CMPE 460 Laboratory Exercise 5

K64 Timers, Interrupts, and Analog-to-Digital Converter

Jacob Meyerson & Charlie Poliwoda
Performed: February 26, 2021
Submitted: March 12, 2021

Lab Section: 2
Instructor: Professor Beato
TA: Brunon Sztuba
Eri Montano
Connor Henley

Lecture Section: 1
Professor: Professor Beato

Your Signature: ______________________________

Contents

Abstract

The purpose of this exercise was to interpret an analog signal being generated by a line scan camera, a photoelectronic transducer. To achieve this, various modules on the K64 board were utilized, including the FlexTimer Module (FTM), Programmable Delay Block (PDB), Analog-to-Digital Converter (ADC), and interrupt service routines (ISRs). The first part of the lab focused on using the FTM timer to measure the time a push button was held, as well as using the PDB timer to generate precise timing delays when flashing an LED. ISRs were written for both timing modules and both on-board input switches to handle the interrupt signals generated by the timers and IO devices on the K64. The second part of the lab required using the ADC to digitize two analog input signals, one from a photocell and one from a TMP36 temperature sensor. The final portion of the lab, and most important objective, was to combine the first two parts to utilize the FTM and PDB timers, interrupts, and the ADC to convert an analog signal from the line scan camera to a digital mapping. Because the data being read from the line scan camera matched up with the expected outputs of a high signal for a white background and a low signal for a black background, this exercise was successful.

Design Methodology

The FlexTimer Module (FTM) is a 16-bit counter timing module on the K64. The FTM counts up to the value stored in the MOD register, and once that value is reached, it will generate an interrupt, indicating that it reached the preset value. Once the interrupt is handled and cleared in the ISR, the FTM will begin counting again. Based on how high the MOD value is set, a certain amount of time will pass before an interrupt gets generated. This is how the FTM keeps track of time. For Part 1 of the exercise, the FTM was configured to use the system clock scaled down by a factor of 128, or 2^7. In order to make any changes to FTM configuration, the WPDIS bit had to be set, which disables write protection on FTM registers. The timer was supposed to generate an interrupt every millisecond. This was accomplished by taking the clock the FTM uses (the default system clock), dividing it by 128 (the prescaler value), and then dividing that by 1000, to convert the clock from the order of MHz to the order of kHz. Equation 1 shows how the MOD value was calculated to produce an interrupt every 1 m sec.

$$FTM\_MOD\_VALUE = \frac{\frac{DEFAULT\_SYSTEM\_CLOCK}{PRE\_SCALER}}{1000} = \frac{\frac{DEFAULT\_SYSTEM\_CLOCK}{128}}{1000} \quad (1)$$

After setting the MOD value and clearing the FTM_CNT register (which is where the current counter value is stored), the Timer Overflow Interrupt Enable mask in the FTM Status and Control register was set, to enable the FTM to generate an interrupt when the FTM_CNT register value reaches the FTM_MOD register value.

The Programmable Delay Block (PDB) is another on-board timing module on the K64. It provides controlled delays that can work on either software or hardware triggers, as well as on a set interval. To configure the PDB to generate interrupts on a 1 sec period, the

PDB Status and Control register was configured to set the PDB in continuous mode, with a prescaler factor of 128, and a multiplication factor of 20. In addition, the PDB enable bit was set, as well as configuring the PDB to work on software triggers. Like the FTM, the PDB has a MOD field that the counter will count up to before triggering an interrupt. Equation 2 below shows how the MOD value was calculated.

$$PDB\_MOD\_VALUE = \frac{DEFAULT\_SYSTEM\_CLOCK}{128 * 20} \tag{2}$$

The mod value was calculated by dividing the system clock by the prescaler value and the multiplication factor. After setting the MOD value, the Interrupt Delay register was set to 10, which is used to schedule an independent interrupt during a PDB cycle. Lastly, the PDB interrupt enable and LDOK bits were set in the Status and Control register. The LDOK bit actually sets the PDB registers configured previously to take the values written to their buffers. Without writing LDOK high, the MOD and IDLY registers would not be updated in the init code. To actually start the PDB, a 1 was written to the SWTRIG bit in the PDB Status and Control register.

Interrupts are signals sent to the main CPU from external devices. When interrupts occur, the CPU stops whatever it is executing to handle the interrupt. Interrupts are handled by writing appropriate interrupt service routines (ISR) for each module that could trigger an interrupt. Every IRQ must be named as the module, followed by "_IRQHandler". For example, the FTM0 IRQ handler must be named "FTM0_IRQHandler" for the processor to go there when the FTM0 triggers an interrupt. Interrupt ISRs were written for PDB0, FTM0, PORTA, and PORTC. PORTA and PORTC handle switches 3 and 2, respectively. To initialize the switches for different edge triggers, the PORT_PCR_IRQC() macro was used from the MK64F12.h header file. Switch 3 passed the argument 9 to configure it for rising edge triggers, and switch 2 passed the argument 11 to configure it to trigger on either edge. To enable interrupts for each module, the NVIC_EnableIRQ() macro was used, also defined in the MK64F12.h header file. The argument passed was also a macro defined in that header file, defined by the module name followed by "_IRQn". This maps each interrupt to the Nested Vector Interrupt Controller (NVIC), which determines which interrupts are enabled and disabled.

To test that the interrupts and timing modules were functioning correctly, the PDB got paired with switch 3, and the FTM got paired with switch 2. The goal was to use the PDB to toggle the on-board LED, and use switch 3 to pause the state of the LED when it was pressed. To accomplish this, the PDB ISR simply had to clear the interrupt mask, and then toggle the state of the LED output bit on PORT B. It's important to note that every ISR must clear the interrupt flag for another interrupt to be correctly generated in the future. Otherwise, the CPU sees the interrupt flag is high again and fires the ISR again, and an infinite loop through the ISR will occur. From the side of switch 3, the ISR cleared the interrupt flag, and toggled the state of the PDB enable bit, which enabled and disabled the PDB timer. This held the state of the LED since the PDB ISR wouldn't get reached when the PDB was disabled, so the LED state didn't toggle. When the PDB got enabled again, the SWTRIG bit was set again to start the timer, and the LED would toggle every second when the PDB ISR fired.

To test the FTM0, the timer would begin when switch 2 was pressed, and upon release, printed the length of time the switch was held, in m sec. To achieve this, in the FTM ISR, a counter got incremented every time the switch was pressed. Because the FTM triggered an interrupt periodically, the size of the counter was equal to the number of milliseconds the switch had been held down. In the switch 2 ISR, if the switch was pressed, a global variable was set indicating the state of the switch. Otherwise, the switch was not pressed, so the counter was displayed to the serial console, indicating how long the switch was held down. The counter was then reset to 0. Both ISRs also cleared the interrupt flags to prevent the ISR from repeatedly firing forever.

The ADC on the K64 is a 16-bit Analog-to-Digital converter. It works by taking an analog signal input and creating a corresponding digital value. The ADC works as a 1-to-1 function, and maps values linearly. To initialize the ADC, it was configured to divide the clock down from 50 MHz down to 6.25 MHz, a factor of 8. It was also set to 16-bit single ended mode. Next, the ADC was calibrated for single ended mode, and then set to select hardware triggering, as opposed to software triggering. The ADC was configured to use the DADP3 channel, and set the DIFF field to 0. The interrupt enable mask in the Status and Control 1 register was set before using the NVIC_EnableIRQ macro to enable the ADC1 interrupt.

An ISR was written for the ADC1, and read the resulting ADC conversion from the ADC_RA register, where the result was placed following the analog to digital conversion. Reading the result also cleared the Conversion Complete flag, which clears the interrupt. The DAC output was then set with the value read from the ADC. The lower 8 bits of the 12 bit ADC value were put in the DAC0_DAT0L register, while the upper 4 bits were stored in the DAC0_DAT0H register.

The PDB in this part of the lab was configured to interrupt every 1 m sec (or at a 1 kHz frequency). To achieve this, the MOD value of the PDB was set to be 50,000. The PDB was set to run in continuous mode, and run on software triggering. The pre trigger output was also enabled. Every time the PDB generated an interrupt, it would trigger the ADC to begin a new conversion.

The photocell used to convert an analog signal to a digital value is a photo-transducer. It converts light to resistance. As more light is exposed to the photocell, the lower the resistance drops. To measure the voltage across the photocell, it was connected according to the schematic below.

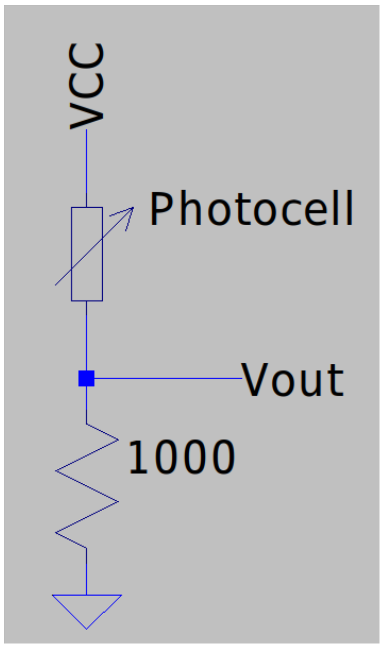


Figure 1: Photocell Schematic

The schematic shown in Figure 1 is a simple voltage divider. The voltage across the photocell is calculated below in Equation 3.

$$V_{photocell} = V_{CC} - V_{out} \tag{3}$$

The analog voltage V_{out} is provided directly to the ADC on the K64, where both the decimal and hexadecimal values were displayed to the serial console through UART. It was expected that the values read in would increase when the photocell was exposed to more light, and would decrease when the photocell was exposed to less light. This was expected because the photocell decreased its variable resistance when exposed to more light, so the voltage drop across the photocell changes and lowers. When the photocell's resistance lowered, less voltage was dropped, so the voltage at the node between the resistive loads increased, which resulted in a higher ADC value. Contrary, when the photocell was exposed to less light, its resistance increased, so the voltage drop across it also increased, and resulted in a smaller voltage at the node between the resistive loads.

In order to verify that the ADC was functioning correctly, it was connected directly to ground, where it was expected to produce either 0, or a very small number close to 0. The ADC was also then connected to V_{CC} (3.3 V, where it was expected to read it's maximum value, 65535, which is $2^{16} - 1$.

The TMP36 is a temperature transducer that converts temperature to voltage. The same configuration for the ADC was used from the photocell circuit shown in Figure 1, but no circuit was necessary, since the TMP36 could be powered directly from the K64, and the output voltage pin could be directly connected to the ADC on the K64. Based on the figure below of the table from the TMP35/TMP36/TMP37 datasheet, an equation was modeled to convert from voltage to temperature in software.

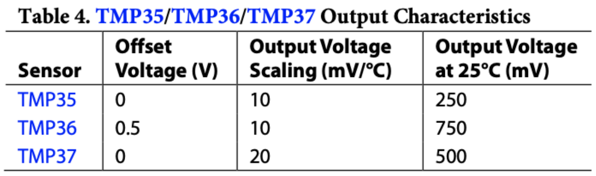


Table 4. TMP35/TMP36/TMP37 Output Characteristics

Sensor	Offset Voltage (V)	Output Voltage Scaling (mV/°C)	Output Voltage at 25°C (mV)
TMP35	0	10	250
TMP36	0.5	10	750
TMP37	0	20	500

Figure 2: TMP36 Output Characteristics[1]

As shown in Figure 2, the output from the TMP36 is directly proportional to temperature in Celsius, so voltage was converted to Celsius before being converted to Fahrenheit. The equation to convert voltage to Celsius is shown below in Equation 4.

$$T_{Celsius} = \frac{(\frac{V_{CC}}{2^n} * V_{in}) - V_{offset}}{V_{scaled}} = \frac{(\frac{3300.0}{65536.0} * V_{in}) - 500.0}{10.0} \tag{4}$$

To convert temperature to voltage, the voltage input must be multiplied by the resolution of the ADC, which is the maximum voltage divided by the number of discrete digital values that can be generated. Then, any offset value must be subtracted from that quantity before being scaled by any scaling value. Based on the Table from Figure 2, the Output Voltage Scaling was 10 mV per degree Celsius, and the Offset Voltage was 0.5 V, which is equal to 500 mV. The V_{CC} voltage was equal to 3.3 V, and the n value was 16, since the ADC on the K64 is 16 bits. All the voltages were substituted in the units of mV. The reason every integer in the equation is followed by a ".0" is to prevent C from doing integer division and rounded the intermediate and final calculations to integers, since floating point precision was expected.

The camera used in part 3 of this exercise is a line scan camera. It has a CMOS linear sensory array of 128 pixels. There are five connections made from the K64 to the camera. There are power and ground lines, where the K64 provides 5 V and a reference ground to the camera, a CLK line and SI line, which are both inputs to the camera, and an Analog Out line, which is the camera's output. To interface with the K64, the CLK and SI lines were configured to work from GPIO pins on the K64, and the Analog Output from the camera gets connected to the ADC on the K64. The figure below shows the timing diagram for the line scan camera, obtained from a forum on NXP's website.[2]

[1] Table taken from Analog Devices Low Voltage Temperature Sensors TMP35/TMP36/TMP37 Datasheet page 8

[2] Timing diagram taken from NXP's website at https://community.nxp.com/t5/University-Programs-Knowledge/Line-Scan-Camera-Use/ta-p/1105313

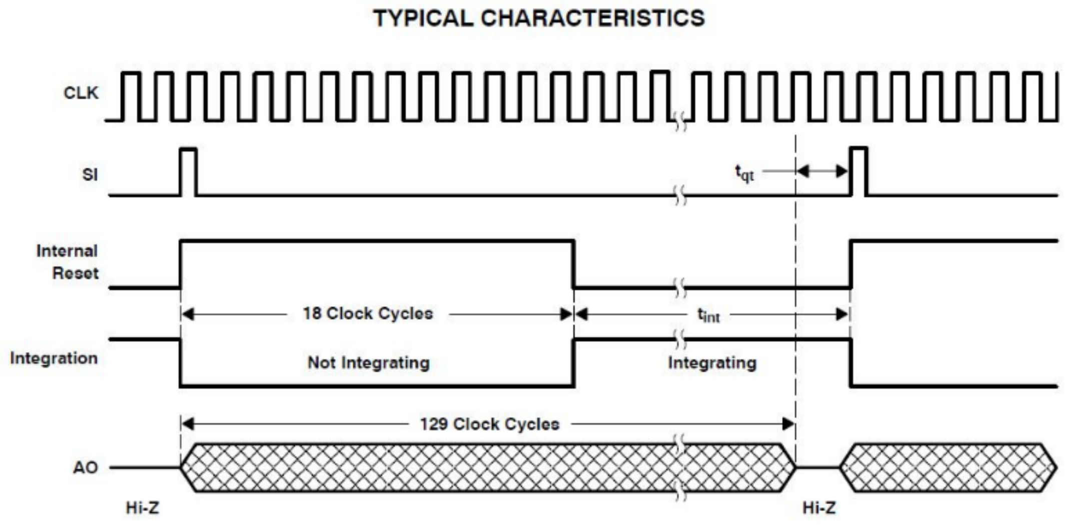


Figure 3: Line Scan Camera Timing Diagram[2]

Based on Figure 3, the CLK signal is expected to be a constant signal that keeps time. Changing the clock frequency changes how quickly the camera can run and return data. The SI line is the System Integration signal, which pulses high when the camera begins collecting data, and takes 18 clock cycles before it begins to integrate the data and put it in the 128 pixel array. The limiting factor as to how quickly data from the camera can be read is the integration time. According to the NXP website regarding the line scan data, the camera has a minimum integration time[3] of 33.75 µ sec, and a maximum time[3] of 100 m sec. If that maximum time is exceeded, then the capacitors connected to the output of the integrator will saturate, and the data would be ruined[3].

To configure the ADC to interpret the data from the line scan camera, the ADC0 module was used. It was configured to use the single ended 16 bit conversion mode, without a clock divider. The ADC was then calibrated, and set to select the hardware trigger, as opposed to the software trigger. The ADC interrupt was enabled using the NVIC_EnableIRQ() macro. The final step was to set up the FTM2 to trigger on ADC0, since the FTM2 controls the CLK signal. To achieve this, the SIM_SOPT7 register was configured to select the FTM2 timing module to trigger on, in addition to selecting alternative trigger edge and pretrigger A. To handle an ADC0 interrupt, an ISR was written that read the data stored and converted from the ADC, and copied it to a local 16 bit variable. Reading this data also cleared the Conversion Complete Flag in the ADC.

The CLK signal was controlled by the FTM2 module. To configure the FTM2 correctly, write protection first had to be disabled for the settings to get changed. Then, the FTM2 output was set to be '1' after being initialized. The CNT register, which stores the timer's current count, was set to 0, and the MOD register was set to be roughly 200. The equation

[3]Camera Limitations section taken from NXP's website at https://community.nxp.com/t5/University-Programs-Knowledge/Line-Scan-Camera-Use/ta-p/1105313

below shows how the MOD value was calculated.

$$FTM2\_MOD = \frac{\frac{DEFAULT\_SYSTEM\_CLOCK}{100000}}{2} \tag{5}$$

The FTM2 MOD register was set according to Equation 5 in order to achieve a period of approximately 10 µsec. To set the FTM to have a 50% duty cycle, the following value was stored in the FTM2_COV register.

$$FTM2\_COV = \frac{\frac{DEFAULT\_SYSTEM\_CLOCK}{100000}}{4} \tag{6}$$

The FTM2_COV register was set to be half of the MOD value, in order to have a 50% duty cycle. The FTM2 was then configured to set edge-aligned mode, enable high-true pulses, enable hardware triggering, use the system clock, enable interrupts in the FTM2 Status and Control register, and enable the FTM2 interrupts in the NVIC. Lastly, write protection was re-enabled. The FTM2 ISR got called once at the start of every integration period. When the ISR got called, it sent out the SI pulse through GPIO pin 23 on PORTB, toggled the clock for 128 cycles, and then stored the data from the ADC into an array of length 128, to store each pixel from the camera.

The K64 Periodic Interrupt Timer (PIT) is an array of timers that can trigger interrupts in the main processor. The PIT0 was used to determine the integration period. To correctly configure the PIT, it was configured to enable interrupts and determine the integration time. When the frequency was set, the PIT_LDVAL0 register was set to a value that would be counted down from until the PIT reached 0. Then, an interrupt gets triggered, and the PIT0 ISR gets fired. To determine the value stored in the PIT_LDVAL, the following equation was used.

$$PIT\_LDVAL0 = DEFAULT\_SYSTEM\_CLOCK * INTEGRATION\_TIME \tag{7}$$

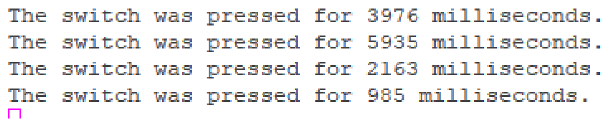
The integration time was changed from the default value of 0.0075 to 0.015. This was to get the PIT to run an integration time as close to 33.75 µsec as possible before the results got skewed. The last step of the PIT configuration was to enable the PIT0 interrupt in the NVIC. In the PIT ISR, the FTM2 got reset because the FTM2 is constantly counting. This way, it was possible to control when the line capture occurred.

It was expected that with a very strong light intensity, and a perfectly white background, the camera's analog output would be 5 V. If the camera lens was covered, it was expected that the output signal would be a flat line at 0 V. If there was a black stripe down the center of a perfectly white background, it was expected that the output signal would be high, with a dip down to 0 V when it detected the black background. Based on this expectation, if the black line were slid to the left or right, there would be a corresponding shift in the output waveform where the output dipped from 5 V down to 0 V.

The data from the camera was then fed from the K64 to a MATLAB script that captured the data, and created two additional plots, in addition to plotting the camera data. A smoother plot was created by using a five point averager, where each data point was added to the two values both above and below it, and then divided by five. This method was expected to result in a smoother graph of the data, and eliminate some of the noise and rough edges in the plot. A binary plot was also created, where every data point was compared with a threshold value. If the data was greater than that threshold, a 1 was graphed. Otherwise, a 0 was graphed. The threshold was determined by dividing the maximum output in half.

Results & Analysis

Figure 4 below shows the PuTTY serial output from the K64 when it was being tested with switch 2 and the FTM.



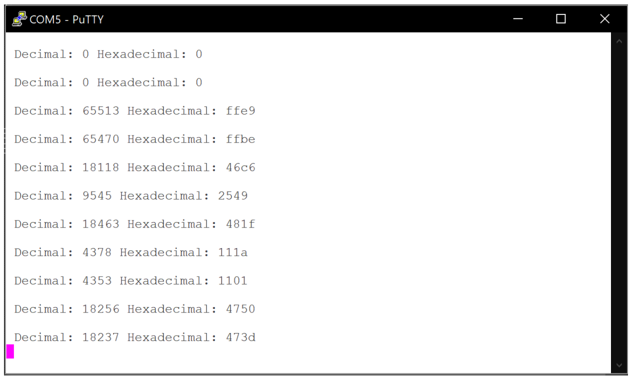
```
The switch was pressed for 3976 milliseconds.
The switch was pressed for 5935 milliseconds.
The switch was pressed for 2163 milliseconds.
The switch was pressed for 985 milliseconds.
```

Figure 4: PuTTY Output For FTM Timing Button Press

As shown in Figure 4, the FTM time was reported in milliseconds. Each output was printed on the release of switch 2, and pressing switch 2 multiple times correctly produced the time it was held, proving that the program reset the counter correctly. It was expected that the terminal output would read values for four seconds, six seconds, two seconds, and one second, respectively. While the values in milliseconds were not perfectly accurate, they were precise enough to see the proper functionality. This matches the expected behavior that the time switch 2 was pressed would be printed to the serial console, in m sec at the time the switch was released.

As expected from the Design Methodology section, pressing switch 3 the first time enabled and started the PDB timer, and began flashing the LED. The switch was tested to hold the LED state both high and low, and was successful in pausing the LED when it was both on and off, which matched the expected behavior.

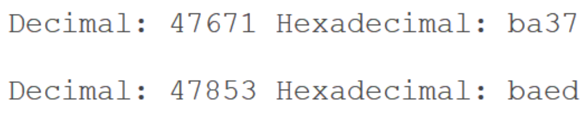
Figure 5 below shows the output from the K64 when using the ADC with the photocell, as well as when the ADC was tested using V_{CC} and GND values.



```
Decimal: 0 Hexadecimal: 0

Decimal: 0 Hexadecimal: 0

Decimal: 65513 Hexadecimal: ffe9

Decimal: 65470 Hexadecimal: ffbe

Decimal: 18118 Hexadecimal: 46c6

Decimal: 9545 Hexadecimal: 2549

Decimal: 18463 Hexadecimal: 481f

Decimal: 4378 Hexadecimal: 111a

Decimal: 4353 Hexadecimal: 1101

Decimal: 18256 Hexadecimal: 4750

Decimal: 18237 Hexadecimal: 473d
```

Figure 5: PuTTY Output For Photocell

It was expected that the hex and decimal outputs would be close to 0 when the ADC was connected to ground, and close to 65535 when connected to 3.3 V (V_{CC}). Then, when the photocell was connected, it was expected that the ADC value would increase from its typical value when the photocell was exposed to more light, and would decrease when the photocell was covered and not exposed to light. In Figure 5, the first two printouts show that the ADC outputted 0 when connected to ground, and the next two lines displayed close to 65535, the max value, when the input to the ADC was 3.3 V. This matches the expected behavior. The next two test cases on the printout are irrelevant, since information was displayed while the ADC was floating, since the photocell was not connected yet. The value of 18463 was displayed when the photocell was sitting idle, without being exposed to excess light. Then, the photocell was covered, and minimal light was exposed to it. This caused the photocell to increase its resistance, so more voltage was dropped across it, leaving less voltage at the node between the resistive loads, and thus produced a smaller ADC conversion value. Figure 6 shows the remaining data taken from the ADC when a bright light was shined on the photocell and displayed on the serial output.

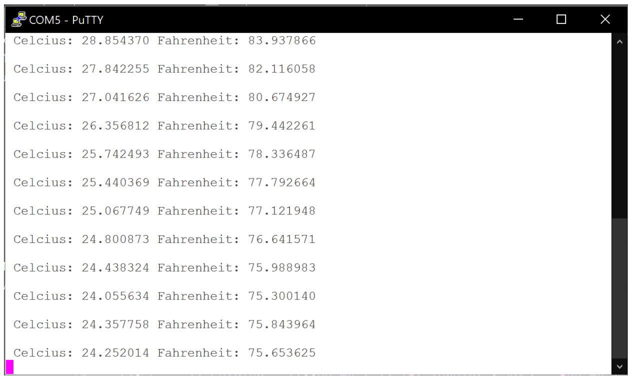


```
Decimal: 47671 Hexadecimal: ba37

Decimal: 47853 Hexadecimal: baed
```

Figure 6: PuTTY Output For Photocell Continued

As shown in Figure 6, the output of the ADC was around 47500 when shining a flashlight directly on the photocell. It was expected that the ADC value would increase when there was more light exposed to the photocell, the photocell's resistance lowers, so the voltage drop across it decreases. As a result, the voltage on the node between both resistive loads increases, which matches the expected results.

Figure 7 below shows the output of the temperature sensor after the temperature of the TMP36 was increased by clamping fingers over it.



```
Celcius: 28.854370 Fahrenheit: 83.937866

Celcius: 27.842255 Fahrenheit: 82.116058

Celcius: 27.041626 Fahrenheit: 80.674927

Celcius: 26.356812 Fahrenheit: 79.442261

Celcius: 25.742493 Fahrenheit: 78.336487

Celcius: 25.440369 Fahrenheit: 77.792664

Celcius: 25.067749 Fahrenheit: 77.121948

Celcius: 24.800873 Fahrenheit: 76.641571

Celcius: 24.438324 Fahrenheit: 75.988983

Celcius: 24.055634 Fahrenheit: 75.300140

Celcius: 24.357758 Fahrenheit: 75.843964

Celcius: 24.252014 Fahrenheit: 75.653625
```

Figure 7: PuTTY Output For TMP36 Temperature Sensor

The lab room was determined to be approximately 70 to 72 degrees Fahrenheit. The temperature sensor initially produced a reading of 71 degrees Fahrenheit, which makes sense since it was sitting idle in the room. Then, the temperature of the sensor was increased by tightly clamping fingers around both ends of the sensor. It was expected that the measurement would rise, and as shown in Figure 7, it rose to above 83 degrees Fahrenheit. After it rose, it was released to once again be sitting idle in the lab. The figure above shows the temperature slowly decreasing back down to the temperature of the lab, around 71 degrees Fahrenheit. In addition, each Celsius measurement was compared to the corresponding Fahrenheit value, and was determined that both temperatures correctly indicated the temperature of the device each time the ADC made a conversion.

The camera captures light while the integration timer is active. During this time, capacitors are charged. When the integration timer is switched off, the output is transmitted. The voltages of these capacitors is what is represented in each pixel. When the light is more intense, the output is able to approach V_{CC}. This can be observed in Figure 8 below, which is the output waveform when the camera scanned a white piece of paper.

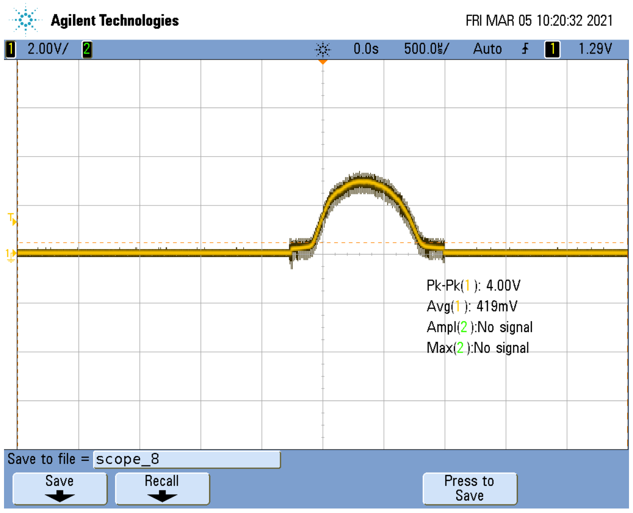
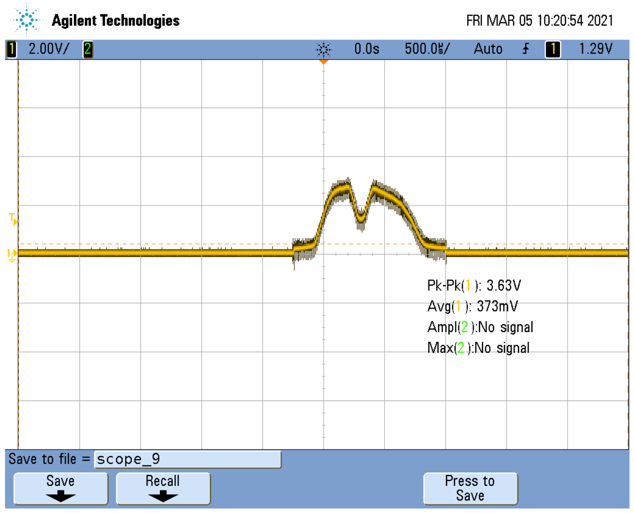


Figure 8: Oscilloscope Capture for High Output

The intensity of the light was the determining factor for the probed results. When a dark stripe was added to the paper, there was a dip in the voltage that matched where the camera was facing. Figure 9 presents the capture with the dark stripe.



Pk-Pk(1): 3.63V
Avg(1): 373mV
Ampl(2):No signal
Max(2):No signal

Figure 9: Oscilloscope Capture for Varied Output

The resulting capture does not dip all the way down to 0 V when pointed at the dark strip for the same reason it does not reach 5 V when pointed at a piece of paper. With a brighter room and a less reflective, dark coating could be used to achieve more accurate results. The results achieved were still able to clearly show proper functionality of the camera.

After running the MATLAB script, the camera output was periodically filling vectors with data to be plotted. Figure 10 shows the plots of the camera's output when looking at an illuminated sheet of paper with a white background.

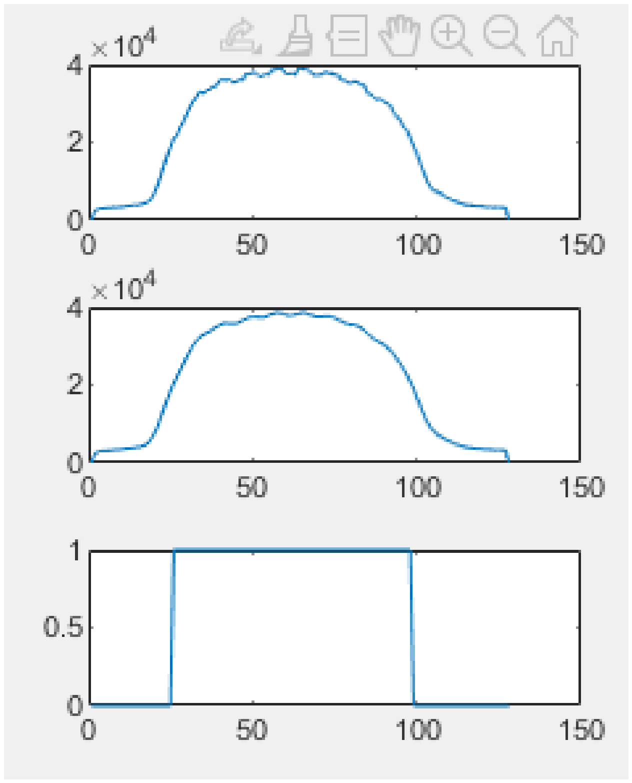


Figure 10: MATLAB Plots for High Output

The three plots presented above show that the MATLAB script was successfully gathering data from the camera. The first plot represented the raw data leaving the camera. The middle plot used a method called five-point averaging. The purpose of this was to smooth out the raw data. By taking the two data points above and the two data points below each data point and averaging them, a more fluid plot was generated. The third plot was a binary representation. The purpose of this was to represent the camera's data as a digital output. This was done by comparing each data point to half of the maximum output. If the output was lower, it was graphed as a zero. If it was greater than that threshold, it was represented as a one.

The results were consistent with the previously presented oscilloscope captures. Similarly, a dark stripe was placed in the center of the camera's vision. The plots with this data can be seen in Figure 11.

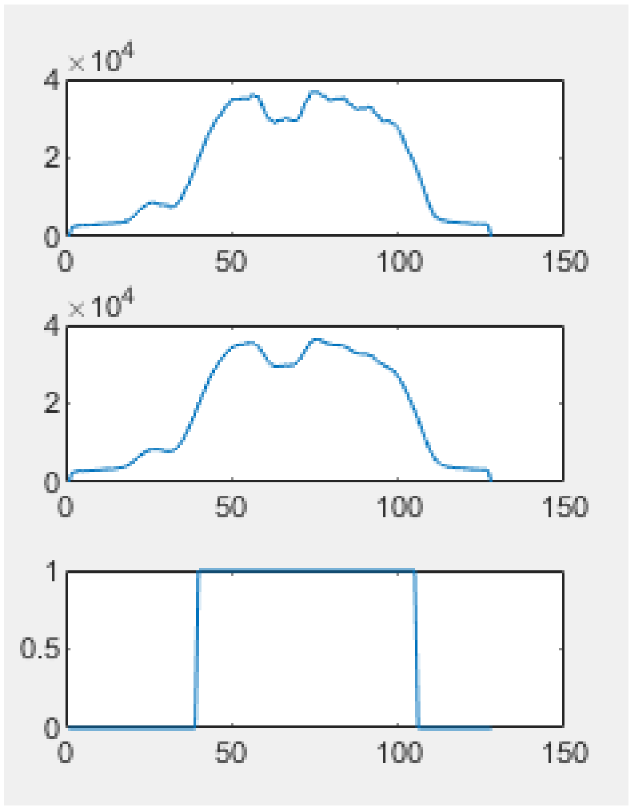


Figure 11: MATLAB Plots for Varied Output

The dark stripe in the center caused a reduced amount of light to be sent to the sensor. This meant that the camera produced less voltage for those points. One thing that could have been changed when conducting the experiment was the calculation for the binary trace as well as the amount of light that was being sent into the camera. When more light was allowed into the camera, the closer the output could get to 5V. With less resolution between bright and dim surfaces, the dark stripe did not cause enough of a dip to send the binary plot to read as zero, which was expected. It's important to note that each side of the camera signal from the dip down will be used to verify that there is a white piece of track on either side of the black line. If one side of the track was completely black, then one side of the camera's output signal would be low, while the other side would remain high.

Conclusion

The exercise performed gave insight into some different modules on the K64 board as well as effectively utilizing interrupts. The FTM was used to time a button press. The PDB toggled an LED periodically. The ADC was able to read in analog signals from different transducers and convert them into a binary representation. Interrupts were the driving force for each module to run its routine. The different modules were able to be used together to correctly implement the line scan camera. The exercise was a success as the camera was able to produce outputs consistent with the data in front of it.

Analysis

Question 1

The smallest amount of time that is able to be measured is $\frac{1}{SYSTEM\_CLK}$. This is equal to around 48.8 n sec. The clock frequency is able to be adjusted so that a greater range is achieved at the cost of resolution. After dividing the clock by the max pre-scaler of 128, combined with the max multiplication factor of 40, the clock can be reduced to measure at around 4 kHz. With a MOD register of 16 bits, the largest amount of time that can be measured is 16.4 seconds.

Question 2

With a pre-scaler value and multiplication factor of 1, the clock was operating at approximately 20 MHz. This means that the MOD register is counting up to about 1.56 m sec. This equates to 640.18 Hz. To prevent aliasing, we need to sample at $\frac{1}{2f}$. This would make the highest frequency to read at 1.28 kHz.

Questions

Question 1

Every IRQ handler must clear the IRQ flag for another interrupt to be correctly generated in the future. Otherwise, the CPU sees the interrupt flag is high again and fires the ISR again, and an infinite loop through the ISR will occur. Without clearing the IRQ flag, the program will never stop entering the ISR.

Question 2

The RTC (real time clock) is the only timing module that is capable of running when the K64 is off, because it has an independent power supply.

Question 3

The LDOK field for the PDB0 SC register needs to be asserted because the values written to the different PDB registers don't actually get written there at the time those lines of code get executed. The data is written to a buffer, and the PDB registers don't get those values until the LDOK field is asserted, and the registers take on the values from the buffer. Without asserting the LDOK field, no changes to the PDB MOD, IDLY, CHnDLYm, DACINTx,and POyDLY registers would ever be made.