Lab 1 - Introduction to Code Composer and the MSP432

Priyanka Makin, Savio Tran, and Zach Passarelli

September 16, 2016

ECEN 2020

This lab serves to ease us into using the MSP432 Launchpad and configuring it using embedded C. In this lab, we utilize pointers and bit masks to configure pins so that we can activate LEDs on the board. Then, we use an oscilloscope to measure the time it takes for a pin to toggle from high to low, which gives us data to configure a square wave signal with a frequency of 10 Hz. Finally, we create a function to output a square wave signal with a configurable duty cycle.

Code Composer

We create a new project in Code Composer Studio and ensure the project is configured to work with the MSP432 LaunchPad, as well as supporting debugging software. The first piece of code initializes pointers to access the addresses of ports 1 and 2 in the board's registers. Using these pointers we can toggle the direction, input, and output states of the pins encompassed in ports 1 and 2.

For the first exercise we configure pin 1.0, which controls a red LED, to output. Within an infinite *while* loop, we perform a NOR operation on pin 1.0's output to toggle its state between on and off, as well as using a *for* loop with a configurable number of iterations to create a delay between the NOR toggles. In essence we simulate a blinking red LED on the LaunchPad.

Within Code Composer we also test the *sizeof* function to investigate the sizes of varying data types used in declaring variables. The data from this test is reported in Table 1. It is important to note that pointer types are all the same size, regardless of the data type they are referencing.

Data Type	**Size (bytes)**	**Data Type**	**Size (bytes)**	**Data Type**	**Size (bytes)**	**Pointer Type**	**Size (bytes)**
char	1	double	8	int64_t	8	uint8_t*	4
int	4	long double	8	uint8_t	1	char*	4
short	2	int8_t	1	uint16_t	2	int*	4
long	4	int16_t	2	uint32_t	4	unsigned int*	4
float	4	int32_t	4	uint64_t	8	uint16_t*	4

Table 1 - Data Type Sizes

Modifying the Blink Code

We amend our blinking red LED code by activating a blinking green LED alongside it. The green LED is controlled by pin 2.1, so we configure the pointer for port 2 direction to activate the pin's output. Another NOR operation is used to toggle pin 2.1's state similar to pin 1.0, and another *for* loop is added for additional delay. This code simulates the red and green LEDs alternating their states for a somewhat impressive light show.

Toggle Frequency

We investigate how the delay caused by our *for* loops is related to the frequency at which the pins toggle their states. Pin 2.5 is configured to output and is toggled between a delay. Using the oscilloscope, we measure the period of the output of pin 2.5 with varying values of iterations set in the delay *for* loop. The data is recorded in Table 2.

Loop Iterations	1	10	100	1,000	10,000
Measured Period (ms)	0.019	0.079	0.674	6.63	72.8

Table 2 - Pin Toggle Periods

Generating a 'best fit' linear equation from this data yields the equation:

$$i = (msec + 0.1625) / 0.0073$$

with i representing the number of iterations in the *for* loop. Using this equation will allow us to find the number of iterations to create a delay for a desired toggle frequency. For a period of 100 milliseconds, the delay loop will have approximately 13,721 iterations.

Using methods similar to the previous section, we configure pin 1.7 to toggle output with a *for* loop delay of 13,721 iterations. This means the pin's output will toggle at a period of roughly 100 milliseconds, or 10 Hertz. We measure the pin's output with the oscilloscope and capture the square wave signal as seen in Figure 1. The period on the device reads 99.86 milliseconds, which is very close to our predicted period.

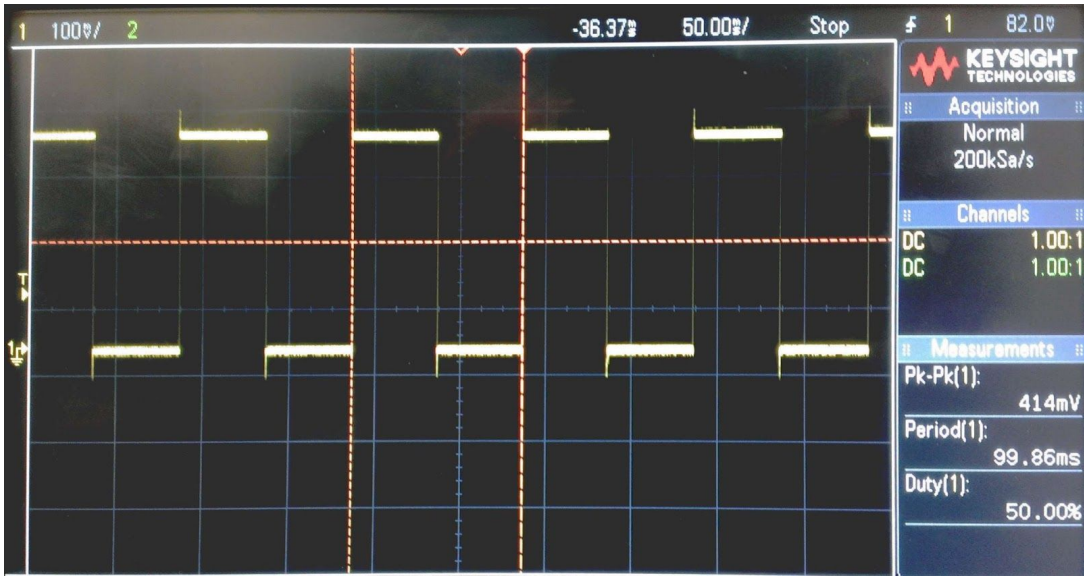

Figure 1 - Oscilloscope Screen Cap at 10 Hz (or T = 10 msec)

Configurable Duty Cycle

In the final lab exercise, we code a function that will pass in an output pointer, an iterations variable similar to one used in the previous section, and a duty cycle variable, simulating a toggling pin output with a customizable duty cycle. The function will perform a calculation with the iterations and duty cycle variables which will be used in two *for* loops of varying delay. Within this function an infinite *while* loop is created that toggles the output of the pointer passed into the function, with the aforementioned delay loops. In essence, the function outputs a square wave signal which will be high for a duration respective to the chosen duty cycle, and then become low for the remaining period.

We test the function using an iterations value of 13,721, and measure the output of a pin for varying duty cycle inputs. Figure 2a-d shows the output with duty cycles of 25%, 50%, 75%, and 99%, respectively.

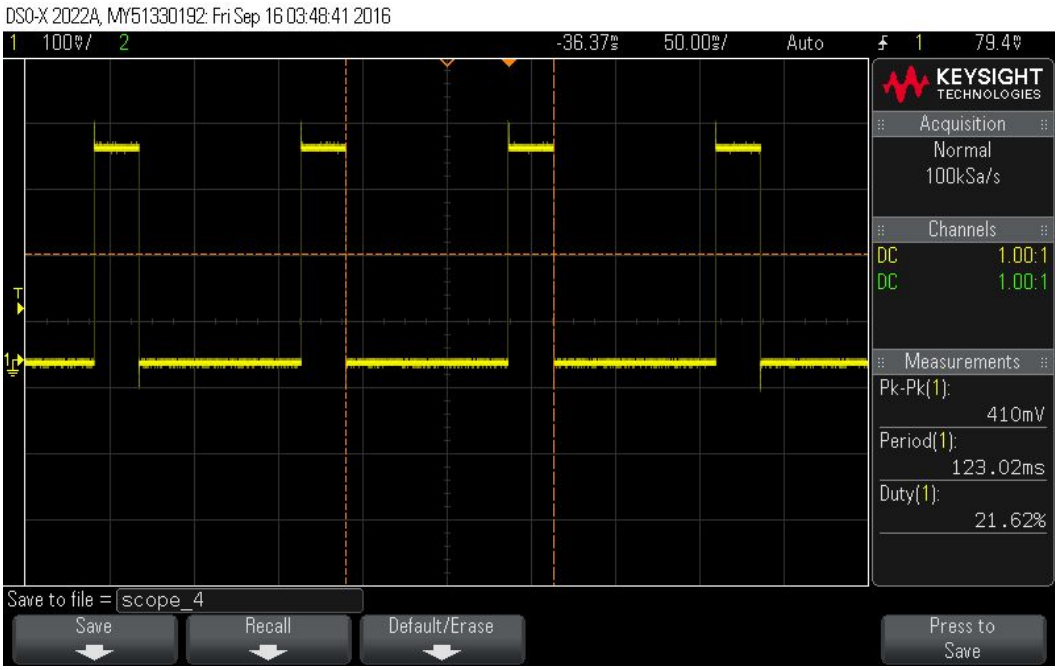

Figure 2a - Duty Cycle of 25%

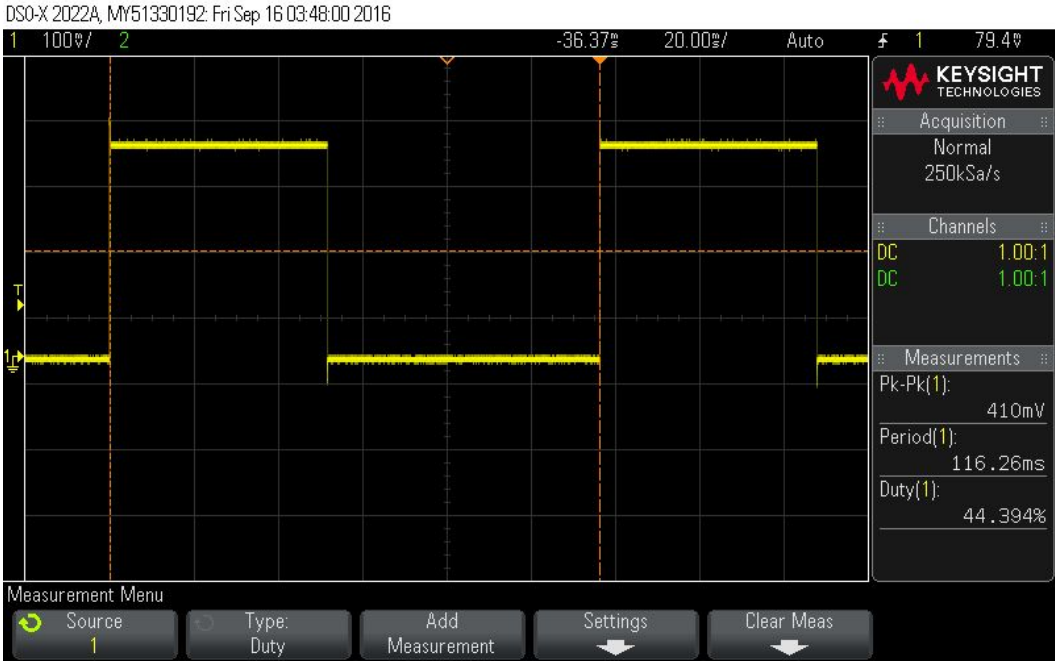

Figure 2b - Duty Cycle of 50%

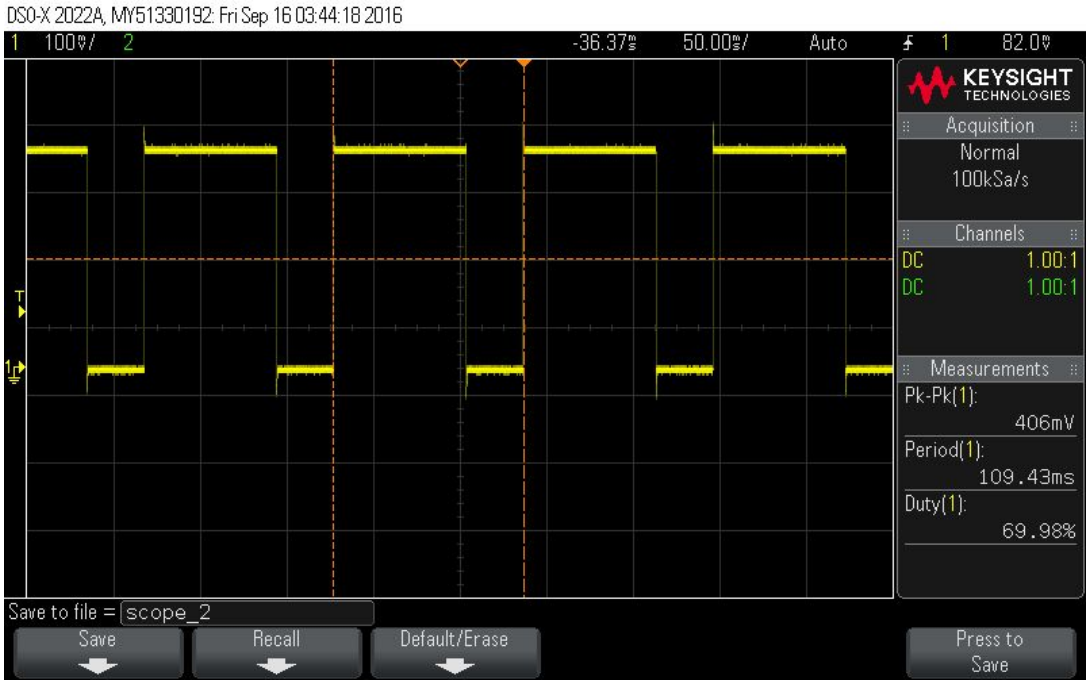

Figure 2c - Duty Cycle of 75%

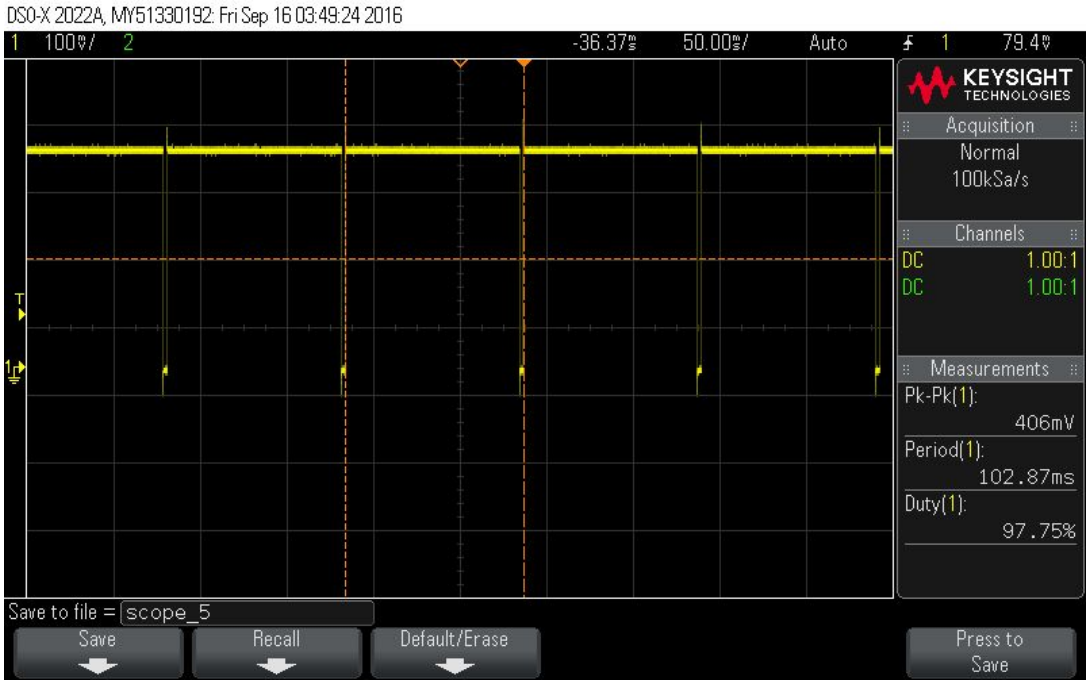

Figure 2d - Duty Cycle of 99%

Lab Report Questions

1. The following are descriptions for an emulator, simulator, loader/installer, and debugger.
    a. An **emulator** is either hardware or software that pretends to be another particular device or program that other components interact with. The "host" is the computer system you are working on and the "guest" hardware/software is what the computer system is behaving like.
    b. A **simulator** is a computer program or device that imitates a process or system over time.
    c. A **loader/installer** is responsible for finding and loading programs and libraries onto a system.
    d. A **debugger** is a computer program that tests and debugs other code. Debuggers can use simulators and/or emulators to be able to gain more control over the code's execution this reduces the speed, however. A debugger can run programs line-by-line to stop and test specific points. Once the debugger runs through the code, it shows at what line there are errors in the program and generally gives an error code to describe what went wrong in execution.
2. One way to power the MSP432 without the USB cable would be to connect the board to an external power supply such as a battery. You can use wires to attach the battery to the 5V trace on the board. In order for the XDS110 to maintain power, the power jumpers must be left in place.
3. Energy Trace technology can be used to make a real world embedded system more efficient. By knowing where and how power is consumed in an embedded system, we can modify it to minimize power consumption and create a more efficient system.
4. Connecting an indicator LED to the board depends on the LED you are using and the voltage, thus the required resistor would vary. However, in any case, you would put the LED in series with the resistor. You would also need to configure the port of the pin the LED is attached to and set the pin to output.
5. A **breakpoint** is a point in a program that, when reached, triggers some special behavior useful to the process of debugging. Sometimes they are used to pause program execution and/or dump some or all values of program variables.
6. The registers PxIN, PxOUT, and PxDIR are all 32-bits. This is also the word size of the MSP432 (word size refers to the size of the registers/instruction operands).
7. Use the following lines of code to create a pointer to P2DIR, then use a bitmask to set pins P2.4 - P2.7 without changing the configuration of the rest of the pins.
   uint8_t* ptrdir2= uint8_t* 0x4000_4C05;
   *ptrdir2 |= 0b11110000;

Real World Application

This lab utilized pointers to configure pins in the microcontrollers. While there are more efficient ways to do this, the configuration of pins is vital in creating an embedded system. Understanding how to configure pins is a basic skill that we will need in order to have the microcontroller take in data and process it the way we would like it to. It can also let us control the microcontroller's feedback to any inputs.

Conclusion

This lab acted as a decent intro to configuring the MSP432. While it was difficult initially to figure out how to use pointers to configure pins, it was a good way to learn how the pins are accessed and activated using binary/hex values, and it was also good practice for using pointers and bit masks. In the end it felt somewhat rewarding to see the LaunchPad produce outputs with seemingly simple lines of code, and gives an idea to what potential this board may have with other types of inputs and outputs.

Appendix

Code file: main.c

```c
#define BLINK_LED //Code block for red and green LED blinking
#define SIZE //Code block for size of data types
#define RATE (13721) //Define number of iterations for delay loops
#define RATE_TEST //Code block for delay loops test
#define FREQ_TEST //Code block for frequency test
#define FUNC //Code block for duty cycle function
#define DUTY_CYCLE_FUNC //Code block in main for calling duty cycle function
#define DUTY_CYCLE (75) //Configurable duty cycle value

#include "msp.h"
#include <stdio.h>
#include <stdint.h>

void configure_duty(uint8_t dutyCycle,uint16_t rate, uint8_t *ptrout2);

void main(void){
  WDTCTL = WDTPW | WDTHOLD;          // Stop watchdog timer

  uint8_t*ptrdir1 = (uint8_t*)0x40004c04; //pointer for P1 direction
  uint8_t*ptrout1 = (uint8_t*)0x40004c02; //pointer for P1 output
  uint8_t*ptrin1 = (uint8_t*)0x40004c00;  //pointer for P1 input
  uint8_t*ptrdir2 = (uint8_t*)0x40004c05; //pointer for P2 direction
  uint8_t*ptrout2 = (uint8_t*)0x40004c03; //pointer for P2 out
  uint8_t*ptrin2 = (uint8_t*)0x40004c01;  //pointer for p2 input

#ifdef BLINK_LED
 int count = 0;
 int i, j;
 *ptrdir1 = 0b00000001; //Set pin 1.0 to output direction
 *ptrdir2 = 0b00000010; //Set pin 2.1 to output direction

 while(1) {
    *ptrout1 ^= 0x01; //Toggle pin 1.0 output
    for(j = 0; j < 10000; j++); //Delay
    *ptrout2 ^= 0x02; //Toggle pin 2.1 output
    for(j = 0; j < 10000; j++); //Delay

  count++;
  for (i = 3000; i > 0; i--);
  printf("Testing %d\n", count); //Counter for debug purposes
 }
#endif

#ifdef RATE_TEST
 int i;
 *ptrdir2 = 0b00100000; //Set pin 2.5 to output direction

  while(1){
   *ptrout2 ^= 0b00100000; //Toggle pin 2.5 output
   for(i=0;i<RATE;i++); //Delay with predefined iterations
  }
#endif
```

```
#ifdef FREQ_TEST
  int i;
  *ptrdir1 = 0b10000000; //Set pin 1.7 to output direction

  while(1){
   *ptrout1 ^= 0b10000000; //Toggle pin 1.7 output
   for(i=0;i<RATE;i++); //Delay with predefined iterations
  }
#endif

#ifdef DUTY_CYCLE_FUNC

  *ptrdir2 = 0b00100000; //Set pin 2.5 to output direction
  configure_duty((uint8_t)DUTY_CYCLE,(uint16_t)RATE, ptrout2); //Call configure_duty function

#endif

#ifdef SIZE
 int size = sizeof(char);
 printf("Size of a char: %d \n", size);

 size = sizeof(int);
 printf("Size of an int: %d \n", size);

 size = sizeof(short);
 printf("Size of a short: %d \n", size);

 size = sizeof(long);
 printf("Size of a long: %d \n", size);

 size = sizeof(float);
 printf("Size of a float: %d \n", size);

 size = sizeof(double);
 printf("Size of double: %d \n", size);

 size = sizeof(long double);
 printf("Size of a long double: %d \n", size);

 size = sizeof(int8_t);
 printf("Size of an int8_t: %d \n", size);

 size = sizeof(int16_t);
 printf("Size of an int16_t: %d \n", size);

 size = sizeof(int32_t);
 printf("Size of an int32_t: %d \n", size);

 size = sizeof(int64_t);
 printf("Size of an int64_t: %d \n", size);

 size = sizeof(uint8_t);
 printf("Size of an uint8_t: %d \n", size);

 size = sizeof(uint16_t);
 printf("Size of an uint16_t: %d \n", size);
```

```
110
111 size = sizeof(int32_t);
112 printf("Size of an uint32_t: %d \n", size);
113
114 size = sizeof(uint64_t);
115 printf("Size of an uint64_t: %d \n", size);
116
117 size = sizeof(uint8_t*);
118 printf("Size of an uint8_t*: %d \n", size);
119
120 size = sizeof(char*);
121 printf("Size of an char*: %d \n", size);
122
123 size = sizeof(int*);
124 printf("Size of an int*: %d \n", size);
125
126 size = sizeof(unsigned int*);
127 printf("Size of an unsigned int*: %d \n", size);
128
129 size = sizeof(uint16_t*);
130 printf("Size of an uint16_t*: %d \n", size);
131 #endif
132 }
133
134 #ifdef FUNC
135  void configure_duty(uint8_t dutyCycle,uint16_t rate, uint8_t *ptrout2) {
136      uint16_t iterations = ((rate*2) * dutyCycle)/100; //Calculation to offset delay by duty cycle
137      uint16_t i,j;
138      while(1){
139         *ptrout2 = 0b00100000; //Set pin 2.5 output to high
140         for(i=0;i<iterations;i++); //Delay for high output based on duty cycle calculation
141         *ptrout2 = 0b00000000; //Set pin 2.5 output to low
142         for(j=0;j<((rate*2)-iterations);j++); //Delay for remaining period
143       }
144  }
145 #endif
```