

```

// RTOS
// Student Name: Dhaval Padia
// T0 D0: Add your name on this line. Do not include your ID number.

// Submit only two .c files in an e-mail to me (not in a compressed
file):
// xx_rtos_coop.c   Single-file with cooperative version of your
project
// xx_rtos_preempt.c Single-file with preemptive version of your project
// (xx is a unique number that will be issued in class)
// Please do not include .intvecs section in your code submissions
// Please do not change any function names in this code or thread
priorities

//-----
// Hardware Target
//-----

// Target Platform: EK-TM4C123GXL Evaluation Board
// Target uC:      TM4C123GH6PM
// System Clock:   40 MHz

// Hardware configuration:
// 5 Pushbuttons and 5 LEDs, UART

//-----
// Device includes, defines, and assembler directives
//-----

#include <stdint.h>
#include <stdbool.h>
#include <ctype.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include "tm4c123gh6pm.h"

// REQUIRED: correct these bitbanding references for the off-board
LEDs
#define BLUE_LED      (*((volatile uint32_t *) (0x42000000 +
(0x400253FC-0x40000000)*32 + 2*4))) // on-board blue LED
#define RED_LED       (*((volatile uint32_t *) (0x42000000 +
(0x400073FC-0x40000000)*32 + 2*4))) // off-board red LED
#define ORANGE_LED    (*((volatile uint32_t *) (0x42000000 +
(0x400243FC-0x40000000)*32 + 2*4))) // off-board orange LED

```

```

#define YELLOW_LED    (*((volatile uint32_t *) (0x42000000 +
(0x400073FC-0x40000000)*32 + 3*4))) // off-board yellow LED
#define GREEN_LED     (*((volatile uint32_t *) (0x42000000 +
(0x400243FC-0x40000000)*32 + 1*4))) // off-board green LED
#define PUSH_BUTTON0  (*((volatile uint32_t *) (0x42000000 +
(0x400063FC-0x40000000)*32 + 7*4))) //BUTTON 0
#define PUSH_BUTTON1  (*((volatile uint32_t *) (0x42000000 +
(0x400063FC-0x40000000)*32 + 6*4))) //BUTTON 1
#define PUSH_BUTTON2  (*((volatile uint32_t *) (0x42000000 +
(0x400063FC-0x40000000)*32 + 5*4))) //BUTTON 2
#define PUSH_BUTTON3  (*((volatile uint32_t *) (0x42000000 +
(0x400063FC-0x40000000)*32 + 4*4))) //BUTTON 3
#define PUSH_BUTTON4  (*((volatile uint32_t *) (0x42000000 +
(0x400053FC-0x40000000)*32 + 3*4))) //BUTTON 4

//-----
// RTOS Defines and Kernel Variables
//-----

// function pointer
typedef void (*_fn)();

// semaphore
#define MAX_SEMAPHORES 5
#define MAX_QUEUE_SIZE 5
struct semaphore
{
    uint16_t count;
    uint16_t queueSize;
    uint32_t processQueue[MAX_QUEUE_SIZE]; // store task index here
    char semaname[30];
} semaphores[MAX_SEMAPHORES];
uint8_t semaphoreCount = 0;

struct semaphore *keyPressed, *keyReleased, *flashReq, *resource;

// task
#define STATE_INVALID    0 // no task
#define STATE_UNRUN      1 // task has never been run
#define STATE_READY      2 // has run, can resume at any time
#define STATE_BLOCKED    3 // has run, but now blocked by semaphore
#define STATE_DELAYED    4 // has run, but now awaiting timer
uint8_t fieldOffset[50];
uint8_t fieldCount;
char fieldType[50];
char str[30], str1[20], str2[20], str3[20];
#define MAX_TASKS 10 // maximum number of valid tasks

```

```

uint8_t taskCurrent = 0;    // index of last dispatched task
uint8_t taskCount = 0;     // total number of valid tasks
uint8_t flag = 0, flagone=0;
uint32_t counts;

//uint8_t next_Task;
struct _tcb
{
    uint8_t state;           // see STATE_ values above
    void *pid;              // used to uniquely identify thread
    void *sp;               // location of stack pointer for
thread
    uint8_t priority;        // 0=highest, 15=lowest
    uint8_t currentPriority; // used for priority inheritance
    uint32_t ticks;          // ticks until sleep complete
    char name[16];          // name of task used in ps command
    void *semaphore;        // pointer to the semaphore that is
blocking the thread
    uint8_t skipCount;       // priority counter
    uint32_t countStart;
    uint32_t countStop;
    uint32_t countSum;
    void *semainfo;
} tcb[MAX_TASKS];

uint32_t stack[MAX_TASKS][256];

//-----
// RTOS Kernel Functions
//-----

void putcUart0(char c)
{
    while (UART0_FR_R & UART_FR_TXFF)
        ;
    UART0_DR_R = c;
}

// Blocking function that writes a string when the UART buffer is not
full
void putsUart0(char* str)
{
    uint8_t i;
    for (i = 0; i < strlen(str); i++)
        putcUart0(str[i]);
}

```

```

// Blocking function that returns with serial data once the buffer is
not empty
char getcUart0()
{
    while (UART0_FR_R & UART_FR_RXFE)
        yield();
    return UART0_DR_R & 0xFF;
}

void getString()
{
    unsigned char i = 0;
    uint8_t l = 0;
    while (i < 81)
    {
        str[i] = getcUart0(); // calling string characters

        if (str[i] == 8) // backspace
        {
            if (i > 0) // if count is more then zero then it will
return back to previous count
            {
                --i;
                continue;
            }
        }
        else if (str[i] == 13) // enter
        {
            str[i] = 0; //null
            break; // break the loop
        }
        else if (str[i] < 32) //signs below space are neglected
        {
            continue;
        }

        else if ((str[i] >= 32) && (str[i] <= 127)) // values which
are to be considered
        {
            i++;
            continue;
        }

        else if (str[i] != 13) // if string is not entered then it
will call yield everytime
        {

```

```

        yield();
    }
}

i = 0;
l = strlen(str); // defining string length with a integer

while (str[i] != 0) // if string is not zero
{
    if ((str[i] >= 65) && (str[i] <= 90)) // capital alphabets
    {
        str[i] = (tolower(str[i])); // converting capital to small
    }
    /* in below function we have converted all the unwanted with
null char */
    if (((str[i] >= 32) && (str[i] <= 37)) || (str[i] == '/')
        || ((str[i] >= 58) && (str[i] <= 64))
        || ((str[i] >= 91) && (str[i] <= 94)) || (str[i] ==
96)
        || ((str[i] >= 123) && (str[i] <= 127)))
    {
        str[i] = 0;
    }

    i++;
}

if (l > 0) // comparing length , if length is not zero , just for
the precaution . here we know that 'i' is equal to 'l'
{
    for (i = 0; i < l; i++)
    {
        if ((str[i] == NULL) && ((str[i + 1] >= 97) && (str[i + 1]
<= 122))) // if string is null and next string char is between small
alphabets then
        {
            fieldType[fieldCount] = 'a'; // insert fieldType as
'a'
            fieldOffset[fieldCount] = i + 1; // offset will set
its value where the filedType is pointed
            fieldCount++; // increased by one
        }
        if ((fieldCount == 0) && ((str[i] >= 97) && (str[i] <=
122))) // this will execute when there is no null char at starting of
the string
        {

```

```

        fieldType[fieldCount] = 'a';
        fieldOffset[fieldCount] = i;
        fieldCount++;
    }

    if ((str[i] == NULL) && ((str[i + 1] >= 38) && (str[i + 1]
<= 57))) // if string is null and next string char is between number
then

    {
        fieldType[fieldCount] = 'n'; // filedType will insert
'n' where fieldCount position is pointed currently
        fieldOffset[fieldCount] = i + 1; // increment the
filedOffset
        fieldCount++; // increment filedCount whenever
fieldType is encountered
    }

    putchar0(str[i]); // just to print on console.

}

}

}

void parse() //
{
    uint8_t a = 0;
    uint8_t b = 0;
    uint8_t c = 0;
    uint8_t i = 0;
    if ((fieldType[0] == 'a') && (fieldOffset[0] == 0) && (fieldCount
== 1)) // if there is only one word written in string
    {
        for (i = 0; i < 20; i++)
        {
            str1[a] = str[i]; // string1 will get the word which is
in main string
            a++;
            if (str[i] == NULL)
                break;
        }

    }

    else if ((fieldType[0] == 'a') && (fieldOffset[0] != 0)
&& (fieldCount == 1)) // if there is space before the word
in string then this will execute
    {

```

```

while (str[i] == NULL) // inresing the string if there is null
{
    i++;
    continue;
}

for (; i < 20; i++)
{
    str1[a] = str[i]; // string1 will get the word which is
in main string
    a++;
    if (str[i] == NULL)
        break;
}

}

else if (((fieldType[1] == 'n') || (fieldType[1] == 'a'))
        && (fieldOffset[1] != '0') && (fieldCount == 2)) // if
string is having two arguments then no matter if its alphabet or
number
{
    while (str[i] == NULL)
    {
        i++;
        continue;
    }
    for (; i <= 20; i++)
    {
        str1[a] = str[i];
        a++;
        if (str[i] == NULL)
            break;
    }
    while (str[i] == NULL)
    {
        i++;
        continue;
    }
    for (; i < 30; i++)
    {
        str2[b] = str[i];
        b++;
        if (str[i] == NULL)
            break;
    }
}

else if (((fieldType[2] == 'n') || (fieldType[2] == 'a'))

```

```

        && (fieldOffset[2] != '0') && (fieldCount == 3)) // if
there is three arguments then no matter if there is alphabet or number
    {
        while (str[i] == NULL)
        {
            i++;
            continue;
        }
        for (; i <= 20; i++)
        {

            str1[a] = str[i]; // first argument
            a++;
            if (str[i] == NULL)
                break;
        }
        while (str[i] == NULL)
        {
            i++;
            continue;
        }
        for (; i <= 30; i++)
        {
            str2[b] = str[i]; // second argument
            b++;
            if (str[i] == NULL)
                break;
        }
        while (str[i] == NULL)
        {
            i++;
            continue;
        }
        for (; i <= 40; i++)
        {
            str3[c] = str[i]; // third argument
            c++;
            if (str[i] == NULL)
                break;
        }
    }

}

void rtosInit()
{
    uint8_t i;
    // no tasks running
    taskCount = 0;
    // clear out tcb records

```

```

    for (i = 0; i < MAX_TASKS; i++)
    {
        tcb[i].state = STATE_INVALID;
        tcb[i].pid = 0;
    }
    // REQUIRED: initialize systick for 1ms system timer
    NVIC_ST_CTRL_R=0;
    NVIC_ST_CTRL_R=NVIC_ST_CTRL_CLK_SRC |
NVIC_ST_CTRL_INTEN|NVIC_ST_CTRL_ENABLE;
    NVIC_ST_RELOAD_R=0X9C3F;
    NVIC_ST_CURRENT_R=0;
}

void rtosStart()
{
    // REQUIRED: add code to call the first task to be run
    WideTimer5Isr();
    _fn fn;

    taskCurrent = rtosScheduler();
    fn=tcb[taskCurrent].pid;
    (*fn)();
    // Add code to initialize the SP with tcb[task_current].sp;
}

bool createThread(_fn fn, char name[], int priority)
{
    bool ok = false;
    uint8_t i = 0;
    bool found = false;

    // REQUIRED: store the thread name
    // add task if room in task list
    if (taskCount < MAX_TASKS)
    {
        // make sure fn not already in list (prevent reentrancy)
        while (!found && (i < MAX_TASKS))
        {
            found = (tcb[i++].pid == fn);
        }
        if (!found)
        {
            // find first available tcb record
            i = 0;
            while (tcb[i].state != STATE_INVALID) {i++;}
            tcb[i].state = STATE_UNRUN;
            tcb[i].pid = fn;
            tcb[i].sp = &stack[i][255];
            tcb[i].priority = priority;
            tcb[i].currentPriority = priority;
        }
    }
}

```

```

        tcb[i].skipCount = priority;
        strcpy(tcb[i].name,name);
//    tcb[i].skipCount= skipCount;
// increment task count
        taskCount++;
        ok = true;
    }
}
// REQUIRED: allow tasks switches again
return ok;
}

// REQUIRED: modify this function to destroy a thread
void destroyThread(_fn fn)
{
    uint8_t dest;
    uint8_t i;
    struct semaphore *pSemaphore=0;

    for (dest=0;dest<MAX_TASKS;dest++)
    {
        if (fn==tcb[dest].pid)
        {
            tcb[dest].state=STATE_INVALID;

            pSemaphore = tcb[dest].semaphore;
            for (i=0;i<taskCount;i++)
            {
                if (pSemaphore->processQueue[i]==tcb[dest].pid)
                {
                    pSemaphore->processQueue[i]=0;
                    pSemaphore->processQueue[i]= pSemaphore->
>processQueue[i+1];
                    pSemaphore->queueSize--;

                }
            }
            tcb[dest].pid=0;
        }
    }
}

// REQUIRED: modify this function to set a thread priority
void setThreadPriority(_fn fn, uint8_t priority)
{
    uint8_t i ;
    for (i=0; i<MAX_TASKS ; i++)
    {

```

```

        if(tcb[i].pid ==fn)
        {
            tcb[i].currentPriority = tcb[i].priority ;
        }
    }
}

struct semaphore* createSemaphore(uint8_t count, char semaname[])
{
    struct semaphore *pSemaphore = 0;
    if (semaphoreCount < MAX_SEMAPHORES)
    {
        strcpy(semaphores[semaphoreCount].semaname,semaname);
        pSemaphore = &semaphores[semaphoreCount++];
        pSemaphore->count = count;
    }
    return pSemaphore;
}

// REQUIRED: modify this function to yield execution back to scheduler
using pendsv
void yield()
{
    NVIC_INT_CTRL_R|=NVIC_INT_CTRL_PEND_SV;

    // push registers, call scheduler, pop registers, return to new
    function

}

// REQUIRED: modify this function to support 1ms system timer
// execution yielded back to scheduler until time elapses using pendsv
void sleep(uint32_t tick)
{
    tcb[taskCurrent].ticks=tick;
    tcb[taskCurrent].state=STATE_DELAYED;
    NVIC_INT_CTRL_R|=NVIC_INT_CTRL_PEND_SV;
    // push registers, set state to delayed, store timeout, call
    scheduler, pop registers,
    // return to new function (separate unrun or ready processing)
}

// REQUIRED: modify this function to wait a semaphore with priority
inheritance
// return if avail (separate unrun or ready processing), else yield to
scheduler using pendsv
void wait(struct semaphore *pSemaphore)
{
    uint32_t wt;

```

```

    if (pSemaphore->count >0)
    {
        pSemaphore->count--;
        tcb[taskCurrent].semainfo=pSemaphore;
    }
    else
    {

        tcb[taskCurrent].state=STATE_BLOCKED;
        pSemaphore->processQueue[pSemaphore->queueSize]= taskCurrent;
        pSemaphore->queueSize++;
        tcb[taskCurrent].semaphore = pSemaphore;
        for (wt=0;wt<taskCount;wt++)
        {
            if(tcb[taskCurrent].semaphore==tcb[wt].semainfo)
            {
                if(tcb[taskCurrent].priority<tcb[wt].priority)
                {
                    tcb[wt].priority=tcb[taskCurrent].priority;
                    tcb[wt].skipCount=tcb[taskCurrent].priority;
                }
            }
        }

    }

    NVIC_INT_CTRL_R|=NVIC_INT_CTRL_PEND_SV;
}

// REQUIRED: modify this function to signal a semaphore is available
using pendsv
void post(struct semaphore *pSemaphore)
{
    uint8_t sema;
    uint8_t data;
    pSemaphore->count++;
    while(tcb[taskCurrent].priority!=tcb[taskCurrent].currentPriority)
    {
        tcb[taskCurrent].priority=tcb[taskCurrent].currentPriority;
        tcb[taskCurrent].skipCount=tcb[taskCurrent].currentPriority;
    }

    if (pSemaphore->processQueue > 0)
    {
        data = pSemaphore->processQueue[0];
        tcb[data].state=STATE_READY;
        pSemaphore->queueSize--;
        pSemaphore->count--;

        for(sema=0; sema<MAX_QUEUE_SIZE-1; sema++)
        {

```

```

        pSemaphore->processQueue[sema]= pSemaphore->
>processQueue[sema+1];

    }

    }
    return;

}

```

// REQUIRED: Implement prioritization to 16 levels

```

int rtosScheduler()
{
    bool ok;
    static uint8_t task = 0xFF;
    ok = false;
    while (!ok)
    {
        task++;
        if (task >= MAX_TASKS)
            task = 0;
        if (tcb[task].state == STATE_UNRUN)
        {
            ok=true;
            return task;
        }

        else if(tcb[task].state == STATE_READY)
        {
            if (tcb[task].skipCount==0)
            {
                tcb[task].skipCount=tcb[task].priority;
                ok= true;
                return task;
            }
            else
            {
                tcb[task].skipCount--;
            }
        }
    }
}

/*
while (!ok)
{

```

```

    task++;
    if (task >= MAX_TASKS)
        task = 0;
    if (tcb[task].state == STATE_READY || tcb[task].state ==
STATE_UNRUN)
    {

        if (tcb[task].skipCount==0)
        {
            tcb[task].skipCount=tcb[task].priority;
            ok= true;
            return task;
        }
        else
        {
            tcb[task].skipCount--;
        }

    }

}*/
// return task;
}
void* storeSp()
{

    __asm("  MOV R0 , SP " );
}
void* restoreSp(void *x)
{

    // __asm("  MOV SP , R0 " );
    __asm("  BX LR " );
}

/*void* sp_Init()
{
    uint32_t s;
    s=tcb[taskCurrent].sp;
    __asm( "      MOV SP, R0      ");
}*/

// REQUIRED: modify this function to add support for the system timer
// REQUIRED: in preemptive code, add code to request task switch
void systickIsr()
{
    uint32_t thrds;
    for(thrds=0;thrds<MAX_TASKS;thrds++)
    {

```

```

        if(tcb[thrds].state == STATE_DELAYED)
        {
            --tcb[thrds].ticks;
            if(tcb[thrds].ticks==0)
            {
                tcb[thrds].state=STATE_READY;
            }
        }
    }
}

```

// REQUIRED: modify this function to add support for the service call
// REQUIRED: in preemptive code, add code to handle synchronization primitives

```

void svCallIsr()
{
}

```

// REQUIRED: in coop and preemptive, modify this function to add support for task switching
// REQUIRED: process UNRUN and READY tasks differently

```

void pendSvIsr()
{
    // __asm ( "      POP      {R1-R3,LR} " );
    __asm ( "      PUSH      { R4 - R11 } " );
    tcb[taskCurrent].sp = storeSp() ;
    taskCurrent = rtosScheduler();
    restoreSp(tcb[taskCurrent].sp);
    __asm( "      MOV  SP,  R0  " );
    if ((tcb[taskCurrent].state) == STATE_READY)
    {
        // __asm ( "      PUSH      {R1-R3,LR} " );
        __asm ( "      POP      { R4-R11 } " );

        /* __asm (" POP      {LR } ");
        __asm (" POP      { R3} ");
        __asm (" POP      { R2} ");
        __asm (" POP      { R1} "); */

        // tcb[taskCurrent].sp=storeSp();
    }
    else if ((tcb[taskCurrent].state) == STATE_UNRUN)

//z=tcb[taskCurrent].pid;
    {
        tcb[taskCurrent].state = STATE_READY;
    }
}

```

```

uint32_t x;
x= 0x01000000;
__asm ( " PUSH      {R0 } " );
uint32_t z;
z=tcb[taskCurrent].pid;
__asm ( " PUSH      {R0 } " );
__asm ( " PUSH      {R0 } " );
__asm ( " PUSH      {R12} " );
__asm ( " PUSH      { R3} " );
__asm ( " PUSH      { R2} " );
__asm ( " PUSH      { R1} " );
__asm ( " PUSH      { R0} " );
uint32_t y=0xFFFFFFFF;
__asm ( " PUSH      {R0 } " );
__asm ( " PUSH      { R2} " );
__asm ( " PUSH      { R1} " );
__asm ( " PUSH      { R0} " );

}

/* {
    stack[taskCurrent][255]= 0x01000000;
    stack[taskCurrent][254]=tcb[taskCurrent].pid;
    stack[taskCurrent][253]=tcb[taskCurrent].pid;
    stack[taskCurrent][252]=0x12;
    stack[taskCurrent][251]=0x3;
    stack[taskCurrent][250]=0x2;
    stack[taskCurrent][249]=0x1;
    stack[taskCurrent][248]=0x0;
    stack[taskCurrent][247]=0x11;
    stack[taskCurrent][246]=0x10;
    stack[taskCurrent][245]=0x9;
    stack[taskCurrent][244]=0x8;
    stack[taskCurrent][243]=0x7;
    stack[taskCurrent][242]=0x6;
    stack[taskCurrent][241]=0x5;
    stack[taskCurrent][240]=0x4;

    // __asm ( "      ADD  SP, #3C " );
    //__asm("      MVN      SP, ~# 0x2000007E0 ");

    }*/
    //__asm ( "      MOV  R2, #FFFFFFF9 " );
    // __asm ( "      LDR  LR, 0xFFFFFFFF " );
    //y=&stack[taskCurrent][248];
    //__asm ( " MOV  SP, R0 " );

```

```

    __asm ( " MVN LR, #6 " );
    return;
}

//-----
// Subroutines
//-----

// Initialize Hardware
void initHw()
{
    // Configure HW to work with 16 MHz XTAL, PLL enabled, system
    // clock of 40 MHz
    SYSCTL_RCC_R = SYSCTL_RCC_XTAL_16MHZ |
SYSCTL_RCC_OSCSRC_MAIN
    | SYSCTL_RCC_USESYSDIV | (4 <<
SYSCTL_RCC_SYSDIV_S);
    //ports
    SYSCTL_RCGC2_R = SYSCTL_RCGC2_GPIOA | SYSCTL_RCGC2_GPIOF
    | SYSCTL_RCGC2_GPIOE | SYSCTL_RCGC2_GPIOD |
SYSCTL_RCGC2_GPIOB
    | SYSCTL_RCGC2_GPIOC;

    GPIO_PORTA_DIR_R |= 0x3C; // make bit 2,3,4,5 an output
PORTA
    GPIO_PORTA_DR2R_R |= 0x3C; // set drive strength to 2mA
PORTA
    GPIO_PORTA_DEN_R |= 0xFF; // enable bit 2,3,4,5 for
digital PORTA (WITH UART)

    GPIO_PORTB_DEN_R |= 0x08; // enable ALL bit for
digital PORTB (WITH UART)
    GPIO_PORTB_PUR_R = 0x08; //enable internal pull-
up for push button

    GPIO_PORTC_DEN_R |= 0xF0; // enable bit 2,3,4,5
for digital PORTA (WITH UART)
    GPIO_PORTC_PUR_R = 0xF0; //enable internal pull-
up for push button

    GPIO_PORTD_DIR_R |= 0x0C; // make bit 2,3,4,5
an output PORTA
    GPIO_PORTD_DR2R_R |= 0x0C; // set drive strength
to 2mA PORTA

```

```

        GPIO_PORTD_DEN_R |= 0x0C; // enable bit 2,3,4,5
for digital PORTA (WITH UART)

        GPIO_PORTE_DIR_R |= 0x06; // make bit 2,3,4,5
an output PORTA
        GPIO_PORTE_DR2R_R |= 0x06; // set drive strength
to 2mA PORTA
        GPIO_PORTE_DEN_R |= 0x06; // enable bit 2,3,4,5
for digital PORTA (WITH UART)

        GPIO_PORTF_DIR_R |= 0x04; // make bit 2,3,4,5
an output PORTA
        GPIO_PORTF_DR2R_R |= 0x04; // set drive strength
to 2mA PORTA
        GPIO_PORTF_DEN_R |= 0x04; // enable bit 2,3,4,5
for digital PORTA (WITH UART)

        // Configure UART0 pins
        SYSCTL_RCGCUART_R |= SYSCTL_RCGCUART_R0; //
turn-on UART0, leave other uarts in same status
        //GPIO_PORTA_DEN_R |= 3;
// default, added for clarity
        GPIO_PORTA_AFSEL_R |= 3;
// default, added for clarity
        GPIO_PORTA_PCTL_R = GPIO_PCTL_PA1_U0TX |
GPIO_PCTL_PA0_U0RX;

        // Configure UART0 to 115200 baud, 8N1 format
(must be 3 clocks from clock enable and config writes)
        UART0_CTL_R = 0; // turn-off
UART0 to allow safe programming
        UART0_CC_R = UART_CC_CS_SYSCCLK;
// use system clock (40 MHz)
        UART0_IBRD_R = 21; // r = 40 MHz /
(Nx115.2kHz), set floor(r)=21, where N=16
        UART0_FBRD_R = 45;
// round(fract(r)*64)=45
        UART0_LCRH_R = UART_LCRH_WLEN_8 |
UART_LCRH_FEN; // configure for 8N1 w/ 16-level FIFO
        UART0_CTL_R = UART_CTL_TXE | UART_CTL_RXE |
UART_CTL_UARTEN; // enable TX, RX, and module

// wide timer for counting the thread timings
        // Configure FREQ_IN for frequency counter
        GPIO_PORTD_AFSEL_R |= 0x40; // select
alternative functions for FREQ_IN pin
        GPIO_PORTD_PCTL_R &= ~GPIO_PCTL_PD6_M;
// map alt fns to FREQ_IN

```

```

        GPIO_PORTD_PCTL_R |=
GPIO_PCTL_PD6_WT5CCP0;
        GPIO_PORTD_DEN_R |= 0x40;
// enable bit 6 for digital input

        // timer initialization

        SYSCTL_RCGCTIMER_R |=
SYSCTL_RCGCTIMER_R5; // turn-on timer
        WTIMER5_CTL_R &= ~TIMER_CTL_TAEN; //
turn-off counter before reconfiguring
        WTIMER5_CFG_R = 4; //
configure as 32-bit counter (A only)
        WTIMER5_TAMR_R = TIMER_TAMR_TACMR |
TIMER_TAMR_TAMR_CAP | TIMER_TAMR_TACDIR; // configure for edge time
mode, count up
        WTIMER5_CTL_R = TIMER_CTL_TAEVENT_POS; //
measure time from positive edge to positive edge
        WTIMER5_IMR_R = TIMER_IMR_CAEIM;
// turn-on interrupts
        WTIMER5_TAV_R = 0;
// zero counter for first period
        // WTIMER5_CTL_R |= TIMER_CTL_TAEN;
// turn-on counter
        NVIC_EN3_R |= 1 << (INT_WTIMER5A - 16 -
96); // turn-on interrupt 120 (WTIMER5A)

// REQUIRED: Add initialization for blue, orange, red, green, and
yellow LEDs
//          5 pushbuttons, and uart
}

// Approximate busy waiting (in units of microseconds), given a 40 MHz
system clock
void waitMicrosecond(uint32_t us)
{
        // Approx clocks per
us
        __asm("WMS_LOOP0:  MOV  R1, #6"); // 1
        __asm("WMS_LOOP1:  SUB  R1, #1"); // 6
        __asm("            CBZ  R1, WMS_DONE1"); // 5+1*3
        __asm("            NOP"); // 5
        __asm("            B    WMS_LOOP1"); // 5*3
        __asm("WMS_DONE1:  SUB  R0, #1"); // 1
        __asm("            CBZ  R0, WMS_DONE0"); // 1
        __asm("            B    WMS_LOOP0"); // 1*3
        __asm("WMS_DONE0:"); // ---

```

```

// 40 clocks/us +
error
}
void setCounterMode()
{
    SYSCTL_RCGCWTIMER_R |= SYSCTL_RCGCWTIMER_R5;    // turn-on timer
    WTIMER5_CTL_R &= ~TIMER_CTL_TAEN;    // turn-off counter before
reconfiguring
    WTIMER5_CFG_R = 4;    // configure as 32-bit
counter (A only)
    WTIMER5_TAMR_R = TIMER_TAMR_TAMR_CAP | TIMER_TAMR_TACDIR; //
configure for edge count mode, count up
    WTIMER5_CTL_R = 0;    //
    WTIMER5_IMR_R = TIMER_IMR_TATOIM;    //
turn-on interrupts
    WTIMER5_TAV_R = 0;    // zero counter for
first period
    WTIMER5_CTL_R |= TIMER_CTL_TAEN;    // turn-on
counter
    NVIC_EN3_R &= ~(1 << (INT_WTIMER5A - 16 - 96)); // turn-off
interrupt 120 (WTIMER5A)
}
void WideTimer5Isr()
{
    counts++;
    WTIMER5_ICR_R =TIMER_ICR_TATOCINT;
}

// REQUIRED: add code to return a value from 0-31 indicating which of
5 PBs are pressed
uint8_t readPbs()
{
    while(PUSH_BUTTON0==0)
    {
        return 1;
    }
    while(PUSH_BUTTON1==0)
    {
        return 2;
    }
    while(PUSH_BUTTON2==0)
    {
        return 4;
    }
    while(PUSH_BUTTON3==0)
    {
        return 8;
    }
    while(PUSH_BUTTON4==0)

```

```

        {
            return 16;
        }
    return 0;
}

// -----
// Task functions
// -----

// one task must be ready at all times or the scheduler will fail
// the idle task is implemented for this purpose
void idle()
{
    while(true)
    {
        ORANGE_LED = 1;
        waitMicrosecond(1000);
        ORANGE_LED = 0;
        tcb[taskCurrent].state = STATE_READY;
        yield();
    }
}

void idle2()
{
    while(true)
    {
        RED_LED = 1;
        waitMicrosecond(10000);
        RED_LED = 0;

        yield();
    }
}

void flash4Hz()
{
    while(true)
    {
        GREEN_LED ^= 1;
        sleep(125);
    }
}

void oneshot()
{
    while(true)

```

```

    {
        wait(flashReq);
        YELLOW_LED = 1;
        sleep(1000);
        YELLOW_LED = 0;
    }
}

void partOfLengthyFn()
{
    // represent some lengthy operation
    waitMicrosecond(1000);
    // give another process a chance to run
    yield();
}

void lengthyFn()
{
    uint16_t i;
    while(true)
    {
        wait(resource);
        for (i = 0; i < 4000; i++)
        {
            partOfLengthyFn();
        }
        RED_LED ^= 1;
        post(resource);
    }
}

void readKeys()
{
    uint8_t buttons;
    while(true)
    {
        wait(keyReleased);
        buttons = 0;
        while (buttons == 0)
        {
            buttons = readPbs();
            yield();
        }
        post(keyPressed);
        if ((buttons & 1) != 0)
        {
            YELLOW_LED ^= 1;
            RED_LED = 1;
        }
        if ((buttons & 2) != 0)

```

```

    {
        post(flashReq);
        RED_LED = 0;
    }
    if ((buttons & 4) != 0)
    {
        createThread(flash4Hz, "Flash4hz", 0);
    }
    if ((buttons & 8) != 0)
    {
        destroyThread(flash4Hz);
    }
    if ((buttons & 16) != 0)
    {
        setThreadPriority(lengthyFn, 4);
    }
    yield();
}
}

```

```

void debounce()
{
    uint8_t count;
    while(true)
    {
        wait(keyPressed);
        count = 10;
        while (count != 0)
        {
            sleep(10);
            if (readPbs() == 0)
                count--;
            else
                count = 10;
        }
        post(keyReleased);
    }
}

```

```

void uncooperative()
{
    while(true)
    {
        while (readPbs() == 8)
        {
        }
        yield();
    }
}

```

```

void important()
{
    while(true)
    {
        wait(resource);
        BLUE_LED = 1;
        sleep(1000);
        BLUE_LED = 0;
        post(resource);
    }
}

int d_atoi(char *s)// https://blog.breadncup.com/2009/12/21/make-your-own-atoi-function-in-c/
{
    int num=0;
    while(*s)
    {
        num = num*10 + (*s)-'0';
        s++;
    }
    return num;
}

void ps()
{
    char string[20];
    char string1[20];
    uint8_t PS;
    uint8_t n;
    putsUart0("\n PID\t\t Name\t\t State\t\t % of CPU \n");
    for (PS=0;PS<MAX_TASKS;PS++)
    {
        if(tcb[PS].state!=STATE_INVALID)
        {
            ltoa(tcb[PS].pid,string);
            ltoa(tcb[PS].state,string1);
            putsUart0("\n");
            putsUart0(string);
            putsUart0("\t\t");
            putsUart0(tcb[PS].name);
            putsUart0("\t\t");
            putsUart0(string1);
            putsUart0("\n");
        }
    }
}

void ipcs()

```

```

{
    char string5[10];
    char string1[20];
    char string6[20];
    uint32_t* x;
    uint8_t i;
    putsUart0("\n semaphore\t\t Count\t\t Waiting \n");
    for (i=0;i<MAX_SEMAPHORES-1;i++)
    {
        x= tcb[i].semaphore;
        ltoa(semaphores[i].count,string5);
        ltoa(x,string6);
        putsUart0("\n");
        putsUart0(semaphores[i].semaname);
        putsUart0("\t\t");
        putsUart0(string5);
        putsUart0("\t\t");
        putsUart0(string6);
        putsUart0("\n");
    }
}

void kill()
{
    uint32_t i=0,y=0;
    char *pro_id;
    y= d_atoi(str2);

    for (i=0;i<MAX_TASKS;i++)
    {
        if(y==tcb[i].pid)
        {
            pro_id= tcb[i].pid;
            // pro_id-=0x10;
            destroyThread(pro_id);
        }
    }
}

}
}

void reboot()
{
    __asm("        .global _c_int00\n"
        "        b.w      _c_int00");
}

void pidof()
{

```

```

        uint32_t i=0;
        // uint32_t pro_id;
        char string3[15];
        char string4[15];
        // putsUart0("\n pidof \n");
        for (i=0;i<MAX_TASKS;i++)
        {
            strcpy(string3,tcb[i].name);
            if ( strcmp(string3,str2)==0)
            {
                ltoa(tcb[i].pid,string4);
                putsUart0("\n pidof\t\t ");
                putsUart0(string3);
                putsUart0("\tis\t\t ");
                putsUart0(string4);
                putsUart0("\n");
            }
        }
    }
}

void process()
{
    uint8_t i=0;
    if(str2[0]==38)
    {
        for(i=0;i<MAX_TASKS;i++)
        {
            if ( strcmp(tcb[i].name,str1)==0)
            {
                if( tcb[i].state==STATE_INVALID)
                {
                    tcb[i].state=STATE_READY;
                }
                else
                {
                    putsUart0("\n");
                    putsUart0("Thread is already active");
                    putsUart0("\n");
                }
            }
        }
    }
}

bool iscommand(char string[], int argCount) // bool will give true if
string is matching with right number of arguments
{
    return ((strcmp(string, &str[fieldOffset[0]]) == 0)
        && (argCount == fieldCount - 1));
}

```

```

void clearStr() // it will clear all the strings before entering new
argument
{
    memset(str, 0, 81);
    fieldCount = 0;
    memset(fieldType, 0, 50);
    memset(fieldOffset, 0, 50);
    // memset(str1,0,20);
    memset(str2, 0, 20);
    memset(str3, 0, 20);
    flag = 0;
}
void shell()
{
    while (true)
    {
        clearStr();
        getString();
        parse();
        if (iscommand("ps", 0))
        {
            ps();
            flag=1;
        }
        if (iscommand("ipcs", 0))
        {
            ipcs();
            flag=1;
        }

        if (iscommand("kill", 1))
        {
            kill();
            flag=1;
        }

        if (iscommand("reboot", 0))
        {
            reboot();
            flag=1;
        }
        if (iscommand("pidof", 1))
        {
            pidof();
            flag=1;
        }
        if (iscommand("idle", 1))
        {
            process();
            flag=1;
        }
    }
}

```

```

    }
    if (iscommand("idle2", 1))
    {
        process();
        flag=1;
    }
    if (iscommand("lengthyfn", 1))
    {
        process();
        flag=1;
    }
    if (iscommand("flash4hz", 1))
    {
        process();
        flag=1;
    }
    if (iscommand("oneshot", 1))
    {
        process();
        flag=1;
    }
    if (iscommand("readkeys", 1))
    {
        process();
        flag=1;
    }
    if (iscommand("debounce", 1))
    {
        process();
        flag=1;
    }
    if (iscommand("important", 1))
    {
        process();
        flag=1;
    }
    if (iscommand("uncoop", 1))
    {
        process();
        flag=1;
    }

    if (flag != 1)
    {
        putsUart0("\r\n Your format is invalid\r\n Please enter valid
Shell command\r\n");
    }

    // REQUIRED: add processing for the shell commands through the
    UART here

```

```

    }

}

//-----
// YOUR UNIQUE CODE
// REQUIRED: add any custom code in this space
//-----

//-----
// Main
//-----

int main(void)
{
    bool ok;

    // Initialize hardware
    initHw();
    rtosInit();

    // Power-up flash
    RED_LED = 1;
    waitMicrosecond(250000);
    RED_LED = 0;
    waitMicrosecond(250000);

    // Initialize semaphores
    keyPressed = createSemaphore(1, "keyPressed");
    keyReleased = createSemaphore(0, "keyReleased");
    flashReq = createSemaphore(5, "flashReq");
    resource = createSemaphore(1, "resource");

    // Add required idle process
    ok = createThread(idle, "idle", 15);
    // ok &= createThread(idle2, "idle2", 3);
    // Add other processes
    ok &= createThread(lengthyFn, "lengthyfn", 12);
    ok &= createThread(flash4Hz, "flash4hz", 4);
    ok &= createThread(oneshot, "oneshot", 4);
    ok &= createThread(readKeys, "readkeys", 8);
    ok &= createThread(debounce, "debounce", 8);
    ok &= createThread(important, "important", 0);
    ok &= createThread(uncooperative, "uncoop", 10);
    ok &= createThread(shell, "shell", 8);

```

```
// Start up RTOS
if (ok)
    rtosStart(); // never returns
else
    RED_LED = 1;

return 0;
// don't delete this unreachable code
// if a function is only called once in your code, it will be
// accessed with two goto instructions instead of call-return,
// so any stack-based code will not function correctly
yield(); sleep(0); wait(0); post(0);
}
```