Sam Caldwell

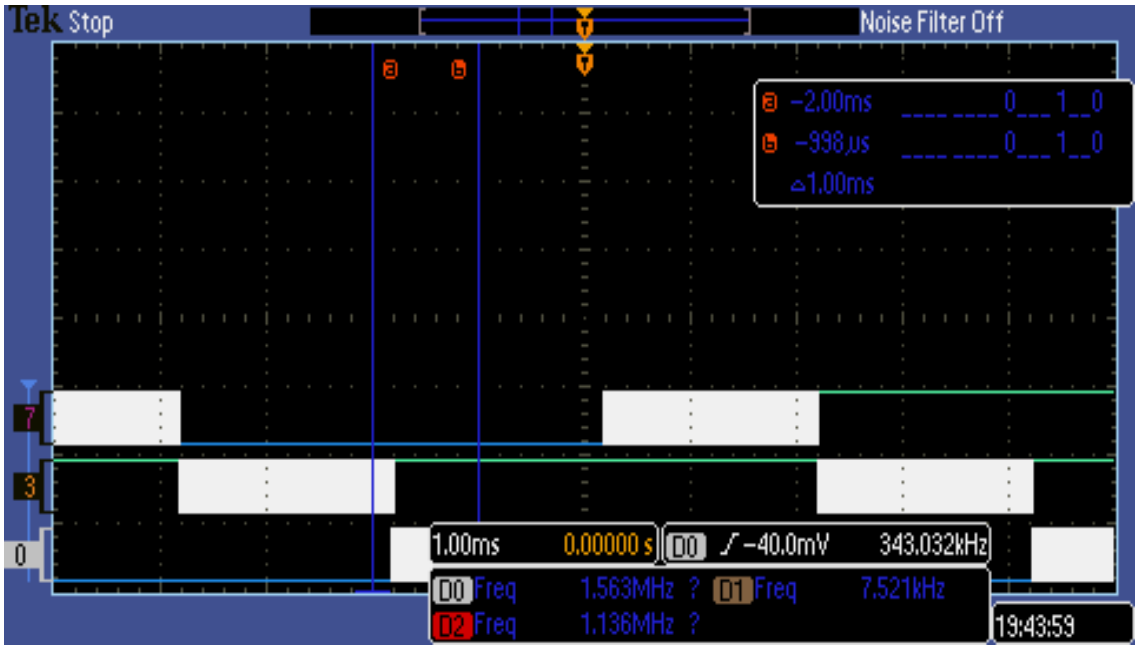
Rohith Prakash

Lab 2 Report

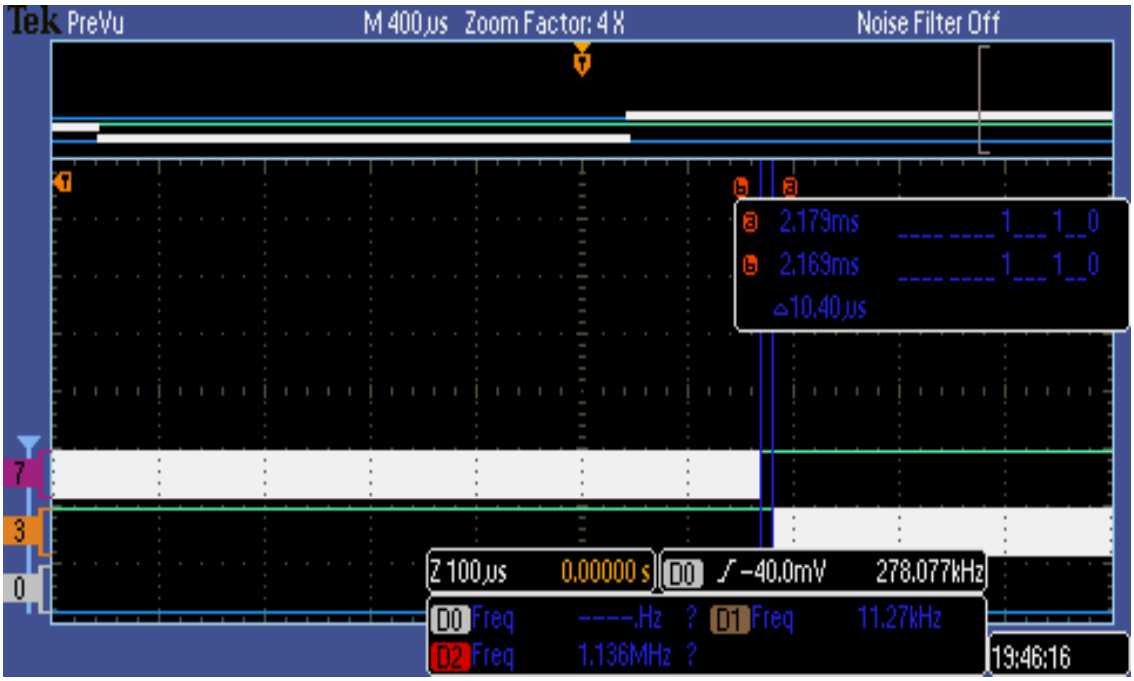
1. **Objectives**

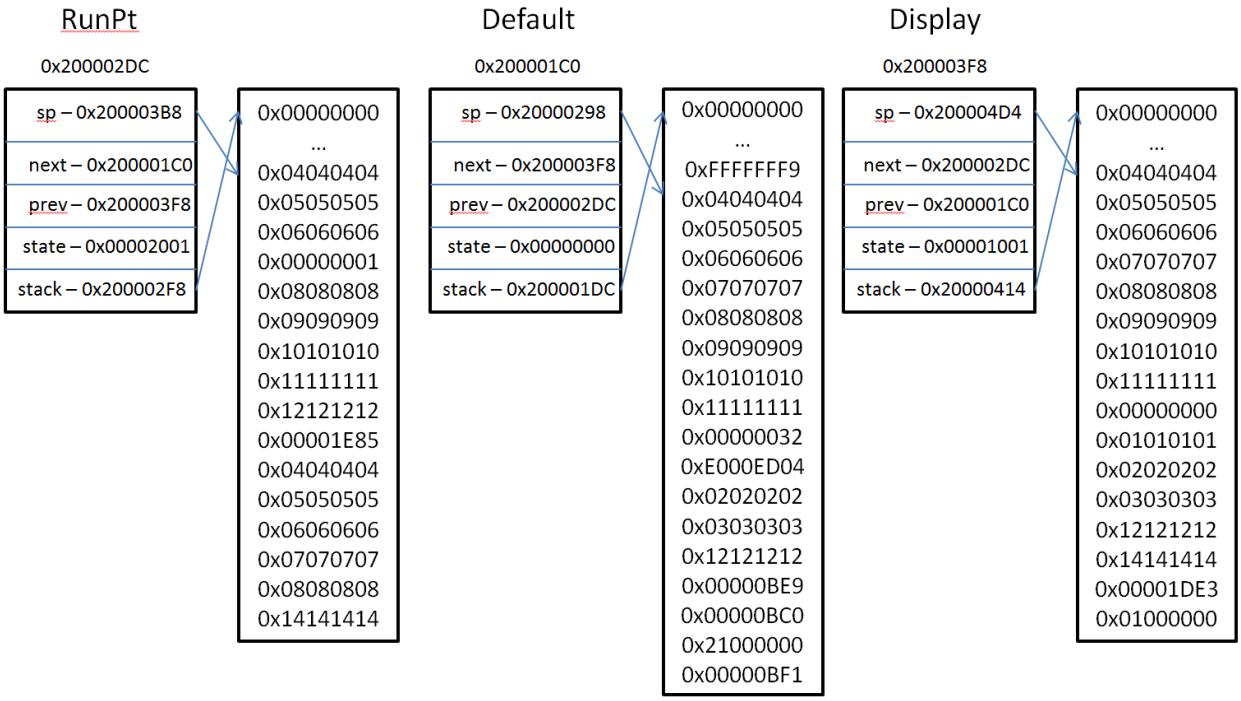
   The objective of this lab was to design a system which is capable of managing multiple foreground and background threads simultaneously, keeping each thread's stack space and state separate. Additionally, this system must implement binary and counting semaphores as well as provide a method for foreground threads to sleep, kill, and suspend themselves. Lastly, this system must be able to provide a way for threads to communicate with each other. A fifo queue which supports a fixed number of elements and a mailbox which supports holding 1 element must be implemented for the purpose of this communication.

2. **Software Design**



3. Measurement Data





Task	With profiling	No profiling	W/ button push
PIDWork (10s)	1432	1592	1519

TIMESLICE (ms)	Data lost	Jitter (us)	PIDWork
2	0	22	1004

1	0	22	1287
10	0	22	1262

4. Analysis and Discussion

1) If the UART is interrupt driven, than unpredictable high-priority interrupts will increase the jitter.
2) ADC data is collected via a hardware interrupt. Even if interrupts are disabled when the interrupt is ready, the data will still be stored into the register because int only disabled software interrupts and handlers.
3) DAS: priority 1; ADC: priority 3; Button: priority 2; SysTick: priority 7. The reason behind these priorities is to give the real-time tasks the highest priorities to minimize jitter.
4) If the stack size is too small, then if too large of a variable is allocated, or too many function calls are nested, the processor will try to write into a memory location that is not allocated specifically for the stack. To detect stack over/underflows, we can use the Memory Protection Unit. The Arm Cortex M3 includes an MPU which can specify regions in memory as protected and can be set to cause a fault on access these protected regions. In the fault handler, we can then kill the foreground thread which caused this fault.
5) The OS_Kill() statements at the end of Consumer and Display should always be executed, as once the ADC has sampled enough data via Producer they should exit their while() loops and then execute OS_Kill(). However, based on observation, this doesn't always happen.
6) *Deterministic* means that for a fixed input, the output is always the same. In this case, the context switch happens after 6 elements are put into the fifo. If the fifo has data, the consumer will get the data, and attempt to put it into thte mailbox. Data can also be lost if the mailbox is full and the consumer can’t get and send any more data.
7) Based on the length of the timeslice, it is unlikely that Consumer ever waits on OS_MailBoxSend, as it has to wait for 64 samples from the ADC and then compute the FFT from them in between calls. As long as a thread switch happens during that time, allowing Display to empty the mailbox, it won't have to wait.

Code:

```
void OS_InitSemaphore(OS_SemaphoreType *s, int permits) {
  s->value = permits;
}

int OS_AddThread(void(*task)(void), unsigned long stackSize, unsigned long priority) {
  OS_CRITICAL_FUNCTION;
  _TCB* thread;// = (_TCB*) malloc(sizeof(_TCB));
  int i, tid = _OS_numThreads++;
  // lock thread linked list
  OS_bWait(&_modifyTCB);
  OS_ENTER_CRITICAL();
  if(_RunPt == NULL) {
    // this is the first thread
    thread = &_threads[0];
    _OS_InitThread(thread, task, thread, thread, tid, 0, 0, priority) // TODO: deide how to handle stacks
    _RunPt = thread;
  }
  else {
    // need to find the first open space in thread storage array
    for(i = 0; i < _OS_MAX_THREADS; i++) {
      if(_threads[i].id == _OS_FREE_THREAD) {
        break;
      }
    }
    if(i == _OS_MAX_THREADS) {
      // maximum number of threads already used
      // TODO - handle this more elgeantly
      OS_bSignal(&_modifyTCB);
      return 0;
    }
    else {
      _TCB* temp = _RunPt;  // store _RunPt in case thread switcher changes it while this is happening
      thread = &_threads[i];
      // add thread between RunPt.previous and RunPt, at the end of the list
      _OS_InitThread(thread, task, temp, temp->prev, tid, 0, 0, priority)
      temp->prev->next = thread;
      temp->prev = thread;
    }
  }
  // unlock thread linked list
  OS_EXIT_CRITICAL();
  OS_bSignal(&_modifyTCB);
  return 1;
}

void OS_Init(void) {
```

```
  int i;
          DisableInterrupts();
  SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_XTAL_8MHZ | SYSCTL_OSC_MAIN);
  /* Initialize SysTick */
          NVIC_ST_CTRL_R = 0;        // disable SysTick during setup
  NVIC_ST_CURRENT_R = 0;      // any write to current clears it
  NVIC_SYS_PRI3_R =(NVIC_SYS_PRI3_R&0x00FFFFFF)|0xE0000000; // priority 7
          /* Initialize PendSV */
          NVIC_INT_CTRL_R = 0;                                  // disable PendSV during setup, may not be
necessary, or my be vital ?!?!
          NVIC_SYS_PRI3_R =(NVIC_SYS_PRI3_R&0xFF00FFFF)|0x00E00000; // lowest priority
          /* Initialize foreground linked list */
  OS_InitSemaphore(&_modifyTCB, OS_BINARY_SEMAPHORE);
  _RunPt = NULL;
  // intitalize array of threads to indicate they are all free
  for(i = 0; i < _OS_MAX_THREADS; i++) {
    _threads[i].id = _OS_FREE_THREAD;
  }
  _OS_numThreads = 0;

  // initialize timers used by OS
  Timer2A_Init();
  Timer2B_Init(2);

  /* Add default thread in case all threads killed */
  OS_AddThread(&_OS_Default_Thread, 0, 0); // should be lowest priority
}

// return the id of the thread pointed to by _RunPt
// if _RunPt is null, return -1
int OS_Id(void) {
  if(_RunPt == NULL) {
    return -1;
  }
  return _RunPt->id;
}

// redefine for debugging that needs access to private variables
void OS_Debug(void) {
  UART_OutString("threads:\r\n");
  _OS_PrintThreads(3);
}

// print information from n threads, starting with RunPt and cycling if necessary
static void _OS_PrintThreads(int n) {
  int i;
  _TCB *temp;
  if (_RunPt != NULL) {
```

```
    temp = _RunPt;
    for(i = 0; i < n; i++) {
      _OS_PrintThread(temp);
      temp = temp->next;
    }
  }
  else {
    UART_OutString("No threads"); UART_OutString(SH_NL);
  }
}

static void _OS_PrintThread(_TCB* thread) {
  UART_OutString("this = "); UART_OutUDec(thread->id); UART_OutString(SH_NL);
  UART_OutString("next = "); UART_OutUDec(thread->next->id); UART_OutString(SH_NL);
  UART_OutString("prev = "); UART_OutUDec(thread->prev->id); UART_OutString(SH_NL);
  UART_OutString("priority = "); UART_OutUDec(thread->priority); UART_OutString(SH_NL);
}

static void _OS_SetInitialStack(_TCB* thread, void(*task)(void)){
//  thread->stack = (unsigned long*) malloc(_OS_STACK_SIZE * sizeof(unsigned long));
  thread->sp = &thread->stack[_OS_STACK_SIZE-16]; // thread stack pointer
  thread->stack[_OS_STACK_SIZE-1] = 0x01000000;   // thumb bit
  thread->stack[_OS_STACK_SIZE-2] = (unsigned long) task; // initial pc
  thread->stack[_OS_STACK_SIZE-3] = 0x14141414;   // R14
  thread->stack[_OS_STACK_SIZE-4] = 0x12121212;   // R12
  thread->stack[_OS_STACK_SIZE-5] = 0x03030303;   // R3
  thread->stack[_OS_STACK_SIZE-6] = 0x02020202;   // R2
  thread->stack[_OS_STACK_SIZE-7] = 0x01010101;   // R1
  thread->stack[_OS_STACK_SIZE-8] = 0x00000000;   // R0
  thread->stack[_OS_STACK_SIZE-9] = 0x11111111;   // R11
  thread->stack[_OS_STACK_SIZE-10] = 0x10101010;  // R10
  thread->stack[_OS_STACK_SIZE-11] = 0x09090909;  // R9
  thread->stack[_OS_STACK_SIZE-12] = 0x08080808;  // R8
  thread->stack[_OS_STACK_SIZE-13] = 0x07070707;  // R7
  thread->stack[_OS_STACK_SIZE-14] = 0x06060606;  // R6
  thread->stack[_OS_STACK_SIZE-15] = 0x05050505;  // R5
  thread->stack[_OS_STACK_SIZE-16] = 0x04040404;  // R4
}

void OS_Launch(unsigned long theTimeSlice) {
  NVIC_ST_RELOAD_R = theTimeSlice - 1; // reload value
  NVIC_ST_CTRL_R = 0x00000007; // enable, core clock and interrupt arm
  StartOS();
}

/* Remove _RunPt from the linked list */
void OS_Kill(void)
{
```

```
  int i;
          _TCB *temp;
  OS_CRITICAL_FUNCTION;
          OS_bWait(&_modifyTCB);
  OS_ENTER_CRITICAL();
          _RunPt->prev->next = _RunPt->next;
          _RunPt->next->prev = _RunPt->prev;
          temp = _RunPt;
          _RunPt = _RunPt->next;  // TODO - could a systick interrupt cause the next thread to be
skipped?
          // search tcb array to find it by checking unique id's
  for(i = 0; i < _OS_MAX_THREADS; i++) {
    if(_threads[i].id == temp->id) {
      _threads[i].id = _OS_FREE_THREAD; // delete thread (symbolically)
      break;
    }
  }
  if(i == _OS_MAX_THREADS) {
     // TODO - this should never happen
  }
  OS_EXIT_CRITICAL();
          OS_bSignal(&_modifyTCB);
  NVIC_ST_CURRENT_R = 0x0; // any write clears
  NVIC_INT_CTRL_R |= NVIC_INT_CTRL_PEND_SV; // trigger pendSV interrupt
  OS_Delay(OS_ARBITRARY_DELAY);
}

static void _OS_Default_Thread(void) {
  while(1)
    OS_Suspend();
}

void OS_Suspend(void) {
  NVIC_ST_CURRENT_R = 0x0; // any write clears
  NVIC_INT_CTRL_R |= NVIC_INT_CTRL_PENDSTSET;  // trigger a systick interrupt to switch threads
  OS_Delay(OS_ARBITRARY_DELAY);
}

void OS_Delay(int count) {
  int i;
  for(i = 0; i < count; i++)
    ;
}

int OS_AddButtonTask(void(*task)(void), unsigned long priority) {
  static unsigned int haveInit = 0;  // only initialize once
  if(!haveInit) {
    PORTF_Init();  // initialize; for now, just select switch (PF1)
```

```
   haveInit = 1;
  }
  // initialize NVIC interrupts for port F
  NVIC_PRI7_R = ((NVIC_PRI7_R&0xFF0FFFFFF)
                                                                    | (priority << 21));
  NVIC_EN0_R |= NVIC_EN0_INT30;
  _OS_SelTask = task; // TODO - handle priority
  return 1;
}

//static unsigned long LastPF1 = 1;
void GPIOPortF_Handler(void) {
//  if(LastPF1 == 1) {
    if(_OS_SelTask != NULL) {
      _OS_SelTask();
    }
//  }
  GPIO_PORTF_IM_R &= ~PORTF_PINS; // disarm interrupt
  OS_AddThread(&DebounceTask, _OS_STACK_SIZE, 5); // TODO - handle priority
}

static void DebounceTask(void) {
  OS_Sleep(200);   // foreground sleeping, must run within 50ms
//  LastPF1 = PF1; // read while it is not bouncing
  GPIO_PORTF_ICR_R |= PORTF_PINS;    // acknowledge interrupt
  GPIO_PORTF_IM_R |= PORTF_PINS;    // re-arm interrupt
  OS_Kill();
  OS_Delay(OS_ARBITRARY_DELAY);
}

void OS_Sleep(unsigned long sleepTime) {
  _RunPt->sleepTime = sleepTime;  // how long to sleep
  _RunPt->sleep = 1;  // indicate this thread is now sleeping
  OS_Suspend(); // trigger thread switch
}

void OS_FindNextThread(void) {
  while(_RunPt->sleep || _RunPt->block) {
    _RunPt = _RunPt->next;
  }
}

void OS_Fifo_Init(unsigned long size) {
  memset(_OS_Fifo.Fifo, 0, sizeof(_OS_Fifo.Fifo));  // initialize all values to 0
  _OS_Fifo.PutIndex = _OS_Fifo.GetIndex = 0;
  OS_InitSemaphore(&_OS_Fifo.notEmpty, 0);
  OS_InitSemaphore(&_OS_Fifo.mutex, OS_BINARY_SEMAPHORE);
}
```

```
int OS_Fifo_Put(unsigned long data) {
  // NOT THREAD SAFE!!
  if((_OS_Fifo.PutIndex + 1) == _OS_Fifo.GetIndex) {
    return 0;
  }
  _OS_Fifo.Fifo[_OS_Fifo.PutIndex] = data;
  _OS_Fifo.PutIndex = (_OS_Fifo.PutIndex + 1) & (_OS_FIFO_SIZE - 1);
  OS_Signal(&_OS_Fifo.notEmpty);
  return 1;
}

unsigned long OS_Fifo_Get(void) {
  unsigned long data;
  OS_Wait(&_OS_Fifo.notEmpty);
  OS_bWait(&_OS_Fifo.mutex);
  data = _OS_Fifo.Fifo[_OS_Fifo.GetIndex];
  _OS_Fifo.GetIndex = (_OS_Fifo.GetIndex + 1) & (_OS_FIFO_SIZE - 1);
  OS_bSignal(&_OS_Fifo.mutex);
  return data;
}

long OS_Fifo_Size(void) {
  return (_OS_Fifo.PutIndex - _OS_Fifo.GetIndex) & (_OS_FIFO_SIZE - 1);
}

void OS_MailBox_Init(void) {
  _OS_Mailbox.data = 0;
   OS_InitSemaphore(&_OS_Mailbox.hasData, 0);
   OS_InitSemaphore(&_OS_Mailbox.gotData, 1);
}

void OS_MailBox_Send(unsigned long data) {
  OS_bWait(&_OS_Mailbox.gotData);
  _OS_Mailbox.data = data;
  OS_bSignal(&_OS_Mailbox.hasData);
}

unsigned long OS_MailBox_Recv(void) {
  unsigned long data;
  OS_bWait(&_OS_Mailbox.hasData);
  data = _OS_Mailbox.data;
  OS_bSignal(&_OS_Mailbox.gotData);
  return data;
}

unsigned long OS_Time(void) {
  return TIMER2_TBR_R / 50;//_us10Count * 10;  // 1us = 20ns * 50
}
```

```
unsigned long OS_TimeDifference(unsigned long start, unsigned long stop) {
  if(stop > start) {
    return (stop - start);  // inputs should already be in 20ns units
  }
  return start - stop;
}
```