i][STACKSIZE-16] = 0x04040404; // R4

}

//\*\*\*\*\*\*\*\* OSAddThreads \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// add three foregound threads to the scheduler

// Inputs: three pointers to a void/void foreground tasks

// Outputs: 1 if successful, 0 if this thread can not be added

int OSAddThreads(void(\*task0)(void),

void(\*task1)(void),

void(\*task2)(void)){ int32\_t status;

status = StartCritical();

tcbs[0].next = &tcbs[1]; // 0 points to 1

tcbs[1].next = &tcbs[2]; // 1 points to 2

tcbs[2].next = &tcbs[0]; // 2 points to 0

SetInitialStack(0); Stacks[0][STACKSIZE-2] = (int32\_t)(task0); // PC

SetInitialStack(1); Stacks[1][STACKSIZE-2] = (int32\_t)(task1); // PC

SetInitialStack(2); Stacks[2][STACKSIZE-2] = (int32\_t)(task2); // PC

RunPt = &tcbs[0]; // thread 0 will run first

EndCritical(status);

return 1; // successful

}

void InitTimer2A(uint32\_t period);

void InitTimer3A(uint32\_t period);

static uint32\_t MSTime;

// \*\*\*\*\*\*\*\* OS\_Init \*\*\*\*\*\*\*\*\*\*\*\*

// initialize operating system, disable interrupts until OS\_Launch

// initialize OS controlled I/O: serial, ADC, systick, LaunchPad I/O and timers

// input: none

// output: none

void OS\_Init(void) {

OS\_DisableInterrupts();

PLL\_Init(); // set processor clock to 80 MHz

#ifdef RROBIN

InitAvailable(); // all locations free

#else

InitPriorities(); // initial priority = 0

#endif

InitTimer2A(TIME\_1MS); // sleep decrementer

#ifdef TESTMAIN5

InitTimer3A(0xFFFFFFFF); // OS time timer

UART\_Init(); // initialize UART

Output\_Init(); // initialize ST7735 LCD

#endif

MSTime = 0; // current OS time = 0

Launched = 0;

NVIC\_ST\_CTRL\_R = 0; // disable SysTick during setup

NVIC\_ST\_CURRENT\_R = 0; // any write to current clears it

// lowest PRI so only foreground interrupted

NVIC\_SYS\_PRI3\_R =(NVIC\_SYS\_PRI3\_R&0x00FFFFFF)|0xE0000000; // priority 7

// Use PendSV to trigger a context switch

NVIC\_SYS\_PRI3\_R =(NVIC\_SYS\_PRI3\_R&0xFF00FFFF)|0x00D00000; // priority 6

}

// \*\*\*\*\*\*\*\* OS\_InitSemaphore \*\*\*\*\*\*\*\*\*\*\*\*

// initialize semaphore

// input: pointer to a semaphore

// output: none

void OS\_InitSemaphore(Sema4Type \*semaPt, long value) {

int i;

semaPt->Value = value; // Should be free first (>0)

for(i = 0; i < NUMPRI; i++) {

semaPt->TcbPri[i] = 0;

}

}

// \*\*\*\*\*\*\*\* OS\_Wait \*\*\*\*\*\*\*\*\*\*\*\*

// decrement semaphore

// Lab2 spinlock

// Lab3 block if less than zero

// input: pointer to a counting semaphore

// output: none

#ifdef SPINLOCK

void OS\_Wait(Sema4Type \*semaPt) {

OS\_DisableInterrupts();

while(semaPt->Value <= 0) {

OS\_EnableInterrupts();

OS\_Suspend(); // run thread switcher

OS\_DisableInterrupts();

}

semaPt->Value -= 1;

OS\_EnableInterrupts();

}

#else

void OS\_Wait(Sema4Type \*semaPt) {

int32\_t status;

status = StartCritical();

semaPt->Value -= 1;

if(semaPt->Value < 0) {

delete\_thread\_pri(RunPt->priority);

add\_thread\_sema4(semaPt, RunPt->priority);

EndCritical(status);

OS\_Suspend();

status = StartCritical();

}

EndCritical(status);

}

#endif

// \*\*\*\*\*\*\*\* OS\_Signal \*\*\*\*\*\*\*\*\*\*\*\*

// increment semaphore

// Lab2 spinlock

// Lab3 wakeup blocked thread if appropriate

// input: pointer to a counting semaphore

// output: none

#ifdef SPINLOCK

void OS\_Signal(Sema4Type \*semaPt) {

int32\_t status;

status = StartCritical();

semaPt->Value += 1;

EndCritical(status);

}

#else

void OS\_Signal(Sema4Type \*semaPt) {

int32\_t status, thread, priority;

status = StartCritical();

semaPt->Value += 1;

if(semaPt->Value <= 0) {

thread = delete\_thread\_sema4(semaPt);

priority = tcbs[thread].priority;

add\_thread\_pri(thread, priority);

if(priority < PriLevel) {

EndCritical(status);

OS\_Suspend();

status = StartCritical();

}

}

EndCritical(status);

}

#endif

// \*\*\*\*\*\*\*\* OS\_bWait \*\*\*\*\*\*\*\*\*\*\*\*

// Lab2 spinlock, set to 0

// Lab3 block if less than zero

// input: pointer to a binary semaphore

// output: none

#ifdef SPINLOCK

void OS\_bWait(Sema4Type \*semaPt) {

OS\_DisableInterrupts();

while(semaPt->Value == 0) {

OS\_EnableInterrupts();

OS\_Suspend(); // run thread switcher

OS\_DisableInterrupts();

}

semaPt->Value = 0;

OS\_EnableInterrupts();

}

#else

void OS\_bWait(Sema4Type \*semaPt) {

OS\_Wait(semaPt);

}

#endif

// \*\*\*\*\*\*\*\* OS\_bSignal \*\*\*\*\*\*\*\*\*\*\*\*

// Lab2 spinlock, set to 1

// Lab3 wakeup blocked thread if appropriate

// input: pointer to a binary semaphore

// output: none

#ifdef SPINLOCK

void OS\_bSignal(Sema4Type \*semaPt) {

int32\_t status;

status = StartCritical();

semaPt->Value = 1;

EndCritical(status);

}

#else

void OS\_bSignal(Sema4Type \*semaPt) {

OS\_Signal(semaPt);

}

#endif

//\*\*\*\*\*\*\*\* OS\_AddThread \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// add a foregound thread to the scheduler

// Inputs: pointer to a void/void foreground task

// number of bytes allocated for its stack

// priority, 0 is highest, 5 is the lowest

// Outputs: 1 if successful, 0 if this thread can not be added

// stack size must be divisable by 8 (aligned to double word boundary)

// In Lab 2, you can ignore both the stackSize and priority fields

// In Lab 3, you can ignore the stackSize fields

static uint32\_t NumThreads = 0;

#ifdef RROBIN

int OS\_AddThread(void(\*task)(void),

unsigned long stackSize, unsigned long priority) {

int32\_t status, thread, prev;

status = StartCritical();

thread = add\_thread();

if(NumThreads == 0) {

tcbs[0].next = &tcbs[0]; // 0 points to 0

RunPt = &tcbs[0]; // thread 0 will run first

}

else {

prev = find\_prev(thread);

tcbs[thread].next = tcbs[prev].next;

tcbs[prev].next = &tcbs[thread];

}

tcbs[thread].status = 0;

tcbs[thread].sleepCt = 0;

tcbs[thread].age = 0;

tcbs[thread].id = thread;

tcbs[thread].priority = priority;

SetInitialStack(thread);

Stacks[thread][STACKSIZE-2] = (int32\_t)(task); // PC

NumThreads++;

EndCritical(status);

return 1; // successful

}

#else

int OS\_AddThread(void(\*task)(void),

unsigned long stackSize, unsigned long priority) {

int32\_t status, thread, error;

status = StartCritical();

thread = add\_thread();

error = add\_thread\_pri(thread, priority);

if(NumThreads == 0) {

RunPt = &tcbs[thread]; // thread 0 will run first

PriLevel = priority;

}

tcbs[thread].status = 0;

tcbs[thread].sleepCt = 0;

tcbs[thread].age = 0;

tcbs[thread].id = thread;

tcbs[thread].priority = priority;

SetInitialStack(thread);

Stacks[thread][STACKSIZE-2] = (int32\_t)(task); // PC

NumThreads++;

EndCritical(status);

return 1; // successful

}

#endif

//\*\*\*\*\*\*\*\* OS\_Id \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// returns the thread ID for the currently running thread

// Inputs: none

// Outputs: Thread ID, number greater than zero

unsigned long OS\_Id(void) {

return RunPt->id;

}

void InitTimer3A(uint32\_t period) {

long sr;

volatile unsigned long delay;

sr = StartCritical();

SYSCTL\_RCGCTIMER\_R |= 0x08;

delay = SYSCTL\_RCGCTIMER\_R;

delay = SYSCTL\_RCGCTIMER\_R;

TIMER3\_CTL\_R &= ~TIMER\_CTL\_TAEN; // 1) disable timer1A during setup

// 2) configure for 32-bit timer mode

TIMER3\_CFG\_R = TIMER\_CFG\_32\_BIT\_TIMER;

// 3) configure for periodic mode, default down-count settings

TIMER3\_TAMR\_R = TIMER\_TAMR\_TAMR\_PERIOD;

TIMER3\_TAILR\_R = 0xFFFFFFFF - 1; // 4) reload value

// 5) clear timer1A timeout flag

TIMER3\_ICR\_R = TIMER\_ICR\_TATOCINT;

TIMER3\_IMR\_R |= TIMER\_IMR\_TATOIM;// 6) arm timeout interrupt

// 7) priority shifted to bits 15-13 for timer1A

NVIC\_PRI8\_R = (NVIC\_PRI8\_R&0x00FFFFFF)|(1 << 29); //3

NVIC\_EN1\_R = NVIC\_EN1\_INT35; // 8) enable interrupt 21 in NVIC

TIMER3\_TAPR\_R = 0;

TIMER3\_CTL\_R |= TIMER\_CTL\_TAEN; // 9) enable timer1A

EndCritical(sr);

}

void Timer3A\_Handler(void){

TIMER3\_ICR\_R = TIMER\_ICR\_TATOCINT;// acknowledge timer1A timeout

// SystemTime = SystemTime + 1;

GPIO\_PORTE\_DATA\_R ^= 0x02;

}

//struct PeriodicTcb {

// void (\*PeriodicTask)(void);

// uint32\_t priority;

// uint32\_t period;

//};

//typedef struct PeriodicTcb PeriodicTcbType;

//PeriodicTcbType PeriodicThreads[5];

//static uint32\_t MinPeriod = 0;

//static uint32\_t NumPeriodicTasks = 0;

void (\*PeriodicTask)(void);

void (\*PeriodicTask2)(void);

void InitTimer1A(uint32\_t period, uint32\_t priority) {

long sr;

volatile unsigned long delay;

sr = StartCritical();

SYSCTL\_RCGCTIMER\_R |= 0x02;

delay = SYSCTL\_RCGCTIMER\_R;

delay = SYSCTL\_RCGCTIMER\_R;

TIMER1\_CTL\_R &= ~TIMER\_CTL\_TAEN; // 1) disable timer1A during setup

// 2) configure for 32-bit timer mode

TIMER1\_CFG\_R = TIMER\_CFG\_32\_BIT\_TIMER;

// 3) configure for periodic mode, default down-count settings

TIMER1\_TAMR\_R = TIMER\_TAMR\_TAMR\_PERIOD;

TIMER1\_TAILR\_R = period - 1; // 4) reload value

// 5) clear timer1A timeout flag

TIMER1\_ICR\_R = TIMER\_ICR\_TATOCINT;

TIMER1\_IMR\_R |= TIMER\_IMR\_TATOIM;// 6) arm timeout interrupt

// 7) priority shifted to bits 15-13 for timer1A

NVIC\_PRI5\_R = (NVIC\_PRI5\_R&0xFFFF00FF)|((priority+3) << 13); //3

NVIC\_EN0\_R = NVIC\_EN0\_INT21; // 8) enable interrupt 21 in NVIC

TIMER1\_TAPR\_R = 0;

TIMER1\_CTL\_R |= TIMER\_CTL\_TAEN; // 9) enable timer1A

EndCritical(sr);

}

void Timer1A\_Handler(void){

TIMER1\_ICR\_R = TIMER\_ICR\_TATOCINT;// acknowledge timer1A timeout

(\*PeriodicTask)();

// for(i = 0; i < NumPeriodicTasks; i++) {

// if(

// }

}

void InitTimer4A(uint32\_t period, uint32\_t priority) {

long sr;

volatile unsigned long delay;

sr = StartCritical();

SYSCTL\_RCGCTIMER\_R |= 0x10;

delay = SYSCTL\_RCGCTIMER\_R;

delay = SYSCTL\_RCGCTIMER\_R;

TIMER4\_CTL\_R &= ~TIMER\_CTL\_TAEN; // 1) disable timer1A during setup

// 2) configure for 32-bit timer mode

TIMER4\_CFG\_R = TIMER\_CFG\_32\_BIT\_TIMER;

// 3) configure for periodic mode, default down-count settings

TIMER4\_TAMR\_R = TIMER\_TAMR\_TAMR\_PERIOD;

TIMER4\_TAILR\_R = period - 1; // 4) reload value

// 5) clear timer1A timeout flag

TIMER4\_ICR\_R = TIMER\_ICR\_TATOCINT;

TIMER4\_IMR\_R |= TIMER\_IMR\_TATOIM;// 6) arm timeout interrupt

// 7) priority shifted to bits 15-13 for timer1A

NVIC\_PRI17\_R = (NVIC\_PRI17\_R&0xFF00FFFF)|((priority+3) << 21); //3

NVIC\_EN2\_R = NVIC\_EN2\_INT70; // 8) enable interrupt 21 in NVIC

TIMER4\_TAPR\_R = 0;

TIMER4\_CTL\_R |= TIMER\_CTL\_TAEN; // 9) enable timer1A

EndCritical(sr);

}

void Timer4A\_Handler(void){

TIMER4\_ICR\_R = TIMER\_ICR\_TATOCINT;// acknowledge timer1A timeout

(\*PeriodicTask2)();

}

//\*\*\*\*\*\*\*\* OS\_AddPeriodicThread \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// add a background periodic task

// typically this function receives the highest priority

// Inputs: pointer to a void/void background function

// period given in system time units (12.5ns)

// priority 0 is the highest, 5 is the lowest

// Outputs: 1 if successful, 0 if this thread can not be added

// You are free to select the time resolution for this function

// It is assumed that the user task will run to completion and return

// This task can not spin, block, loop, sleep, or kill

// This task can call OS\_Signal OS\_bSignal OS\_AddThread

// This task does not have a Thread ID

// In lab 2, this command will be called 0 or 1 times

// In lab 2, the priority field can be ignored

// In lab 3, this command will be called 0 1 or 2 times

// In lab 3, there will be up to four background threads, and this priority field

// determines the relative priority of these four threads

int OS\_AddPeriodicThread(void(\*task)(void),

unsigned long period, unsigned long priority) {

// PeriodicThreads[NumPeriodicTasks].PeriodicTask = task;

// PeriodicThreads[NumPeriodicTasks].priority = priority;

// PeriodicThreads[NumPeriodicTasks].period = period;

// if(period < MinPeriod) {

// InitTimer1A(period);

// }

static uint32\_t NumPeriodicTasks = 0;

if(NumPeriodicTasks == 0) {

PeriodicTask = task;

InitTimer1A(period, priority);

}

else {

PeriodicTask2 = task;

InitTimer4A(period, priority);

}

NumPeriodicTasks++;

return 1;

}

uint32\_t LastPF4, LastPF0;

void (\*SWOneTask)(void);

void SWOneInit(uint32\_t priority){

unsigned long volatile delay;

SYSCTL\_RCGCGPIO\_R |= 0x00000020; // (a) activate clock for port F

delay = SYSCTL\_RCGCGPIO\_R;

GPIO\_PORTF\_CR\_R = 0x10; // allow changes to PF4,0

GPIO\_PORTF\_DIR\_R &= ~0x10; // (c) make PF4,0 in (built-in button)

GPIO\_PORTF\_AFSEL\_R &= ~0x10; // disable alt funct on PF4,0

GPIO\_PORTF\_DEN\_R |= 0x10; // enable digital I/O on PF4,0

GPIO\_PORTF\_PCTL\_R &= ~0; // PF4,PF0 is not both edges

GPIO\_PORTF\_ICR\_R = 0x10; // configure PF4,0 as GPIO

GPIO\_PORTF\_AMSEL\_R &= ~0x10; // disable analog functionality on PF4,0

GPIO\_PORTF\_PUR\_R |= 0x10; // enable weak pull-up on PF4,0

GPIO\_PORTF\_IS\_R &= ~0x10; // (d) PF4,PF0 is edge-sensitive

GPIO\_PORTF\_IBE\_R |= 0x10; //

GPIO\_PORTF\_ICR\_R |= 0x10; // (e) clear flags 4,0

GPIO\_PORTF\_IM\_R |= 0x10; // (f) arm interrupt on PF4,PF0

LastPF4 = GPIO\_PORTF\_DATA\_R & 0x10;

NVIC\_PRI7\_R = (NVIC\_PRI7\_R&0xFF00FFFF)|((priority+2) << 21); // (g) priority 2

NVIC\_EN0\_R = 0x40000000; // (h) enable interrupt 30 in NVIC

}

void (\*SWTwoTask)(void);

void SWTwoInit(uint32\_t priority){

unsigned long volatile delay;

SYSCTL\_RCGCGPIO\_R |= 0x00000020; // (a) activate clock for port F

delay = SYSCTL\_RCGCGPIO\_R;

GPIO\_PORTF\_LOCK\_R = 0x4C4F434B; // unlock GPIO Port F

GPIO\_PORTF\_CR\_R = 0x01; // allow changes to PF4,0

GPIO\_PORTF\_DIR\_R &= ~0x01; // (c) make PF4,0 in (built-in button)

GPIO\_PORTF\_AFSEL\_R &= ~0x01; // disable alt funct on PF4,0

GPIO\_PORTF\_DEN\_R |= 0x01; // enable digital I/O on PF4,0

GPIO\_PORTF\_PCTL\_R &= ~0x000F000F; // configure PF4,0 as GPIO

GPIO\_PORTF\_AMSEL\_R &= ~0x01; // disable analog functionality on PF4,0

GPIO\_PORTF\_PUR\_R |= 0x01; // enable weak pull-up on PF4,0

GPIO\_PORTF\_IS\_R &= ~0x01; // (d) PF4,PF0 is edge-sensitive

GPIO\_PORTF\_IBE\_R |= 0x01; // PF4,PF0 is both edges

GPIO\_PORTF\_ICR\_R |= 0x01; // (e) clear flags 4,0

GPIO\_PORTF\_IM\_R |= 0x01; // (f) arm interrupt on PF4,PF0

LastPF0 = GPIO\_PORTF\_DATA\_R & 0x01;

NVIC\_PRI7\_R = (NVIC\_PRI7\_R&0xFF00FFFF)|((priority+2) << 21); // (g) priority 5 0x00400000

NVIC\_EN0\_R = 0x40000000; // (h) enable interrupt 30 in NVIC

}

void static DebouncePF4(void) {

OS\_Sleep(2); //foreground sleep, must run within 5ms

LastPF4 = GPIO\_PORTF\_DATA\_R & 0x10;

GPIO\_PORTF\_ICR\_R = 0x10;

GPIO\_PORTF\_IM\_R |= 0x10;

OS\_Kill();

}

void static DebouncePF0(void) {

OS\_Sleep(2); //foreground sleep, must run within 5ms

LastPF0 = GPIO\_PORTF\_DATA\_R & 0x01;

GPIO\_PORTF\_ICR\_R = 0x01;

GPIO\_PORTF\_IM\_R |= 0x01;

OS\_Kill();

}

void GPIOPortF\_Handler(void) { // called on touch of either SW1 or SW2

if(GPIO\_PORTF\_RIS\_R&0x01) { // SW2 touch

if (LastPF0) { (\*SWTwoTask)(); }

GPIO\_PORTF\_IM\_R &= ~0x01;

OS\_AddThread(&DebouncePF0, 128, 2);

}

if(GPIO\_PORTF\_RIS\_R&0x10) { // SW1 touch

if (LastPF4) { (\*SWOneTask)(); }

GPIO\_PORTF\_IM\_R &= ~0x10;

OS\_AddThread(&DebouncePF4, 128 ,2);

}

}

//\*\*\*\*\*\*\*\* OS\_AddSW1Task \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// add a background task to run whenever the SW1 (PF4) button is pushed

// Inputs: pointer to a void/void background function

// priority 0 is the highest, 5 is the lowest

// Outputs: 1 if successful, 0 if this thread can not be added

// It is assumed that the user task will run to completion and return

// This task can not spin, block, loop, sleep, or kill

// This task can call OS\_Signal OS\_bSignal OS\_AddThread

// This task does not have a Thread ID

// In labs 2 and 3, this command will be called 0 or 1 times

// In lab 2, the priority field can be ignored

// In lab 3, there will be up to four background threads, and this priority field

// determines the relative priority of these four threads

int OS\_AddSW1Task(void(\*task)(void), unsigned long priority) {

SWOneTask = task;

SWOneInit(priority);

return 1;

}

//\*\*\*\*\*\*\*\* OS\_AddSW2Task \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// add a background task to run whenever the SW2 (PF0) button is pushed

// Inputs: pointer to a void/void background function

// priority 0 is highest, 5 is lowest

// Outputs: 1 if successful, 0 if this thread can not be added

// It is assumed user task will run to completion and return

// This task can not spin block loop sleep or kill

// This task can call issue OS\_Signal, it can call OS\_AddThread

// This task does not have a Thread ID

// In lab 2, this function can be ignored

// In lab 3, this command will be called will be called 0 or 1 times

// In lab 3, there will be up to four background threads, and this priority field

// determines the relative priority of these four threads

int OS\_AddSW2Task(void(\*task)(void), unsigned long priority) {

SWTwoTask = task;

SWTwoInit(priority);

return 1;

}

// \*\*\*\*\*\*\*\* OS\_Sleep \*\*\*\*\*\*\*\*\*\*\*\*

// place this thread into a dormant state

// input: number of msec to sleep

// output: none

// You are free to select the time resolution for this function

// OS\_Sleep(0) implements cooperative multitasking

void OS\_Sleep(unsigned long sleepTime) {

RunPt->sleepCt = sleepTime;

OS\_Suspend(); // May need to change

}

// \*\*\*\*\*\*\*\* OS\_Kill \*\*\*\*\*\*\*\*\*\*\*\*

// kill the currently running thread, release its TCB and stack

// input: none

// output: none

#ifdef RROBIN

void OS\_Kill(void) {

uint32\_t thread, prev;

thread = RunPt->id;

prev = find\_prev(thread);

delete\_thread(thread);

NumThreads--;

tcbs[prev].next = tcbs[thread].next;

OS\_Suspend();

}

#else

void OS\_Kill(void) {

uint32\_t thread, priority;

GPIO\_PORTD\_DATA\_R ^= 0x01;

thread = RunPt->id;

priority = RunPt->priority;

delete\_thread\_pri(priority);

delete\_thread(thread);

NumThreads--;

GPIO\_PORTD\_DATA\_R ^= 0x01;

OS\_Suspend();

}

#endif

// \*\*\*\*\*\*\*\* OS\_Suspend \*\*\*\*\*\*\*\*\*\*\*\*

// suspend execution of currently running thread

// scheduler will choose another thread to execute

// Can be used to implement cooperative multitasking

// Same function as OS\_Sleep(0)

// input: none

// output: none

void OS\_Suspend(void) {

GPIO\_PORTD\_DATA\_R ^= 0x02;

#ifdef WITH\_SYSTICK

NVIC\_ST\_CURRENT\_R = 0; // clear counter

GPIO\_PORTD\_DATA\_R ^= 0x02;

NVIC\_INT\_CTRL\_R = 0x04000000; // trigger SysTick

#else

NextRunPt = RunPt->next;

GPIO\_PORTD\_DATA\_R ^= 0x02;

NVIC\_INT\_CTRL\_R = 0x10000000; // trigger PendSV

#endif

}

uint16\_t static OS\_Fifo [OSFIFOSIZE];

uint16\_t \*PutPt, \*GetPt;

Sema4Type FifoAvailable;

//Sema4Type DataRoomLeft;

// \*\*\*\*\*\*\*\* OS\_Fifo\_Init \*\*\*\*\*\*\*\*\*\*\*\*

// Initialize the Fifo to be empty

// Inputs: size

// Outputs: none

// In Lab 2, you can ignore the size field

// In Lab 3, you should implement the user-defined fifo size

// In Lab 3, you can put whatever restrictions you want on size

// e.g., 4 to 64 elements

// e.g., must be a power of 2,4,8,16,32,64,128

void OS\_Fifo\_Init(unsigned long size) {

long sr;

sr = StartCritical();

OS\_InitSemaphore(&FifoAvailable, 0);

//OS\_InitSemaphore(&DataRoomLeft, OSFIFOSIZE);

PutPt = GetPt = &OS\_Fifo[0];

EndCritical(sr);

}

// \*\*\*\*\*\*\*\* OS\_Fifo\_Put \*\*\*\*\*\*\*\*\*\*\*\*

// Enter one data sample into the Fifo

// Called from the background, so no waiting

// Inputs: data

// Outputs: true if data is properly saved,

// false if data not saved, because it was full

// Since this is called by interrupt handlers

// this function can not disable or enable interrupts

int OS\_Fifo\_Put(unsigned long data) {

uint16\_t volatile \*nextPutPt;

nextPutPt = PutPt + 1;

if(nextPutPt == &OS\_Fifo[OSFIFOSIZE]){

nextPutPt = &OS\_Fifo[0];

}

if(nextPutPt == GetPt ){

return(0);

}

else{

\*( PutPt ) = data;

PutPt = nextPutPt;

OS\_Signal(&FifoAvailable);

return(1);

}

}

// \*\*\*\*\*\*\*\* OS\_Fifo\_Get \*\*\*\*\*\*\*\*\*\*\*\*

// Remove one data sample from the Fifo

// Called in foreground, will spin/block if empty

// Inputs: none

// Outputs: data

unsigned long OS\_Fifo\_Get(void) {

uint16\_t data;

// ERROR CHECKING

OS\_Wait(&FifoAvailable);

if( PutPt == GetPt ){

return(0);

}

data = \*( GetPt++);

if( GetPt == &OS\_Fifo[OSFIFOSIZE]){

GetPt = &OS\_Fifo[0];

}

return(data);

}

// \*\*\*\*\*\*\*\* OS\_Fifo\_Size \*\*\*\*\*\*\*\*\*\*\*\*

// Check the status of the Fifo

// Inputs: none

// Outputs: returns the number of elements in the Fifo

// greater than zero if a call to OS\_Fifo\_Get will return right away

// zero or less than zero if the Fifo is empty

// zero or less than zero if a call to OS\_Fifo\_Get will spin or block

long OS\_Fifo\_Size(void) {

if( PutPt < GetPt ){

return ((unsigned short)( PutPt - GetPt + (OSFIFOSIZE\*sizeof(uint16\_t)))/sizeof(uint16\_t));

}

return ((unsigned short)( PutPt - GetPt )/sizeof(uint16\_t));

}

/\*FOLLOW GUIDELINES IN LAB MANUAL\*/

static Sema4Type DataValid;

static Sema4Type BoxFree;

static uint32\_t MailBox;

// \*\*\*\*\*\*\*\* OS\_MailBox\_Init \*\*\*\*\*\*\*\*\*\*\*\*

// Initialize communication channel

// Inputs: none

// Outputs: none

void OS\_MailBox\_Init(void) {

OS\_InitSemaphore(&DataValid, 0);

OS\_InitSemaphore(&BoxFree, 1);

}

// \*\*\*\*\*\*\*\* OS\_MailBox\_Send \*\*\*\*\*\*\*\*\*\*\*\*

// enter mail into the MailBox

// Inputs: data to be sent

// Outputs: none

// This function will be called from a foreground thread

// It will spin/block if the MailBox contains data not yet received

void OS\_MailBox\_Send(unsigned long data) {

OS\_Wait(&BoxFree);

MailBox = data;

OS\_Signal(&DataValid);

}

// \*\*\*\*\*\*\*\* OS\_MailBox\_Recv \*\*\*\*\*\*\*\*\*\*\*\*

// remove mail from the MailBox

// Inputs: none

// Outputs: data received

// This function will be called from a foreground thread

// It will spin/block if the MailBox is empty

unsigned long OS\_MailBox\_Recv(void) {

uint32\_t retVal;

OS\_Wait(&DataValid);

retVal = MailBox;

OS\_Signal(&BoxFree);

return retVal;

}

// \*\*\*\*\*\*\*\* OS\_Time \*\*\*\*\*\*\*\*\*\*\*\*

// return the system time

// Inputs: none

// Outputs: time in 12.5ns units, 0 to 4294967295

// The time resolution should be less than or equal to 1us, and the precision 32 bits

// It is ok to change the resolution and precision of this function as long as

// this function and OS\_TimeDifference have the same resolution and precision

unsigned long OS\_Time(void) {

return TIMER3\_TAILR\_R - TIMER3\_TAV\_R;

}

// \*\*\*\*\*\*\*\* OS\_TimeDifference \*\*\*\*\*\*\*\*\*\*\*\*

// Calculates difference between two times

// Inputs: two times measured with OS\_Time

// Outputs: time difference in 12.5ns units

// The time resolution should be less than or equal to 1us, and the precision at least 12 bits

// It is ok to change the resolution and precision of this function as long as

// this function and OS\_Time have the same resolution and precision

unsigned long OS\_TimeDifference(unsigned long start, unsigned long stop) {

unsigned long difference;

if (stop < start) {

difference = 0xFFFFFFFF - start + stop;

}

else {

difference = stop-start;

}

return difference;

}

// \*\*\*\*\*\*\*\* OS\_ClearMsTime \*\*\*\*\*\*\*\*\*\*\*\*

// sets the system time to zero (from Lab 1)

// Inputs: none

// Outputs: none

// You are free to change how this works

void OS\_ClearMsTime(void) {

MSTime = 0;

}

// \*\*\*\*\*\*\*\* OS\_MsTime \*\*\*\*\*\*\*\*\*\*\*\*

// reads the current time in msec (from Lab 1)

// Inputs: none

// Outputs: time in ms units

// You are free to select the time resolution for this function

// It is ok to make the resolution to match the first call to OS\_AddPeriodicThread

unsigned long OS\_MsTime(void) {

//uint32\_t retVal = (OS\_Time()-MSTime)/TIME\_1MS;

return MSTime;

}

void InitTimer2A(uint32\_t period) {

long sr;

volatile unsigned long delay;

sr = StartCritical();

SYSCTL\_RCGCTIMER\_R |= 0x04;

delay = SYSCTL\_RCGCTIMER\_R;

delay = SYSCTL\_RCGCTIMER\_R;

TIMER2\_CTL\_R &= ~TIMER\_CTL\_TAEN; // 1) disable timer1A during setup

// 2) configure for 32-bit timer mode

TIMER2\_CFG\_R = TIMER\_CFG\_32\_BIT\_TIMER;

// 3) configure for periodic mode, default down-count settings

TIMER2\_TAMR\_R = TIMER\_TAMR\_TAMR\_PERIOD;

TIMER2\_TAILR\_R = period - 1; // 4) reload value

// 5) clear timer1A timeout flag

TIMER2\_ICR\_R = TIMER\_ICR\_TATOCINT;

TIMER2\_IMR\_R |= TIMER\_IMR\_TATOIM;// 6) arm timeout interrupt

// 7) priority shifted to bits 31-29 for timer2A

NVIC\_PRI5\_R = (NVIC\_PRI5\_R&0x00FFFFFF)|(1 << 29);

NVIC\_EN0\_R = NVIC\_EN0\_INT23; // 8) enable interrupt 23 in NVIC

TIMER2\_TAPR\_R = 0;

TIMER2\_CTL\_R |= TIMER\_CTL\_TAEN; // 9) enable timer2A

EndCritical(sr);

}

void Timer2A\_Handler(void){

int j;

TIMER2\_ICR\_R = TIMER\_ICR\_TATOCINT;// acknowledge timer2A timeout

MSTime++;

for(j = 0; j < NUMTHREADS; j++) {

if(!available[j]) {

tcbs[j].age += 1;

if(tcbs[j].sleepCt) {

tcbs[j].sleepCt -= 1;

}

}

}

}

//\*\*\*\*\*\*\*\* OS\_Launch \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// start the scheduler, enable interrupts

// Inputs: number of 12.5ns clock cycles for each time slice

// you may select the units of this parameter

// Outputs: none (does not return)

// In Lab 2, you can ignore the theTimeSlice field

// In Lab 3, you should implement the user-defined TimeSlice field

// It is ok to limit the range of theTimeSlice to match the 24-bit SysTick

void OS\_Launch(unsigned long theTimeSlice) {

#ifdef WITH\_SYSTICK

NVIC\_ST\_RELOAD\_R = theTimeSlice - 1; // reload value

NVIC\_ST\_CTRL\_R = 0x00000007; // enable, core clock and interrupt arm

#endif

Launched = 1;

StartOS(); // start on the first task

}

#ifdef RROBIN

void SysTick\_Handler(void) {

NextRunPt = RunPt->next;

while(NextRunPt->sleepCt) {

NextRunPt = NextRunPt->next;

}

NVIC\_INT\_CTRL\_R = 0x10000000; // trigger PendSV

}

#else

static int pri;

void SysTick\_Handler(void) {

GPIO\_PORTD\_DATA\_R ^= 0x08;

if(PriCount[PriLevel] > 0) {

RunPtArray[PriLevel] = RunPtArray[PriLevel]->next;

}

pri = 0;

while((pri < NUMPRI) && (PriCount[pri] == 0)) {

pri++;

}

if(pri == NUMPRI) {}// ERROR, NO ACTIVE THREADS, ADD IDLE TASK

PriLevel = pri;

while(PriLevel < NUMPRI) {

for(pri = 0; (pri < PriCount[PriLevel]) && (RunPtArray[PriLevel]->sleepCt); pri++) {

RunPtArray[PriLevel] = RunPtArray[PriLevel]->next;

}

if(pri == PriCount[PriLevel]) {

PriLevel++;

}

else { break; }

}

if(PriLevel == NUMPRI) {} // Error

NextRunPt = RunPtArray[PriLevel];

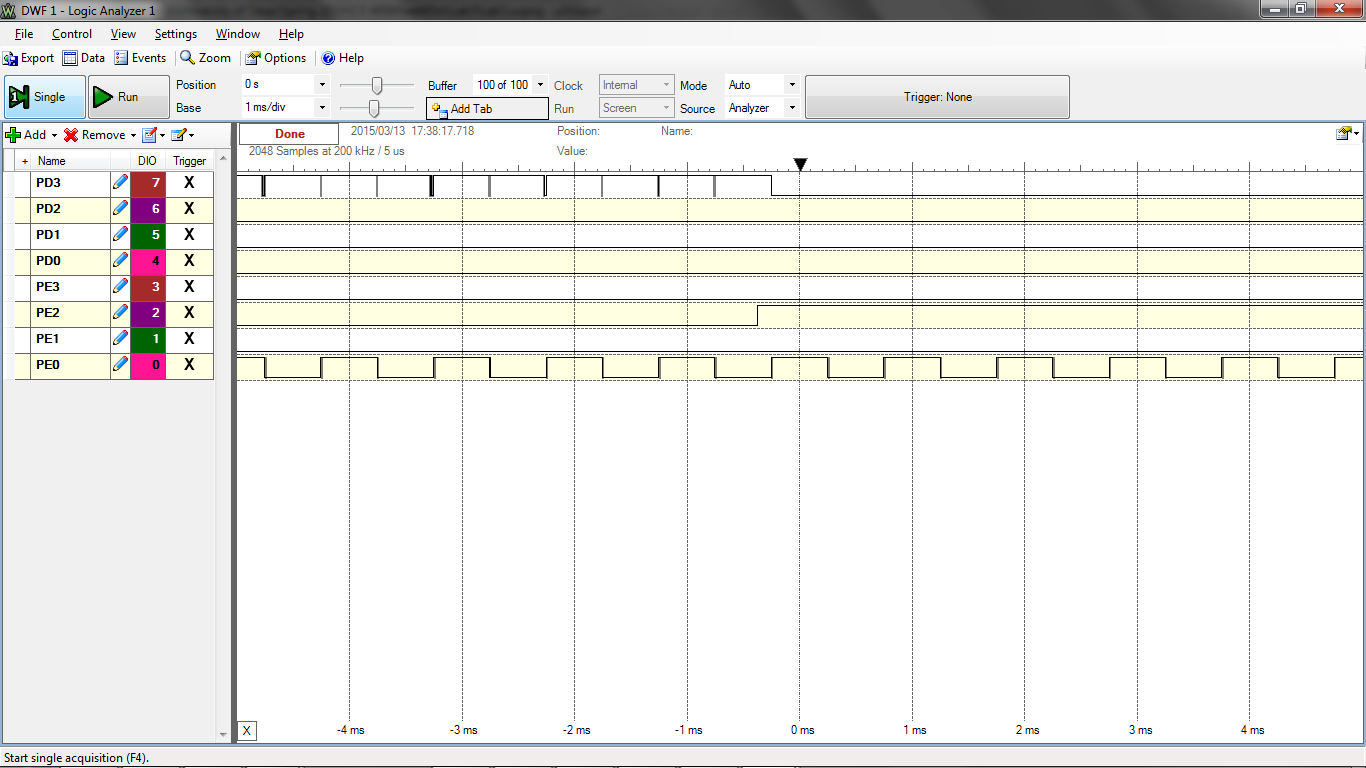
GPIO\_PORTD\_DATA\_R ^= 0x08;

NVIC\_INT\_CTRL\_R = 0x10000000; // trigger PendSV

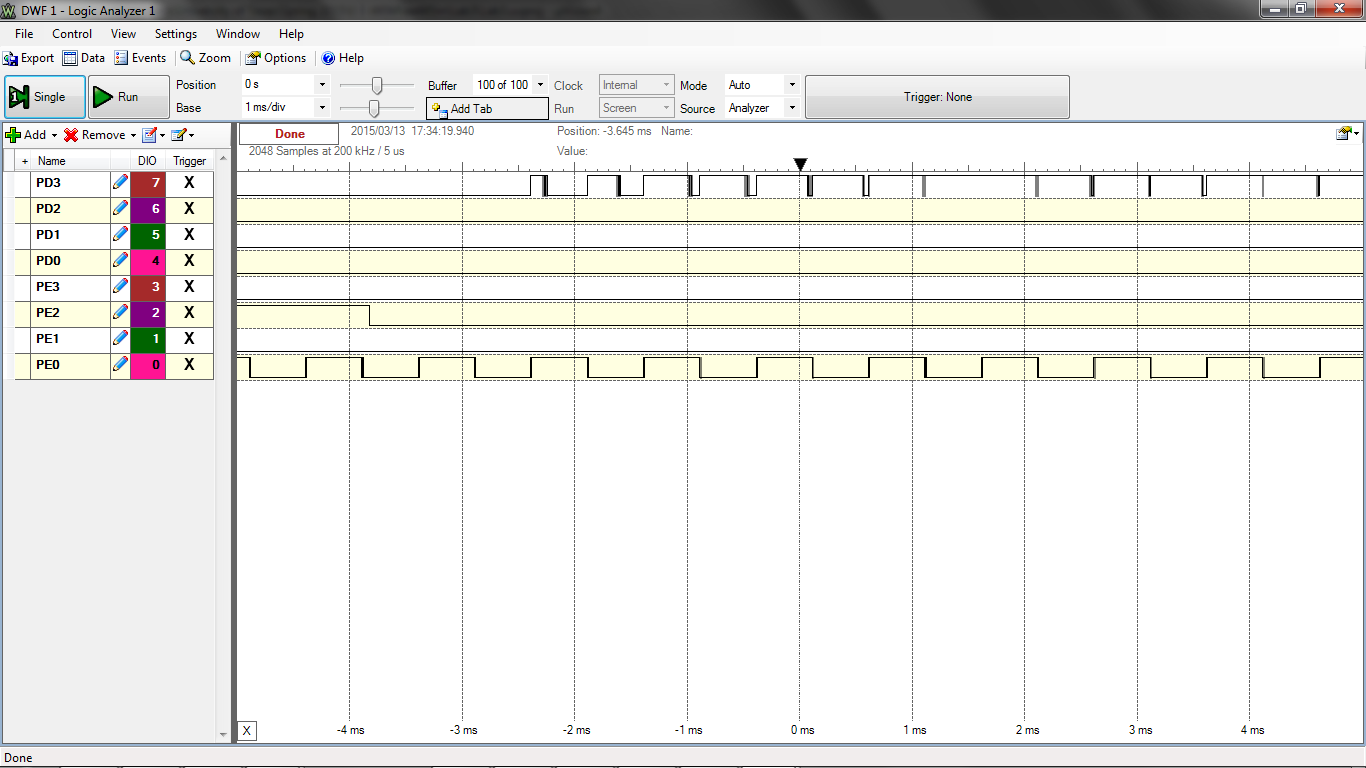
}

#endif

**Measurement Data**



**Figure 1. Blocking/Sleeping/Killing/Round Robin System**



**Figure 2. Blocking/Sleeping/Killing/Priority System**

**Table 1. Comparison of Performance Measurements vs. FIFO size and Timeslice**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **Spinlock/Round Robin** | | | **Block/Round Robin** | | | **Blocking/Priority** | | |
| **FIFO Size** | **Tslice (ms)** | **Data Lost** | **Jitter (µs)** | **PID Work** | **Data Lost** | **Jitter (µs)** | **PID Work** | **Data Lost** | **Jitter (µs)** | **PID Work** |
| 4 | 2 | 0 | 0.2 | 12124 | 0 | 0.2 | 14143 | 2 | 0 | 11943 |
| 32 | 2 | 0 | 0.2 | 12124 | 0 | 0.2 | 14527 | 0 | 0.2 | 11947 |
| 32 | 1 | 0 | 0.2 | 12086 | 0 | 0.2 | 14325 | 0 | 0 | 11916 |
| 32 | 10 | 0 | 0.2 | 12154 | 0 | 0.2 | 14325 | 0 | 0 | 11972 |

**Analysis and Discussion**

1. To support two periodic background threads, we added a second timer to run the second periodic task. This would not work for the case of 10 periodic threads because we don’t have 10 different priorities and 10 different timers to dedicate to these tasks. So, we could have one timer counting at a fast rate (< 1µs) and counters for each thread. When the count reaches the thread’s period, it will execute the periodic task and reset the counter.
2. We implemented blocking semaphores by creating separate linked lists for each semaphore. When a thread needs to wait, it is moved from the list of active threads to the semaphore’s linked list. After the semaphore is signaled, it wakes the highest priority thread on a FIFO basis. An alternative method is to have a status field in each TCB and only run those threads with a status of zero. Otherwise, the status is set to the blocking semaphore. This is inefficient for 100 threads because the list must be searched for every context switch, increasing the time of the interrupt. Our method is also somewhat inefficient because we remove the node from the list of active threads and add it to the semaphore’s linked list, changing all the next and previous pointers. So, we could just track the thread ID and move those numbers between active and blocked.
3. Our priority scheduler uses a separate linked list for each priority, always choosing the highest priority threads first. Alternatively, we could have one large linked list and search through for the most important thread in every context switch. Searching through one list has O(n) time complexity with the number of threads, so it is inefficient for 100 tasks. For 100 threads, our implementation is mostly acceptable because even if all threads had the same priority, the scheduler would just choose the next thread in a round robin fashion. However, if there was a large number of sleeping threads, the algorithm would need to search through each list in turn. Another method would be to have a matrix of priorities like that used by µCOS-II, which is good for a very large number of threads and many levels of priorities.
4. If all threads are blocked or sleeping, our OS would never find another task to run. We have included error checking functionality for both these conditions that could be extended to force the OS to run an idle task (which is only run when no other tasks are active). Had we not accounted for these cases, the OS would access an uninitialized memory location (address 0) with random garbage in it and attempt to execute nonexistent threads, eventually crashing. It would have invalid stack pointers, link register values, and addresses.
5. If a foreground thread returns, it will execute a BX LR. Since the last value in the LR (assuming the task was not called by another function) is 0x14141414 (the same value when the stack was initialized), the program will hard fault or crash. That section of memory may be protected. We should prepopulate the LR location (when setting up the stack for each thread) with the address of a special function that is only called when a foreground thread tries to return. This function could kill the offending thread or simply make sure the OS doesn’t crash.
6. Spinlock semaphores require less overhead than blocking semaphores. Threads do not have to be moved between linked lists, and the context switch algorithm is much simpler. Less code and memory is also needed. On the other hand, the OS wastes time context switching and executing a thread that is not ready to run when it could be performing other tasks. Additionally, spinlock semaphores do not work with priority schedulers. Blocking semaphores solve both problems if implemented efficiently. They ensure that the highest priority thread that has been waiting the longest is always executed first. This enforces a greater degree of fairness compared to spinlocks.
7. Probability that a particular place in T2 is never selected after n interrupts:

Probability that all locations were interrupted at least once: