**EE445M – Lab 2: RTOS Kernel**

Sourabh Shirhatti and Nelson Wu, Spring 2015

**Objectives**

The objective of this lab was to develop OS facilities for real-time applications. In particular, our RTOS will be used to control a robot later in the semester. We coordinated multiple foreground and background threads using a round robin multi-thread scheduler and spinlock semaphores for thread synchronization. The threads communicated with each other through FIFOs and a mailbox. Switch interrupts and thread sleeping were also implemented.

**Hardware Design**

None

**Software Design**

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

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

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

// Used with Testmain2; comment out when running Testmain1

#define WITH\_SYSTICK

// 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 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 16

struct tcb{

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

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

int32\_t \*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 #

uint32\_t priority; // used in priority scheduling

};

typedef struct tcb tcbType;

tcbType tcbs[NUMTHREADS];

tcbType \*RunPt;

tcbType \*NextRunPt;

int32\_t Stacks[NUMTHREADS][STACKSIZE];

static int i;

int available[NUMTHREADS];

void InitAvailable(void) {

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

available[i] = 1;

}

}

int add\_thread() {

int ret;

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

if (available[i]) {

ret = i;

available[i] = 0;

return ret;

}

}

return -1;

}

int delete\_thread(int thread) {

if (available[thread]) {

return -1;

// Cannot release thread which is already available

}

available[thread] = 1;

return 0;

}

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;

}

int find\_next(int thread) {

int ret;

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

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 50 MHz

InitAvailable();

InitTimer2A(TIME\_1MS);

InitTimer3A(80);

UART\_Init(); // initialize UART

Output\_Init();

// SystemTime = 0;

MSTime = 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) {

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

}

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

// decrement semaphore

// Lab2 spinlock

// Lab3 block if less than zero

// input: pointer to a counting semaphore

// output: none

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();

}

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

// increment semaphore

// Lab2 spinlock

// Lab3 wakeup blocked thread if appropriate

// input: pointer to a counting semaphore

// output: none

void OS\_Signal(Sema4Type \*semaPt) {

int32\_t status;

status = StartCritical();

semaPt->Value += 1;

EndCritical(status);

}

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

// Lab2 spinlock, set to 0

// Lab3 block if less than zero

// input: pointer to a binary semaphore

// output: none

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();

}

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

// Lab2 spinlock, set to 1

// Lab3 wakeup blocked thread if appropriate

// input: pointer to a binary semaphore

// output: none

void OS\_bSignal(Sema4Type \*semaPt) {

int32\_t status;

status = StartCritical();

semaPt->Value = 1;

EndCritical(status);

}

//\*\*\*\*\*\*\*\* 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;

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

unsigned long stackSize, unsigned long priority) {

int32\_t status, thread, prev;

status = StartCritical();

if(NumThreads == 0) {

add\_thread();

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

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

}

else {

// NumThreads++;

thread = add\_thread();

prev = find\_prev(thread);

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

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

// tcbs[find\_prev(thread)].next = &tcbs[thread];

// tcbs[thread].next = &tcbs[find\_next(thread)];

}

tcbs[thread].status = 0;

tcbs[thread].sleepCt = 0;

tcbs[thread].age = 0;

tcbs[thread].id = thread;

SetInitialStack(thread);

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

NumThreads++;

EndCritical(status);

return 1; // successful

}

//\*\*\*\*\*\*\*\* 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;

}

void (\*PeriodicTask)(void);

void InitTimer1A(uint32\_t period) {

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)|(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)();

}

//\*\*\*\*\*\*\*\* 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) {

PeriodicTask = task;

InitTimer1A(period);

return 1;

}

uint32\_t LastPF4, LastPF0;

void (\*SWOneTask)(void);

void SWOneInit(void){

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\_IEV\_R &= ~0x10; // PF4,PF0 falling edge event

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; // (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)|0x00A00000; // (g) priority 2

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

}

void (\*SWTwoTask)(void);

void SWTwoInit(void){

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 not both edges

GPIO\_PORTF\_IEV\_R &= ~0x01; // PF4,PF0 falling edge event

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)|0x00A00000; // (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();

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();

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\_Kill \*\*\*\*\*\*\*\*\*\*\*\*

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

// input: none

// output: none

void OS\_Kill(void) {

uint32\_t thread, next, prev;

thread = RunPt->id;

// next = find\_next(thread);

prev = find\_prev(thread);

delete\_thread(thread);

NumThreads--;

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

OS\_Suspend();

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

}

// \*\*\*\*\*\*\*\* 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) {

#ifdef WITH\_SYSTICK

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

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

#else

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\_bWait(&BoxFree);

MailBox = data;

OS\_bSignal(&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\_bWait(&DataValid);

retVal = MailBox;

OS\_bSignal(&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)|(2 << 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].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

StartOS(); // start on the first task

}

void SysTick\_Handler(void) {

NextRunPt = RunPt->next;

while(NextRunPt->sleepCt) {

NextRunPt = NextRunPt->next;

}

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

}

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

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

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

// filename \*\*\*\*\*\*\*\*\*\*OS.H\*\*\*\*\*\*\*\*\*\*\*

// Real Time Operating System for Labs 2 and 3

// Jonathan W. Valvano 1/27/14, valvano@mail.utexas.edu

// EE445M/EE380L.6

// You may use, edit, run or distribute this file

// You are free to change the syntax/organization of this file

// You are required to implement the spirit of this OS

#ifndef \_\_OS\_H

#define \_\_OS\_H 1

// edit these depending on your clock

#define TIME\_1MS 80000

#define TIME\_2MS (2\*TIME\_1MS)

#define TIME\_500US (TIME\_1MS/2)

#define TIME\_250US (TIME\_1MS/5)

// feel free to change the type of semaphore, there are lots of good solutions

struct Sema4{

long Value; // >0 means free, otherwise means busy

// add other components here, if necessary to implement blocking

};

typedef struct Sema4 Sema4Type;

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

// initialize semaphore

// input: pointer to a semaphore

// output: none

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

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

// decrement semaphore

// Lab2 spinlock

// Lab3 block if less than zero

// input: pointer to a counting semaphore

// output: none

void OS\_Wait(Sema4Type \*semaPt);

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

// increment semaphore

// Lab2 spinlock

// Lab3 wakeup blocked thread if appropriate

// input: pointer to a counting semaphore

// output: none

void OS\_Signal(Sema4Type \*semaPt);

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

// Lab2 spinlock, set to 0

// Lab3 block if less than zero

// input: pointer to a binary semaphore

// output: none

void OS\_bWait(Sema4Type \*semaPt);

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

// Lab2 spinlock, set to 1

// Lab3 wakeup blocked thread if appropriate

// input: pointer to a binary semaphore

// output: none

void OS\_bSignal(Sema4Type \*semaPt);

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

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

unsigned long stackSize, unsigned long priority);

//\*\*\*\*\*\*\*\* 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);

//\*\*\*\*\*\*\*\* 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);

//\*\*\*\*\*\*\*\* 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);

//\*\*\*\*\*\*\*\* 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);

// \*\*\*\*\*\*\*\* 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);

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

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

// input: none

// output: none

void OS\_Kill(void);

// \*\*\*\*\*\*\*\* 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);

// \*\*\*\*\*\*\*\* 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);

// \*\*\*\*\*\*\*\* 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);

// \*\*\*\*\*\*\*\* 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);

// \*\*\*\*\*\*\*\* 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);

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

// Initialize communication channel

// Inputs: none

// Outputs: none

void OS\_MailBox\_Init(void);

// \*\*\*\*\*\*\*\* 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\_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);

// \*\*\*\*\*\*\*\* 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);

// \*\*\*\*\*\*\*\* 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);

// \*\*\*\*\*\*\*\* 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);

// \*\*\*\*\*\*\*\* 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);

//\*\*\*\*\*\*\*\* 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);

#endif

;/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

;/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

; osasm.s: low-level OS commands, written in assembly \*/

; 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/

; \*/

PE3 EQU 0x40024020

AREA |.text|, CODE, READONLY, ALIGN=2

THUMB

REQUIRE8

PRESERVE8

EXTERN RunPt ; currently running thread

EXTERN NextRunPt ; calculated next thread

EXPORT OS\_DisableInterrupts

EXPORT OS\_EnableInterrupts

EXPORT StartOS

EXPORT PendSV\_Handler

OS\_DisableInterrupts

CPSID I

BX LR

OS\_EnableInterrupts

CPSIE I

BX LR

PendSV\_Handler ; 1) Saves R0-R3,R12,LR,PC,PSR

CPSID I ; 2) Prevent interrupt during switch

PUSH {R4-R11} ; 3) Save remaining regs r4-11

LDR R0, =RunPt ; 4) R0=pointer to RunPt, old thread

LDR R1, [R0] ; R1 = RunPt

STR SP, [R1] ; 5) Save SP into TCB

LDR R1, =NextRunPt ; 5.1) R1=pointer to NextRunPt, new thread

LDR R1, [R1] ; 6) R1 = NextRunPt

STR R1, [R0] ; RunPt = R1

LDR SP, [R1] ; 7) new thread SP; SP = RunPt->sp;

POP {R4-R11} ; 8) restore regs r4-11

CPSIE I ; 9) tasks run with interrupts enabled

BX LR ; 10) restore R0-R3,R12,LR,PC,PSR

StartOS

LDR R0, =RunPt ; currently running thread

LDR R2, [R0] ; R2 = value of RunPt

LDR SP, [R2] ; new thread SP; SP = RunPt->stackPointer;

POP {R4-R11} ; restore regs r4-11

POP {R0-R3} ; restore regs r0-3

POP {R12}

POP {LR} ; discard LR from initial stack

POP {LR} ; start location

POP {R1} ; discard PSR

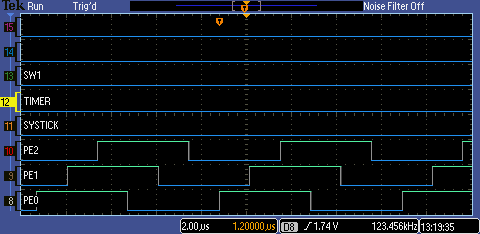
CPSIE I ; Enable interrupts at processor level

BX LR ; start first thread

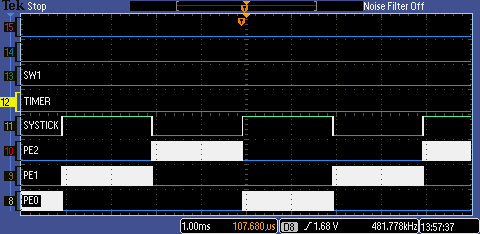
ALIGN

END

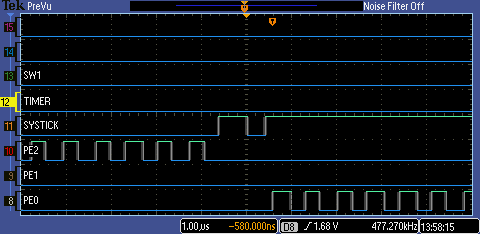
**Measurement Data**



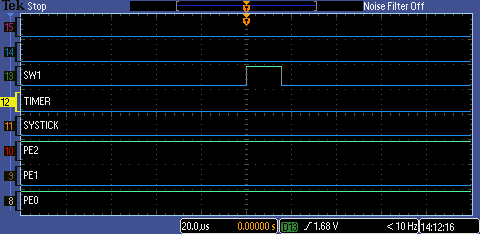
**Figure 1. Testmain1 (cooperative thread switching)**

****

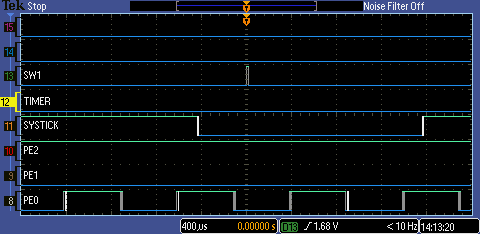
**Figure 2. Testmain2 (preemptive thread switch zoomed out)**

****

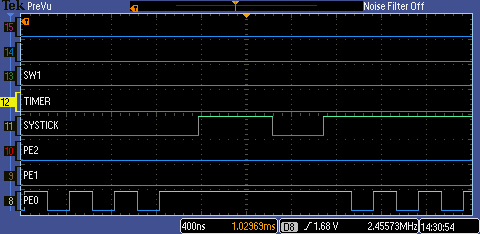
**Figure 3. Testmain2 (preemptive thread switch zoomed in)**



**Figure 4. Latency of Task 2 (switch interrupt)**

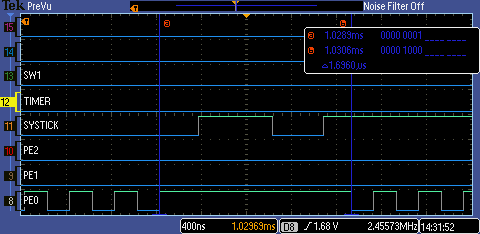
****

**Figure 5. Latency Zoomed Out**

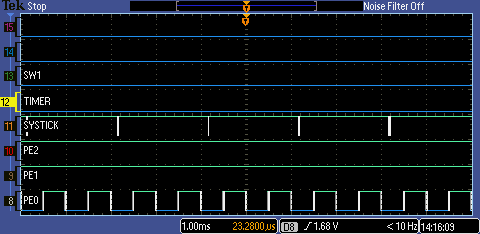
****

**Figure 6. Testmain7 Thread Switch Time**

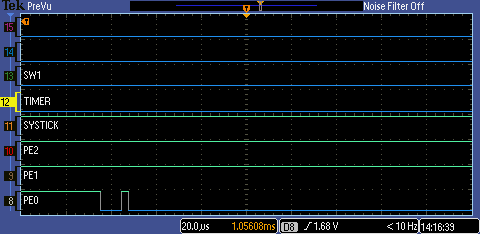
1. Measurement of thread switch time: 1.696 µs



**Figure 7. Measurement of Thread Switch Time**

1. 

**Figure 8. Profile data while running the spinlock/round-robin OS**



**Figure 9. Zoomed in view of Figure 8**

|  |
| --- |
| **TCB** |
| sp |
| next |

|  |
| --- |
| [Rest of Stack] |
| R4 – R11 |
| R0- R3 |
| R12 |
| LR |
| PC |
| PSR |

1. Preparation 3: RunPt

|  |
| --- |
| **Thread1** |
| sp: 0x200007CC |
| next: 0x20000120 |

|  |
| --- |
| **Thread2** |
| sp: 0x2000095C |
| next: 0x2000013C |

|  |
| --- |
| **Thread3** |
| sp: 0x20000AEC |
| next: 0x20000104 |

Preparation 5:

**Before:** RunPt

|  |
| --- |
| **Thread1** |
| sp: 0x200007CC |
| next: 0x20000120 |

|  |
| --- |
| **Thread2** |
| sp: 0x2000095C |
| next: 0x20000104 |

**After:** RunPt

|  |
| --- |
| **Thread1** |
| sp: 0x200007C8 |
| next: 0x20000120 |

|  |
| --- |
| **Thread2** |
| sp: 0x2000095C |
| next: 0x20000104 |

1. Performance data collected with no interpreter or switch input:

**Table 1. Performance Measurements vs. Sizes of OS\_Fifo and Timeslices**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **FIFOSize** | **TIMESLICE (ms)** | **DataLost** | **Jitter (µs)** | **PIDWork** |
| 4 | 2 | 0 | 0 | 12158 |
| 8 | 2 | 0 | 0 | 12158 |
| 32 | 2 | 0 | 0 | 12158 |
| 32 | 1 | 0 | 0 | 12120 |
| 32 | 10 | 0 | 0 | 12189 |

1. Performance measurements with and without debugging instruments:

**Table 2. Performance Measurements vs. Use of Debugging Instruments**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **With Debug?** | **TIMESLICE (ms)** | **DataLost** | **Jitter (µs)** | **PIDWork** |
| Yes/No | 0.1 | 39/0 | 0.2/0 | 8837/11422 |
| Yes/No | 1 | 0/0 | 4.7/0 | 9366/12120 |
| Yes/No | 2 | 96/0 | 5.5/0 | 9217/12158 |
| Yes/No | 5 | 434/0 | 3.8/0 | 8934/12181 |
| Yes/No | 10 | 44/0 | 2.7/0 | 11475/12189 |

**Analysis and Discussion**

1) Why did the time jitter in my solution jump from 4 to 6 μs when interpreter I/O occurred?

The timer interrupt had a lower priority than the UART Handler. When interpreter I/O occurs, jitter increases since we are waiting for the UART Handler to return.

2) Justify why Task 3 has no time jitter on its ADC sampling.

Task 3 uses timer triggered ADC interrupts in hardware. By definition, it can have no jitter since the ADC will be sampled immediately at the specified period times.

3) There are four (or more) interrupts in this system DAS, ADC, Select, and SysTick (thread switch). Justify your choice of hardware priorities in the NVIC?

SysTick (7) – Least priority since we want the thread switch to only interrupt a foreground task

PendSV (6) – We want the PendSV to be able to interrupt on SysTick completion, but not interrupt any other threads.

Select (5) – We want to reduce the switch latency. However, it is less critical than the pair of ADC tasks because they must always execute at the specified period to sample a signal correctly.

ADC (4) – Timer triggered ADC must execute at a fixed rate (in hardware).

DAS (4) – Software triggered ADC also executes at a fixed rate with a different timer, so this task has the same priority as the first ADC task.

4) Explain what happens if your stack size is too small. How could you detect stack overflow? How could you prevent stack overflow from crashing the OS?

If the stack is too small, you will have a stack overflow that will write data to memory you do not own, causing other data to be lost or corrupted.

To prevent stack overflows, we could check whether the stack pointer is still in a valid location prior to performing a context switch (for a fixed stack size). Alternatively, we could run the OS and initialize all stack locations to a dummy value. At some point, we could look at the stack and see how many locations have been overwritten, and that will be our maximum stack size.

To prevent a stack overflow from crashing the OS, we could kill the thread that violated the memory accesses. Kill any additional threads whose stacks were corrupted. The user program should be able to handle the terminated threads.

5) Both Consumer and Display have an OS\_Kill() at the end. Do these OS\_Kills always execute, sometime execute, or never execute? Explain.

Consumer

In the event that data is lost, the consumer thread cannot finish executing as it waits for 64 entries in the mailbox before computing an FFT. Hence, it will never reach the OS\_Kill().

In the event that data is not lost, or a multiple of 64 entries are lost, OS\_Kill() will get executed.

Display

Similarly, for Display, in the event of lost data, OS\_Kill() will not execute. However, if there is no data lost, OS\_Kill() will execute.

6) The interaction between the producer and consumer is deterministic. What does deterministic mean? Assume for this question that if the OS\_Fifo has 5 elements data is lost, but if it has 6 elements no data is lost. What does this tell you about the timing of the consumer plus display?

Deterministic means there is no probability, and the order of execution of the producer and consumer is guaranteed. In other words, the producer will put data into the FIFO, the consumer will remove data from the FIFO, and each will wait if the FIFO is full or empty, respectively. If data is lost with five elements and not with six elements, this means the consumer plus display removes data at a rate that is too slow for the producer with five, but not with six.

7) Without going back and actually measuring it, do you think the Consumer ever waits when it calls OS\_MailBox\_Send? Explain

Our system is I/O bound. The display thread spawned by the consumer that shares the mailbox with consumer will take longer to execute because writing to the LCD is slower than reading from the ADC and calculating the FFT. The display will hold the semaphore and the consumer will have to wait when it calls OS\_Mailbox\_Send.