EE 445M: Lab 01_ Graphics, LCD, Timer, and Interpreter

Tejasree Ramanuja (td23239) and Chioma Okorie (coo279)

Objectives

The main objectives of this lab is to familiarize ourselves with the LaunchPad board, Vision development system and the TM4C123 ARM Cortex-M4 microcontroller. We wrote a command line interpreter that takes request from the user via the UART and issues command to the LCD and ADC. We implemented both a timer triggered ADC (when multiple samples are requested) using the ADC_Collect function and a busy wait ADC (when only one sample is requested) using ADC_In function. We also implemented a periodic task which will be used to schedule tasks in upcoming labs.

Hardware Design

No hardware design required. We used the ST7735 display, TM4C123 microcontroller and the robot sensor board

Software Design

Here is a documentation generated using Doxygen. It contains the main functions written for this lab. Included is a brief description of each function, input argument(s), a return value, a call and a caller graph if applicable.

1. **Low level LCD driver (ST7735_.c and ST7735_.h files)**

◆ ST7735_Message()

```
void ST7735_Message ( int       device,
                      int       line,
                      char *    string,
                      int32_t   value,
                      int       clearLine
                    )
```

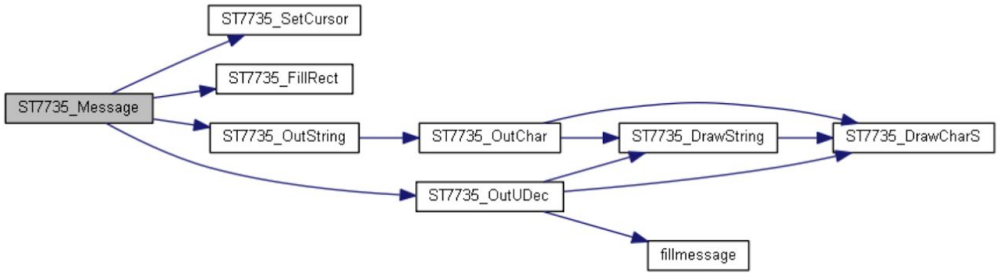
Outputs message to one of the two logically separate displays (top half and bottom half of the LCD). There are 5 lines per display

----------ST7735_Message----------

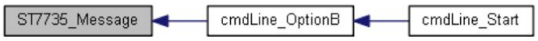
Parameters

device	Species top (0) or bottom (1)
line	Specifies the line number (0 to 3)
string	A pointer to a null terminated ASCII string
value	Number to display

Here is the call graph for this function:



Here is the caller graph for this function:



2. Low level ADC driver (ADC.c and ADC.h files)

◆ ADC0_InitTimer0ATriggerSeq3()

```
void ADC0_InitTimer0ATriggerSeq3 ( uint8_t  channelNum,
                                   uint32_t period
                                 )
```

Initializes ADC channel and timer with the specified channel number and period

---------—ADC0_InitTimer0ATriggerSeq3---------—

Parameters

channelNum channel number to be initialized

period timer interrupt rate

Here is the call graph for this function:

◆ ADC_Collect()

```
void ADC_Collect ( uint32_t channelNum,
                   uint32_t fs,
                   int32_t  buffer[],
                   uint32_t numberOfSamples
                 )
```

returns multiple samples from the ADC

---------—ADC_Collect------—

Parameters

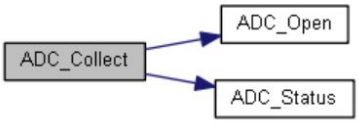
channelNum specifies which analog channel to sample

fs sampling frequency

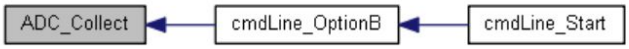
buffer array to store sampled data

numberOfSamples specifies the size of the sample to collect from the ADC

Here is the call graph for this function:



Here is the caller graph for this function:



◆ ADC_In()

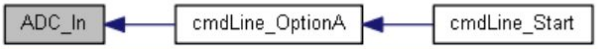
uint16_t ADC_In (void)

---------ADC_In----------

Returns

Result of a single ADC sample

Here is the caller graph for this function:



◆ ADC_Open()

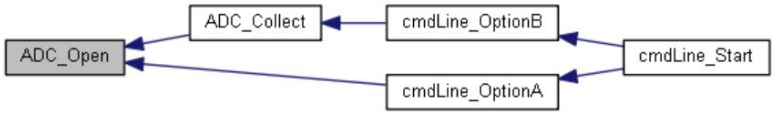
void ADC_Open (uint32_t channelNum)

---------ADC_Open---------

Parameters

channelNum Initializes the specified channel.

Here is the caller graph for this function:



◆ ADC_Status()

int ADC_Status (void)

---------ADC_Status-------

Returns

Status of ADC Conversion; 0 -> complete , 1->not complete

Here is the caller graph for this function:

3. Low level timer driver (OS.c and OS.h files)

◆ OS_AddPeriodicThread()

```
int OS_AddPeriodicThread ( void(*)(void)  task,
                           uint32_t       period,
                           uint32_t       priority
                         )
```

Initializes a periodic task

---------—OS_AddPeriodicThread---------—

Parameters

task periodic task to be performed

period determines how often the task execute (milliseconds)

priority determines the priority of task

Return 1 if successful, 0 if this thread can not be added

◆ OS_ClearPeriodicTime()

```
void OS_ClearPeriodicTime ( void  )
```

Resets the 32-bit global timer

---------— OS_ClearPeriodicTime----------—

◆ OS_ReadPeriodicTime()

```
uint32_t OS_ReadPeriodicTime ( void  )
```

Returns the current value of the global counter

---------— OS_ReadPeriodicTime----------—

Returns

32-bit global counter

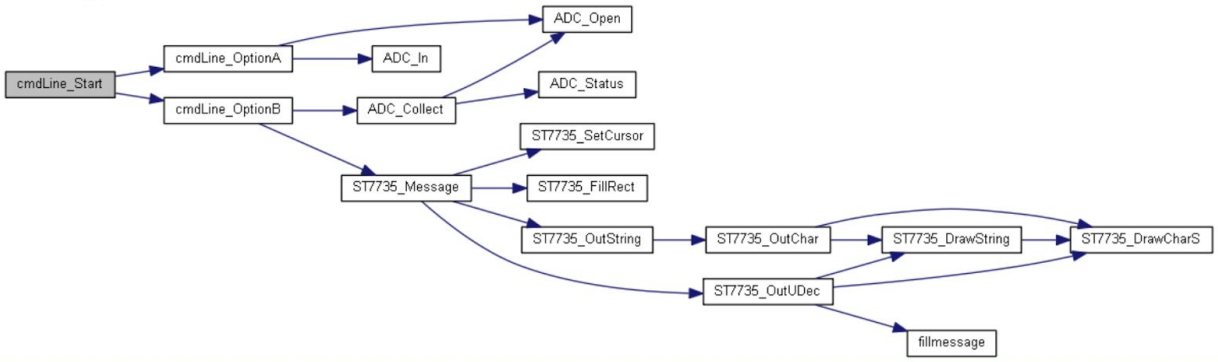
4. High level main program (the interpreter)

cmdLine_Start()

void cmdLine_Start (void)

This functions allows the user to request either 1 or multiple samples from the ADC.

Here is the call graph for this function:

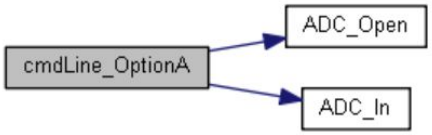


cmdLine_OptionA()

void cmdLine_OptionA (void)

This functions executes when the user chooses to request one sample from the ADC

Here is the call graph for this function:



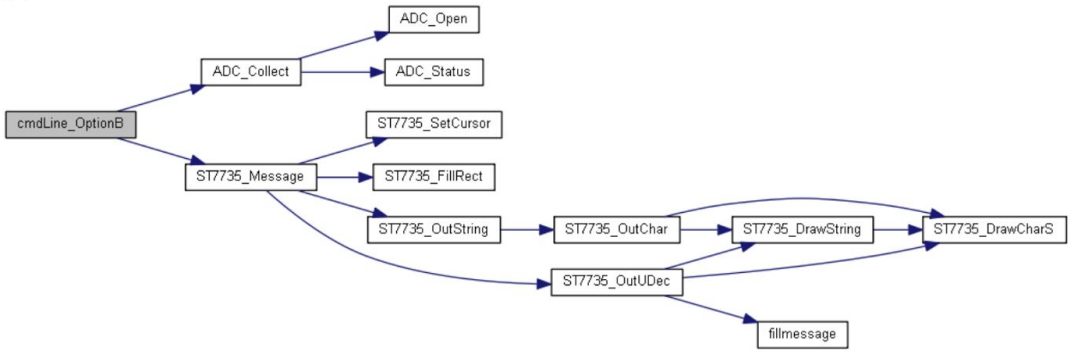
Here is the caller graph for this function:

◆ cmdLine_OptionB()

void cmdLine_OptionB (void)

This functions executes when the user chooses to request multiple samples from the ADC Also allows the user to issue command to the LCD User will specify channel number to sample, the sampling frequency, and LCD device #1 to view the output

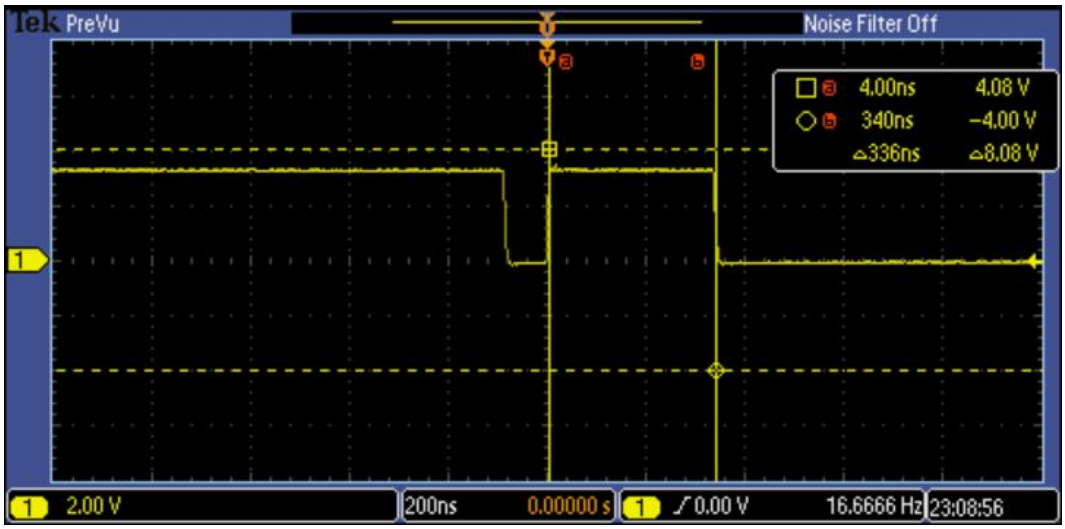
Here is the call graph for this function:



Here is the caller graph for this function:

Measurement Data

1. **Estimated time:** 12.5ns* 25 assembly instructions = **312.5 ns**
2. **Measured time: 336ns**



Analysis and Discussion

1. **Calculating ADC Range, Resolution, and Precision**
   ADC Range: 0 to 3.3V
   ADC Precision: 12-bit (4098 alternatives)
   ADC Resolution: Resolution = Range/ Precision = (3.3V/4098) = 0.000805V

2. **Ways to Start ADC Conversion**
   We can start ADC conversion by calling it from software, using timer (periodic interrupts), PWM (generate external pulse), Analog comparators, GPIO port, or continuously sampling it. We used the timer triggered ADC approach because it allowed us to collect multiple samples at a specific rate. It also allowed us to only trigger the ADC sampling when we need the data instead of either always running it or waiting. This allows the CPU to perform other tasks.

3. **Comparing ISR time measurements**
   3.1. We calculated the time required run the interrupt handler by toggling an LED twice at the beginning of the interrupt and once at the end and measured the time difference in the oscilloscope. This allowed us to determine when the interrupt actually begins. (measured time to run ISR = **336 ns**)
   3.2. We calculated the time required to run the interrupt by using the sysTick timer. We calculated the time difference at the beginning and at the end of the ISR and multiplied it with 12.5ns.
   (systick time difference = 26; time to run ISR = 26*12.5ns = **325 ns**)
   3.3. Measuring the ISR indirectly (when the LED is moved from the timer ISR to the main) gives the same result as first one since the time spent not running the main program is spent in the interrupt and we only have ISR running at a time.

4. **Average time to execute an Instruction**
   Time to execute the ISR (oscilloscope measurement) = 336 ns; we have 25 assembly instructions in the ISR. Average time to execute an instruction = 336ns / 25 = **13.44 ns.**
   There is a **0.94 ns** difference between the system time and the measured time. This might be as a result of the intrusiveness of the debugging instrument or measurement errors.

5. **Range, resolution, and precision of the SysTick timer**
   Range (Interrupt freq): bus freq /(NVIC_ST_CURRENT_R+1)
   Resolution (time between each tick): 1/bus freq
   Precision (size of that the NVIC_ST_CURRENT_R register): 24 bit