# EE445M Lab 3 Report

Yen-Kai Huang
Siavash Zanganeh Kamali

March 8, 2014

## 1 Objective

The goal of this lab is to extend the RTOS we built in Lab1 and Lab2 to include blocking and priority scheduling. We will also extend the RTOS to include two real-time periodic tasks, two aperiodic button tasks, and minimally invasive tools to carry out performance measures. The interpreter will be extended to allow recording debugging/performance data and downloading them to the PC screen through UART.

## 2 Software Design

In this lab we implemented priority scheduler, and blocking semaphore. Implementing priority scheduler requires rewriting all the functions that deal with the thread link-list structures (**AddThread**, **wake-up background thread**, **Signal**, **Kill**, **Sleep**, **Block**)

```
/*************** System Performance Measurement *****************/

#ifdef __Performance_Measurement__


  #ifdef __Jitter_Measurement__
  // Measure jitter of periodic tasks
  #define JITTERSIZE 64

  unsigned long Period1;
  long MaxJitter1;             // largest time jitter between interrupts in usec
  long MinJitter1;             // smallest time jitter between interrupts in usec
  unsigned long const JitterSize1=JITTERSIZE;
  unsigned long JitterHistogram1[JITTERSIZE]={0,};

  unsigned long Period2;
  long MaxJitter2;             // largest time jitter between interrupts in usec
  long MinJitter2;             // smallest time jitter between interrupts in usec
  unsigned long const JitterSize2=JITTERSIZE;
  unsigned long JitterHistogram2[JITTERSIZE]={0,};
  #endif

  #ifdef __Critical_Interval_Measurement__

  void DisableInterrupts_check(void);
  void EnableInterrupts_check(void);
```

```
  long StartCritical_check(void);
  void EndCritical_check(long sr);
  #endif

  #ifdef __Profiling__

  #define PROFILELENGTH 100

  #define Profile_PendSV_Trigger 1
  #define Profile_SysTick_Starts 2
  #define Profile_SysTick_End    3
  #define Profile_Timer1_Starts  4
  #define Profile_Timer1_End     5
  #define Profile_Timer2_Starts  6
  #define Profile_Timer2_End     7
  #define Profile_Timer3_Starts  8
  #define Profile_Timer3_End     9
  #define Profile_Timer4_Starts  10
  #define Profile_Timer4_End     11

  typedef struct {
    unsigned char value;
    unsigned long time;
  } TimeStamp;

  TimeStamp Profile[PROFILELENGTH];
  TimeStamp *ProfilePt = Profile;

  void timestamp(unsigned char value);

  #endif
#endif

/*********** Datastructure for OS Round-robin scheduler *********/

#define NUM_PRIORITY 8
#define STACKSIZE_MAX  200
#define STACKSIZE_MIN  64

typedef struct tcb_s {
  long *sp;
  struct tcb_s *next;
  struct tcb_s *prev;
  long *stack;
  unsigned int id;
  unsigned long sleepCount;
  unsigned char priority;
} TCB;

static unsigned long uniqid = 0;

TCB *RunPt, *NextPt, *SleepPt;
TCB *PriPt[NUM_PRIORITY];

static unsigned long Timer=0;
```

```c
/******************************************************************/


void OS_Init(void) {  volatile unsigned long delay;
  OS_DisableInterrupts();
  NVIC_ST_CTRL_R = 0;
  NVIC_ST_CURRENT_R = 0;
  NVIC_SYS_PRI3_R =(NVIC_SYS_PRI3_R&0x00FFFFFF)|0xC0000000;
  NVIC_SYS_PRI3_R =(NVIC_SYS_PRI3_R&0xFF00FFFF)|0x00E00000;

    SYSCTL_RCGC1_R |= SYSCTL_RCGC1_TIMER1;    delay = SYSCTL_RCGC1_R;
  TIMER1_CTL_R &= ~TIMER_CTL_TAEN;
  TIMER1_CFG_R = TIMER_CFG_32_BIT_TIMER;
  TIMER1_TAMR_R = TIMER_TAMR_TAMR_PERIOD;
  TIMER1_TAILR_R = 800000-1;
  TIMER1_ICR_R = TIMER_ICR_TATOCINT;
  TIMER1_IMR_R |= TIMER_IMR_TATOIM;
  NVIC_PRI5_R = (NVIC_PRI5_R&0xFFFF00FF)|0x00008000;
  NVIC_EN0_R = NVIC_EN0_INT21;
  TIMER1_CTL_R |= TIMER_CTL_TAEN;

    SYSCTL_RCGC1_R |= SYSCTL_RCGC1_TIMER2;
  delay = SYSCTL_RCGC1_R;
  TIMER2_CTL_R &= ~TIMER_CTL_TAEN;
  TIMER2_CFG_R = TIMER_CFG_32_BIT_TIMER;
  TIMER2_TAMR_R = TIMER_TAMR_TAMR_PERIOD;
  TIMER2_TAILR_R = 0xFFFFFFFF;
  TIMER2_CTL_R = TIMER_CTL_TAEN;

    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOF;

  GPIO_PORTF_LOCK_R = GPIO_LOCK_KEY;
  GPIO_PORTF_CR_R |= (SW1|SW2);
  GPIO_PORTF_LOCK_R = 0;

  GPIO_PORTF_DIR_R &= ~(SW1|SW2);
  GPIO_PORTF_AFSEL_R &= ~(SW1|SW2);
  GPIO_PORTF_DEN_R |= (SW1|SW2);
  GPIO_PORTF_PCTL_R &= ~0x000F000F;
  GPIO_PORTF_AMSEL_R = 0;
  GPIO_PORTF_PUR_R |= (SW1|SW2);
  GPIO_PORTF_IS_R &= ~(SW1|SW2);
  GPIO_PORTF_IBE_R |= (SW1|SW2);
  GPIO_PORTF_IEV_R &= ~(SW1|SW2);
  GPIO_PORTF_ICR_R = (SW1|SW2);
  GPIO_PORTF_IM_R |= (SW1|SW2);

  Heap_Init();

  OS_AddThread(dummyThread,32,NUM_PRIORITY-1);
}

void OS_InitSemaphore(Sema4Type *semaPt, long value){
  semaPt->Value = value;
  semaPt->BlocketListPt = 0;
}
```

```
void OS_Wait(Sema4Type *semaPt){
  #ifdef __Critical_Interval_Measurement__
  long sr = StartCritical_check();
  #else
  long sr = StartCritical();
  #endif

  semaPt->Value -= 1;

  if(semaPt->Value < 0) {
            if (RunPt->next == RunPt) {
      PriPt[RunPt->priority] = NULL;
    } else {
      DLUNLINK(RunPt);
      PriPt[RunPt->priority] = RunPt->next;
    }

            if(semaPt->BlocketListPt) {
      DLLINKLAST(RunPt, semaPt->BlocketListPt);
    } else {
      RunPt->next = RunPt->prev = RunPt;
      semaPt->BlocketListPt = RunPt;
    }
    updateNextPt(RunPt->priority);
    triggerPendSV();
}

  #ifdef __Critical_Interval_Measurement__
  EndCritical_check(sr);
  #else
  EndCritical(sr);
  #endif
}

void OS_Signal(Sema4Type *semaPt){
  #ifdef __Critical_Interval_Measurement__
  long sr = StartCritical_check();
  #else
  long sr = StartCritical();
  #endif

  TCB *current = semaPt->BlocketListPt;
  semaPt->Value += 1;

  if(semaPt->Value <= 0) {
            if (current->next == current) {
      semaPt->BlocketListPt = NULL;
    } else {
      DLUNLINK(current);
      semaPt->BlocketListPt = current->next;
    }

        if(PriPt[current->priority]) {
      DLLINK(current, PriPt[current->priority]);
    } else {
```

```c
      current->next = current->prev = current;
      PriPt[current->priority] = current;
    }
    updateNextPt(0);
    triggerPendSV();
  }

  #ifdef __Critical_Interval_Measurement__
  EndCritical_check(sr);
  #else
  EndCritical(sr);
  #endif
}

void OS_bWait(Sema4Type *semaPt){
  OS_Wait(semaPt);
}

void OS_bSignal(Sema4Type *semaPt){
  #ifdef __Critical_Interval_Measurement__
  long sr = StartCritical_check();
  #else
  long sr = StartCritical();
  #endif

  TCB *current = semaPt->BlocketListPt;
  if(semaPt->Value < 1) semaPt->Value += 1;
  if(semaPt->Value <= 0) {
            if (current->next == current) {
      semaPt->BlocketListPt = NULL;
    } else {
      DLUNLINK(current);
      semaPt->BlocketListPt = current->next;
    }

        if(PriPt[current->priority]) {
      DLLINK(current, PriPt[current->priority]);
    } else {
      current->next = current->prev = current;
      PriPt[current->priority] = current;
    }
    updateNextPt(0);
    triggerPendSV();
  }

  #ifdef __Critical_Interval_Measurement__
  EndCritical_check(sr);
  #else
  EndCritical(sr);
  #endif
}

int OS_AddThread(void(*task)(void),
                unsigned long stackSize,
                unsigned long priority) {
  TCB *tcb; long sr;
```

```c

  if (priority >= NUM_PRIORITY) priority = NUM_PRIORITY-1;

  stackSize = (stackSize < STACKSIZE_MIN) ? STACKSIZE_MIN : ( (
   stackSize > STACKSIZE_MAX) ? STACKSIZE_MAX : stackSize ) ;

  #ifdef __Critical_Interval_Measurement__
  sr = StartCritical_check();
  #else
  sr = StartCritical();
  #endif



  if (!(tcb = (TCB *) Heap_Malloc(sizeof(TCB)))){
    return NULL;
  };
  if (!(tcb->stack = (long *) Heap_Malloc (sizeof(long)*stackSize))){
    Heap_Free (tcb);
    return NULL;
  };
  SetInitialStack(tcb, stackSize);

    if(PriPt[priority]) {
    DLLINKLAST(tcb, PriPt[priority]);
  } else {
    tcb->next = tcb->prev = tcb;
    PriPt[priority] = tcb;
  }

  tcb->stack[stackSize-2] = (long) (task);
  tcb->sleepCount = 0;
  tcb->priority = priority;
  tcb->id = OS_Id();
  updateNextPt(0);

  if(!RunPt) RunPt = NextPt;

  updateNextPt(0);
  triggerPendSV();
  #ifdef __Critical_Interval_Measurement__
  EndCritical_check(sr);
  #else
  EndCritical(sr);
  #endif

  return 1;
}

unsigned long OS_Id(void) {
  return uniqid++;
}

#define IDLE       0
#define ONE_IN_USE 1
int OS_AddPeriodicThread(void(*task)(void), unsigned long period,
```

```
  unsigned long priority){
  long sr; volatile unsigned long delay;
  static char currentTimer = 0;
  if (priority > 5) {
    priority = 5;
  }

  #ifdef __Critical_Interval_Measurement__
  sr = StartCritical_check();
  #else
  sr = StartCritical();
  #endif

  switch(currentTimer) {
    case IDLE:
      SYSCTL_RCGC1_R |= SYSCTL_RCGC1_TIMER3;

      delay = SYSCTL_RCGC1_R;
      delay = SYSCTL_RCGC1_R;
      PeriodicTask1 = task;
      TIMER3_CTL_R &= ~TIMER_CTL_TAEN;
      TIMER3_CFG_R = TIMER_CFG_32_BIT_TIMER;
      TIMER3_TAMR_R = TIMER_TAMR_TAMR_PERIOD;
      TIMER3_TAILR_R = period-1;
      TIMER3_ICR_R = TIMER_ICR_TATOCINT;
      TIMER3_IMR_R |= TIMER_IMR_TATOIM;
      NVIC_PRI8_R = (NVIC_PRI8_R&0x00FFFFFF)|((priority)<<29);
      NVIC_EN1_R |= NVIC_EN1_INT35;
      TIMER3_CTL_R |= TIMER_CTL_TAEN;
      currentTimer++;

      #ifdef __Jitter_Measurement__
      Period1 = period;
      #endif

      break;
    case ONE_IN_USE:
      SYSCTL_RCGCTIMER_R |= SYSCTL_RCGCTIMER_R4;
      delay = SYSCTL_RCGC1_R;
      delay = SYSCTL_RCGC1_R;
      PeriodicTask2 = task;
      TIMER4_CTL_R &= ~TIMER_CTL_TAEN;
      TIMER4_CFG_R = TIMER_CFG_32_BIT_TIMER;
      TIMER4_TAMR_R = TIMER_TAMR_TAMR_PERIOD;
      TIMER4_TAILR_R = period-1;
      TIMER4_ICR_R = TIMER_ICR_TATOCINT;
      TIMER4_IMR_R |= TIMER_IMR_TATOIM;
      NVIC_PRI17_R = (NVIC_PRI17_R&0xFF00FFFF)|((priority)<<21);
      NVIC_EN2_R |= NVIC_EN2_INT70;
      TIMER4_CTL_R |= TIMER_CTL_TAEN;
      currentTimer++;

      #ifdef __Jitter_Measurement__
      Period2 = period;
      #endif

```

```c
      break;
    default:
      break;
  }

  #ifdef __Critical_Interval_Measurement__
  EndCritical_check(sr);
  #else
  EndCritical(sr);
  #endif

  return 1;
}
#undef IDLE
#undef ONE_IN_USE


#define IDLE        0
#define ONE_IN_USE  1
#define BOTH_IN_USE 2
static unsigned char buttonuse = IDLE;
static unsigned long sw1priority;
static unsigned long sw2priority;

int OS_AddSW1Task(void(*task)(void), unsigned long priority){
  SW1Task = task;
  sw1priority = (priority & 0x07)<<21;

  if(buttonuse == IDLE) {
    NVIC_EN0_R |= NVIC_EN0_INT30;
  }

  NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|sw1priority;

  buttonuse += 1;

  return 1;
}

int OS_AddSW2Task(void(*task)(void), unsigned long priority){
  SW2Task = task;
  sw2priority = (priority & 0x07)<<21;

  if(buttonuse == IDLE) {
    NVIC_EN0_R |= NVIC_EN0_INT30;
  }

  NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|sw2priority;

  buttonuse += 1;

  return 1;
}

int OS_RemoveSW1Task(void) {
  if(SW1Task == dummy) {
```

```
    return 0;
  }

  if(--buttonuse) {     NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|
   sw2priority;   } else {
    NVIC_EN0_R &= ~NVIC_EN0_INT30;    }

  SW1Task = dummy;
  return 1;
}

int OS_RemoveSW2Task(void) {
  if(SW2Task == dummy) {
    return 0;
  }

  if(--buttonuse) {
    NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|sw1priority;
  } else {
    NVIC_EN0_R &= ~NVIC_EN0_INT30;
  }

  SW2Task = dummy;
  return 1;
}

#ifdef DEPRECATE
int OS_AddButtonTask(void(*task)(void), unsigned long priority) {
  OS_AddSW1Task(task, priority);
}
#endif

#undef IDLE
#undef ONE_IN_USE
#undef BOTH_IN_USE

int OS_AddDownTask(void(*task)(void), unsigned long priority);

void OS_Sleep(unsigned long sleepTime){
  TCB *tcb = RunPt;

  #ifdef __Critical_Interval_Measurement__
  DisableInterrupts_check();
  #else
  DisableInterrupts();
  #endif

    if (RunPt->next == RunPt) {
    PriPt[RunPt->priority] = NULL;
    updateNextPt(RunPt->priority);
  } else {
    DLUNLINK(RunPt);
    PriPt[RunPt->priority] = RunPt->next;
    NextPt = RunPt->next;
  }

```

```c
    if(SleepPt) {
    DLLINK(tcb, SleepPt);
  } else {
    tcb->next = tcb->prev = tcb;
    SleepPt = tcb;
  }
  tcb->sleepCount = sleepTime;

  triggerPendSV();

  #ifdef __Critical_Interval_Measurement__
  EnableInterrupts_check();
  #else
  EnableInterrupts();
  #endif
}

void OS_Kill(void){

  #ifdef __Critical_Interval_Measurement__
  DisableInterrupts_check();
  #else
  DisableInterrupts();
  #endif


  Heap_Free(RunPt->stack);
  if (RunPt->next == RunPt) {
    PriPt[RunPt->priority] = NULL;
  } else {
    DLUNLINK(RunPt);
    PriPt[RunPt->priority] = RunPt->next;
  }
  Heap_Free(RunPt);
  updateNextPt(0);
  triggerPendSV();

  #ifdef __Critical_Interval_Measurement__
  EnableInterrupts_check();
  #else
  EnableInterrupts();
  #endif

}

void OS_Suspend(void){
  NVIC_ST_CURRENT_R = 0;        NVIC_INT_CTRL_R |=
   NVIC_INT_CTRL_PENDSTSET; }

void OS_Fifo_Init(unsigned long size) {
  OSFifo_Init();
  OS_InitSemaphore(&Sema4fifo, 0);
}

int OS_Fifo_Put(unsigned long data) {
  if(OSFifo_Put(data) == FIFOSUCCESS) {
```

```c
    OS_Signal(&Sema4fifo);
    return FIFOSUCCESS;
  } else {
    return FIFOFAIL;
  }
}

unsigned long OS_Fifo_Get(void) {
  unsigned long data;
  OS_Wait(&Sema4fifo);
  OSFifo_Get(&data);
  return data;
}

long OS_Fifo_Size(void) {
  return OSFifo_Size();
}

void OS_MailBox_Init(void) {
  OS_InitSemaphore(&Sema4MailboxEmpty, 1);
  OS_InitSemaphore(&Sema4MailboxFull, 0);
}

void OS_MailBox_Send(unsigned long data) {
  OS_bSignal(&Sema4MailboxFull);
  OS_bWait(&Sema4MailboxEmpty);
  Mailbox = data;
}

unsigned long OS_MailBox_Recv(void) {
  OS_bSignal(&Sema4MailboxEmpty);
  OS_bWait(&Sema4MailboxFull);
  return Mailbox;
}

unsigned long OS_Time(void){
  return TIMER2_TAV_R;
}

unsigned long OS_TimeDifference(unsigned long stop, unsigned long start
    ){
  return (start-stop);
}

void OS_ClearMsTime(void){
  Timer = 0;
}

unsigned long OS_MsTime(void){
  return Timer;
}

void OS_Launch(unsigned long theTimeSlice){
  NVIC_ST_RELOAD_R = theTimeSlice - 1;   NVIC_ST_CTRL_R = 0x00000007;
    StartOS();                          }

```

```c

void SysTick_Handler(void) { long sr;
  #ifdef __Critical_Interval_Measurement__
  sr = StartCritical_check();
  #else
  sr = StartCritical();
  #endif

  #ifdef __Profiling__
  timestamp(Profile_SysTick_Starts);
  #endif

  NextPt = RunPt->next;
  NVIC_INT_CTRL_R |= NVIC_INT_CTRL_PEND_SV;
  #ifdef __Profiling__
  timestamp(Profile_PendSV_Trigger);
  #endif

  #ifdef __Profiling__
  timestamp(Profile_SysTick_End);
  #endif

  #ifdef __Critical_Interval_Measurement__
  EndCritical_check(sr);
  #else
  EndCritical(sr);
  #endif


}

static void triggerPendSV (void) {
  #ifdef __Profiling__
  timestamp(Profile_PendSV_Trigger);
  #endif

  NVIC_ST_CURRENT_R = 0;
  NVIC_INT_CTRL_R |= NVIC_INT_CTRL_PEND_SV;
}

static void updateNextPt (int startLevel) {
  int i;
  for (i=startLevel; i<NUM_PRIORITY; i++) {
    if (PriPt[i]){
      NextPt = PriPt[i];
      break;
    }
  }
}


void Timer1A_Handler(void){
  TCB *current;
  int minPriority = NUM_PRIORITY;

  #ifdef __Profiling__
```

```c
  timestamp(Profile_Timer1_Starts);
  #endif

  TIMER1_ICR_R = TIMER_ICR_TATOCINT;
  ++Timer;

    if(!SleepPt) {
    #ifdef __Profiling__
    timestamp(Profile_Timer1_End);
    #endif

    return;
  }
  current = SleepPt;
  do {
    TCB *currentNext = current->next;
    if(!(--(current->sleepCount))) {
      minPriority = (current->priority < minPriority) ? (current->
   priority) : minPriority;
      if (current->next == current) {
        SleepPt = NULL;
                if(PriPt[current->priority]) {
          DLLINK(current, PriPt[current->priority]);
        } else {
          current->next = current->prev = current;
          PriPt[current->priority] = current;
        }
        break;
      } else {
        DLUNLINK (current);
        SleepPt = current->next;
                if(PriPt[current->priority]) {
          DLLINK(current, PriPt[current->priority]);
        } else {
          current->next = current->prev = current;
          PriPt[current->priority] = current;
        }
      }
    }
    current = currentNext;
  } while (current != SleepPt);
  if (minPriority < RunPt->priority ) {
    updateNextPt(0);
    triggerPendSV();
  }

  #ifdef __Profiling__
  timestamp(Profile_Timer1_End);
  #endif
}




#ifdef __Jitter_Measurement__
unsigned static long LastTime1;  #define IDLE    0
```

```
#define STARTED 1
unsigned char MeasureState1 = IDLE;
#endif

void Timer3A_Handler(void){
#ifdef __Jitter_Measurement__
  long jitter;
  int index;
  unsigned long thisTime = OS_Time();
#endif

  #ifdef __Profiling__
  timestamp(Profile_Timer3_Starts);
  #endif

  TIMER3_ICR_R = TIMER_ICR_TATOCINT;

  (*PeriodicTask1)();

  #ifdef __Jitter_Measurement__
    if(MeasureState1 == STARTED) {
    jitter = (long)(OS_TimeDifference(thisTime,LastTime1)/80)-(long)(
    Period1/80);      MaxJitter1 = (jitter > MaxJitter1)? jitter:
    MaxJitter1;
    MinJitter1 = (jitter < MinJitter1)? jitter: MinJitter1;
    index = jitter+JITTERSIZE/2;       if(index < 0) index = 0;
    if(index>=JitterSize1) index = JITTERSIZE-1;
    JitterHistogram1[index]++;

  } else {     MeasureState1 = STARTED;
  }
  LastTime1 = thisTime;
  #endif

  #ifdef __Profiling__
  timestamp(Profile_Timer3_End);
  #endif
}

#ifdef __Jitter_Measurement__
unsigned static long LastTime2;
unsigned char MeasureState2 = IDLE;
#endif

void Timer4A_Handler(void){
#ifdef __Jitter_Measurement__
  long jitter;
  int index;
  unsigned long thisTime = OS_Time();
#endif

  #ifdef __Profiling__
  timestamp(Profile_Timer4_Starts);
  #endif

  TIMER4_ICR_R = TIMER_ICR_TATOCINT;
```

```c

  (*PeriodicTask2)();

#ifdef __Jitter_Measurement__
    if(MeasureState2 == STARTED) {
    jitter = (long)(OS_TimeDifference(thisTime,LastTime2)/80)-(long)(
   Period2/80);      MaxJitter2 = (jitter > MaxJitter2)? jitter:
   MaxJitter2;
    MinJitter2 = (jitter < MinJitter2)? jitter: MinJitter2;
    index = jitter+JITTERSIZE/2;       if(index < 0) index = 0;
    if(index>=JitterSize2) index = JITTERSIZE-1;
    JitterHistogram2[index]++;
  } else {     MeasureState2 = STARTED;
  }
  LastTime2 = thisTime;
#endif

  #ifdef __Profiling__
  timestamp(Profile_Timer4_End);
  #endif
}

#ifdef __Jitter_Measurement__
void Jitter(void) {
  printf("Timer1 jitter = %ld\r\nTimer2 jitter = %ld\r\n", MaxJitter1 -
    MinJitter1, MaxJitter2 - MinJitter2);
}
#endif

#undef IDLE
#undef STARTED

void static DebounceTask1(void){
  OS_Sleep(1);
  LastPF4 = PF4;
  GPIO_PORTF_ICR_R = SW1;
  GPIO_PORTF_IM_R |= SW1;

  OS_Kill();
}

void static DebounceTask2(void){
  OS_Sleep(1);
  LastPF0 = PF0;
  GPIO_PORTF_ICR_R = SW2;
  GPIO_PORTF_IM_R |= SW2;

  OS_Kill();
}

void GPIOPortF_Handler(void){
  unsigned long vector = GPIO_PORTF_MIS_R;
  if(vector & SW1) {
    if(LastPF4){
      (*SW1Task)();
    }
```

```
    OS_AddThread(&DebounceTask1, 32, 0);
    GPIO_PORTF_IM_R &= ~SW1;

  } else {
    if(LastPF0){
      (*SW2Task)();
    }

    OS_AddThread(&DebounceTask2, 32, 0);
    GPIO_PORTF_IM_R &= ~SW2;
  }
}

/************ Performance Measurement *************/
#ifdef __Critical_Interval_Measurement__

unsigned long MaxCriticalInterval;
unsigned long TotalCriticalInterval;
unsigned long StartCriticalTime;

unsigned int CS_Duration;

long StartCritical_check(void) {
  long sr = StartCritical();

  StartCriticalTime = OS_Time();

  return sr;
}

void EndCritical_check(long sr) {
  unsigned long endCriticalTime = OS_Time();
  long criticalInterval = (long)(OS_TimeDifference(endCriticalTime,
   StartCriticalTime)/80);
  TotalCriticalInterval += criticalInterval;
  MaxCriticalInterval = (criticalInterval > MaxCriticalInterval)?
   criticalInterval : MaxCriticalInterval;

  EndCritical(sr);
}

void EnableInterrupts_check(void) {
  EnableInterrupts();
  StartCriticalTime = OS_Time();
}

void DisableInterrupts_check(void) {
  unsigned long endCriticalTime = OS_Time();
  long criticalInterval = (long)(OS_TimeDifference(endCriticalTime,
   StartCriticalTime)/80);
  TotalCriticalInterval += criticalInterval;
  MaxCriticalInterval = (criticalInterval > MaxCriticalInterval)?
   criticalInterval : MaxCriticalInterval;

  DisableInterrupts();
```

```

}

Sema4Type Sema4CriticalIntervalReady;

static void CriticalIntervalThread(void) {
  OS_Sleep(CS_Duration);    OS_Signal(&Sema4CriticalIntervalReady);
  OS_Kill();
  while(1);
}

void OS_CriticalInterval_Start(int duration) {
  CS_Duration = duration;
  StartCriticalTime = MaxCriticalInterval = TotalCriticalInterval = 0;

  OS_InitSemaphore(&Sema4CriticalIntervalReady, 0);
  OS_AddThread(CriticalIntervalThread, 32, 0);

  OS_Wait(&Sema4CriticalIntervalReady);
  printf("Maximum time in critical secsion = %ld\r\n" \
         "Percentage of time in critical section = %ld/%d = %ld.%21d%%\r\n", \
         MaxCriticalInterval, TotalCriticalInterval, duration*10000,
         TotalCriticalInterval/(duration*100), TotalCriticalInterval%(duration*100));
}

#endif

#ifdef __Profiling__

Sema4Type Sema4ProfilingDone;

unsigned char Profiling;
unsigned long ProfileStartTime;

void timestamp(unsigned char value) {
  if(!Profiling) return;

  if(ProfilePt >= &Profile[PROFILELENGTH]) {
    Profiling = 0;
    OS_Signal(&Sema4ProfilingDone);

    return;
  }

  ProfilePt->value = value;
  ProfilePt->time = ProfileStartTime - OS_Time();
  ProfilePt++;


}

void Profile_Dump(void) {
  TimeStamp *pt;

```

```c
  for(pt = Profile; pt < ProfilePt; ++pt) {
    printf("[%07lu] Thread-%d\r\n", pt->time, pt->value);
  }
}

void OS_Profile_Start(void) {
  Profiling = 1;
  ProfilePt = Profile;
  ProfileStartTime = OS_Time();

  OS_InitSemaphore(&Sema4ProfilingDone, 0);
  OS_Wait(&Sema4ProfilingDone);

  Profile_Dump();
}

#endif
```

Code/OS.c

The change does not involve osasm.c

```
;/* OSasm.s: low-level OS commands, written in assembly */
;// Real Time Operating System

; This example accompanies the book
;  "Embedded Systems: Real Time Interfacing to the Arm Cortex M3",
;  ISBN: 978-1463590154, Jonathan Valvano, copyright (c) 2011
;
;  Programs 6.4 through 6.12, section 6.2
;
;Copyright 2011 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/
; */

        AREA |.text|, CODE, READONLY, ALIGN=2
        THUMB
        REQUIRE8
        PRESERVE8

        EXTERN  RunPt            ; currently running thread
    EXTERN  NextPt
        EXPORT  OS_DisableInterrupts
        EXPORT  OS_EnableInterrupts
        EXPORT  StartOS

```

```
        EXPORT  PendSV_Handler

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, =NextPt
    LDR     R1, [R1]            ; 6) R1 = NextPt
    STR     R1, [R0]            ;    RunPt = NextPt
    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
```

Code/osasm.s

Then we test the blocking semaphore with this program.

```
Sema4Type Readyc;        // set in background
int Lost;
void BackgroundThread1c(void){
  Count1++;
  OS_Signal(&Readyc);
}
void Thread5c(void){
  for(;;){
    OS_Wait(&Readyc);
    Count5++;   // Count2 + Count5 should equal Count1
    Lost = Count1-Count5-Count2;
  }
}
void Thread2c(void){
  OS_InitSemaphore(&Readyc,0);
  Count1 = 0;    // number of times signal is called
  Count2 = 0;
  Count5 = 0;    // Count2 + Count5 should equal Count1
```

```c
  NumCreated += OS_AddThread(&Thread5c,128,3);
  OS_AddPeriodicThread(&BackgroundThread1c,TIME_1MS,0);
  for(;;){
    OS_Wait(&Readyc);
    Count2++;   // Count2 + Count5 should equal Count1
  }
}

int Testmain3(void){
  Count4 = 0;
  OS_Init();

  NumCreated = 0 ;
  OS_AddSW1Task(&BackgroundThread5c,2);
  NumCreated += OS_AddThread(&Thread2c,128,2);
  NumCreated += OS_AddThread(&Thread3c,128,3);
  NumCreated += OS_AddThread(&Thread4c,128,3);
  OS_Launch(TIME_2MS);
  return 0;
}
```

Code/Testmain3_blocking.c

We also measured the jitter with this program.

```c
extern void Jitter(void);   // prints jitter information (write this)
void Thread7(void){
  OS_bWait(&Sema4UART);
    UART_OutString("\n\rEE345M/EE380L, Lab 3 Preparation 2\n\r");
  OS_bSignal(&Sema4UART);

  OS_Sleep(1000);   // 10 seconds

  OS_bWait(&Sema4UART);
    Jitter();    // print jitter information
    UART_OutString("\n\r\n\r");
  OS_bSignal(&Sema4UART);

  OS_Kill();
}

#define workA 500       // {5,50,500 us} work in Task A
#define counts1us 80    // number of OS_Time counts per 1us
void TaskA(void){
  PE1 = 0x02;
  CountA++;
  PseudoWork(workA*counts1us); //  do work (100ns time resolution)
  PE1 = 0x00;
}
#define workB 250       // 250 us work in Task B
void TaskB(void){
  PE2 = 0x04;
  CountB++;
  PseudoWork(workB*counts1us); //  do work (100ns time resolution)
  PE2 = 0x00;
}

```

```c
int main(void) {
  PLL_Init();
  OS_InitSemaphore(&Sema4UART, 1);

  Debug_PortE_Init();
  OS_Init();
  NumCreated = 0 ;
  NumCreated += OS_AddThread(&Interpreter,128,1);
  NumCreated += OS_AddThread(&Thread6,128,2);
  NumCreated += OS_AddThread(&Thread7,128,1);

  OS_AddPeriodicThread(&TaskA,TIME_1MS,0);
  OS_AddPeriodicThread(&TaskB,2*TIME_1MS,1);

  OS_Launch(TIME_2MS);
  return 0;
}
```

Code/Jitter_measurement.c

# 3 Measurement

**(a) Plot of the logic analyzerrunning the blocking/sleeping/killing/round-robin system**

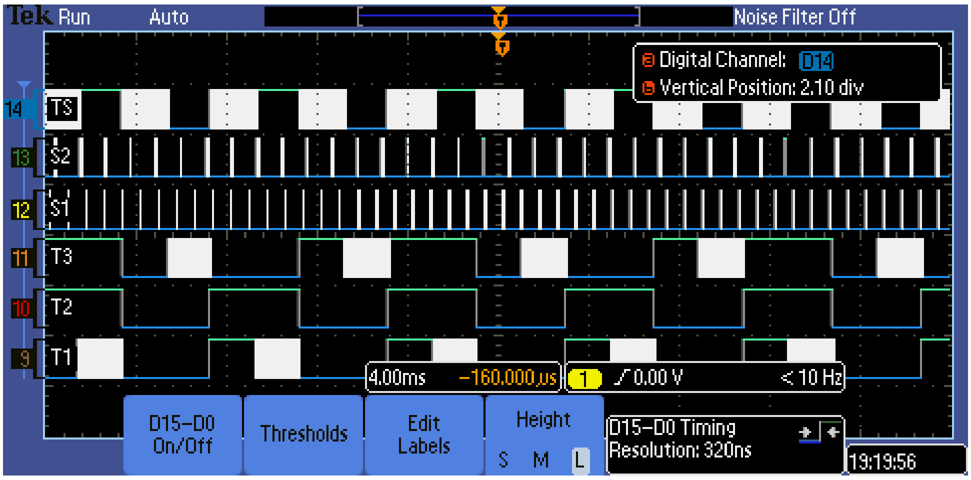
We imitate a round-robin system by setting all the priorities of threads (T1, T2, T3) to an equal value. Background thread S1 and S2 are signalling until a counter counts to a threshold value. And foreground thread ST signals until it kills itself. See Figure-1.

**(b) Plot of the scope window running the blocking/sleeping/killing/priority system**

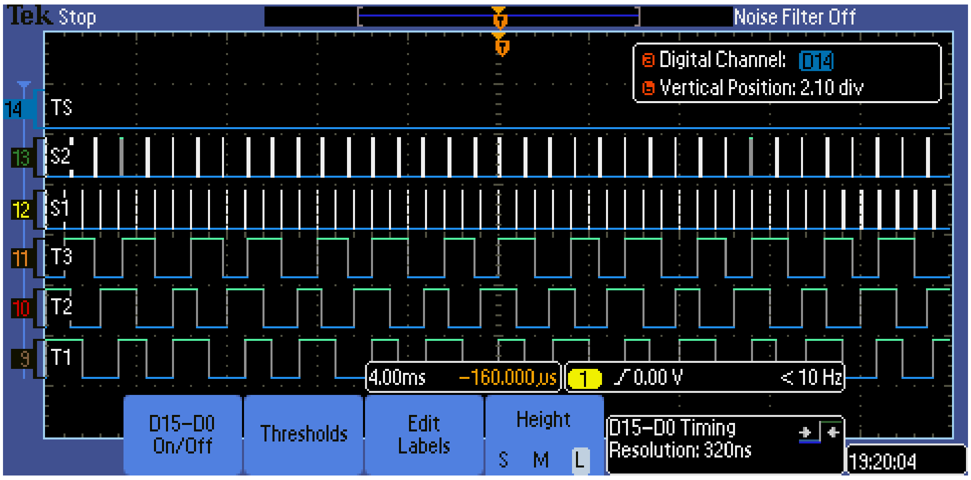
Figure-2 shows the scope picture of running a priority scheduler on the Producer-Consumer main. The labels show the name of the threads. Among those, DAS is a periodic background thread, all the other threads are foreground threads. Button is an aperiodic thread caused by pressing the switch. The Interpreter thread is a foreground thread that waits on user input, therefore also aperiodic.

**(c) Table like Table 3.1 each showing performance measurements versus sizes of the Fifo and timeslices.**

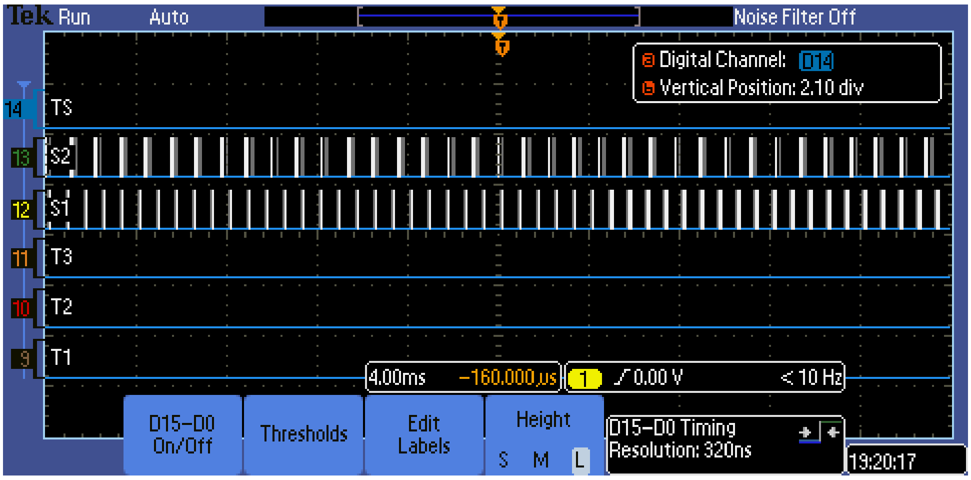
See Table-1.




(a) Initial execution

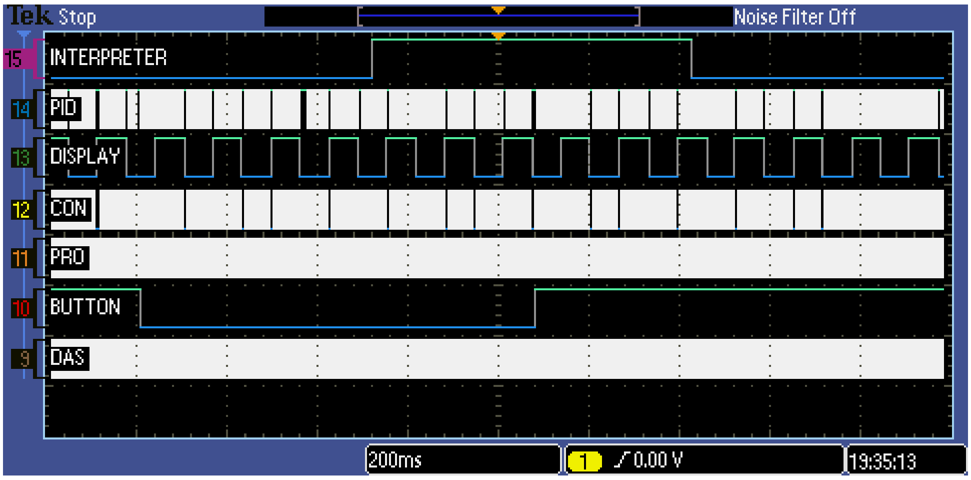



(b) After ST kills itself

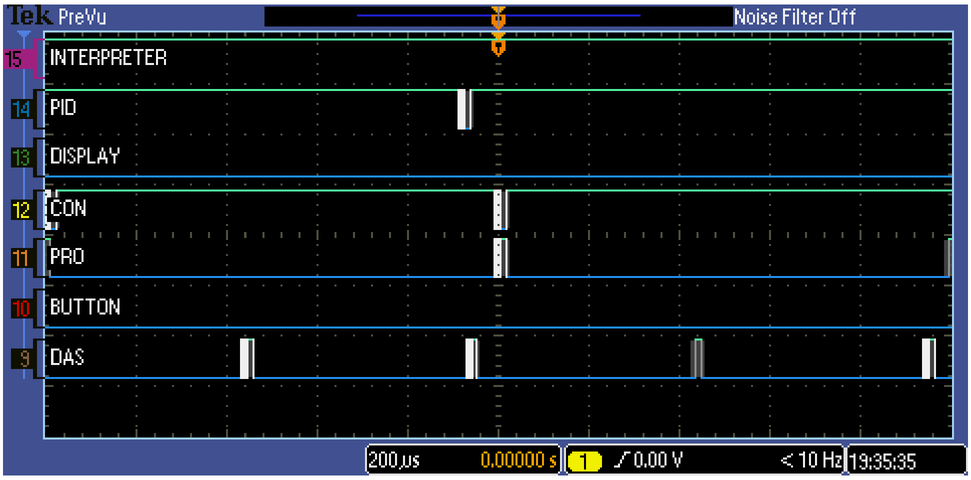



(c) After the signalling stops

Figure 1: Scope showing Round-robin scheduler running




(a) Initial execution




(b) Zoomed-in picture

Figure 2: Scope showing Priority scheduler running

| Performance Measurement | | | | |
|---|---|---|---|---|
| Spinlock/Non-Cooperative | | | | |
| FIFOSize | TIMESLICE(ms) | DataLost | Jitter($\mu s$) | PIDWork |
| 4 | 2 | 1895 | 9 | 1842 |
| 8 | 2 | 0 | 9 | 1743 |
| 32 | 2 | 0 | 9 | 1743 |
| 32 | 1 | 0 | 10 | 1745 |
| 32 | 10 | 0 | 9 | 1778 |
| Spinlock/Cooperative | | | | |
| FIFOSize | TIMESLICE(ms) | DataLost | Jitter($\mu s$) | PIDWork |
| 4 | 2 | 1916 | 10 | 2656 |
| 8 | 2 | 0 | 9 | 2507 |
| 32 | 2 | 0 | 10 | 2507 |
| 32 | 1 | 0 | 10 | 2507 |
| 32 | 10 | 0 | 9 | 2510 |
| Block/Priority | | | | |
| FIFOSize | TIMESLICE(ms) | DataLost | Jitter($\mu s$) | PIDWork |
| 4 | 2 | 1473 | 1 | 5388 |
| 8 | 2 | 986 | 1 | 5316 |
| 32 | 2 | 0 | 1 | 5153 |
| 32 | 1 | 0 | 1 | 5153 |
| 32 | 10 | 0 | 1 | 5153 |

**Table 1. Performance Measurements**

# 4 Analysis and Discussion

**How would your implementation of OS_AddPeriodicThread be different if there were 10 background threads? (Preparation 1)**

Instead of using different timers for each background thread, use one background thread that runs all the other threads. For each background thread, there should be a data structure that holds a pointer to the background function, the period of the thread, and the current time remaining to next execution. (In units of the period of the timer interrupt) The interrupt will pass through each thread, decrements their counter, execute if the counter is 0, and then sets their timer to the reload value again. This can work for any number of periodic tasks.

**How would your implementation of blocking semaphores be different if there were 100 foreground threads? (Preparation 4)**

Since we are using linking and unlinking, our implementation is functional if there were 100 foreground threads. The scheduler is as fast as a scheduler without blocking semaphores. The only limitation will be the heap space in our dynamic memory allocation module.

**How would your implementation of the priority scheduler be different if there were 100 foreground threads? (Preparation 5)**

Since we're using different linked list for every priority, our implementation is compatible with 100 foreground threads. However, if the number of priority levels increases, a better method of searching should be implemented rather than a simple linear search to find the available highest priority.

**What happens to your OS if all the threads are blocked? If your OS would crash, describe exactly what the OS does? What happens to your OS if all the threads are sleeping? If your OS would crash, describe exactly what the OS does? If you answered crash to either or both, explain how you might change your OS to prevent the crash.**

If last active thread blocks, the scheduler assigns the last thread to be blocked as the next thread to be run, and therefore the OS continues running the foreground thread that was supposed to be blocked. This is an unexpected behaviour and will probably cause a crash in the system.

If last active thread sleeps, the scheduler assigns the last thread to be slept as the next thread to be run, and therefore the OS continues running the foreground threads that were supposed to be asleep. This can cause undefined behaviour and eventually crash the system.

Currently, our OS_Init always adds a dummy thread at the lowest priority that will always remain active which prevents crashes. a more robust method will be to make the microcontroller sleeps in case of no active foreground thread and waits for interrupts to occur.

**What happens to your OS if one of the foreground threads returns? e.g., what if you added this foreground**

```
void BadThread(void){ int i;
  for(i=0; i<100; i++){};
  return;
}
```

**What should your OS have done in this case? Do not implement it, rather, with one sentence, say what the OS should have done? Hint: I asked this question on an exam.**

OS should have detected that a foreground thread has tried to return and therefore killed the faulty thread.

**What are the advantages of spinlock semaphores over blocking semaphores? What are the advantages of blocking semaphores over spinlock?**

Spinlock semaphores are easier to implement and do not slow down the process of choosing next thread in the scheduler. However, blocking semaphores can be a lot faster, more CPU efficient, and can provide the means to implement bounded waiting.

**Consider the case where thread $T_1$ interrupts thread $T_2$, and we are experimentally verifying the system operates without critical sections. Let $n$ be the number of times $T_1$ interrupts $T_2$. Let $m$ be the total number of interruptible locations within $T_2$. Assume the time during which $T_1$ triggers is random with respect to the place (between which two instructions of $T_2$) it gets interrupted. In other words, there are $m$ equally-likely places within $T_2$ for the $T_1$ interrupt to occur. What is the probability after $n$ interrupts that a particular place in $T_2$ was never selected? Furthermore, what is the probability that all locations were interrupted at least once?**

Probability that a line was never selected is $(m-1/m)^n$

Probability that all locations were interrupted once

$$p = \begin{cases} 0 & \text{if } n < m \\ \left(1 - \left(\frac{m-1}{m}\right)^n\right)^m & \text{if } n \geq m \end{cases}$$