Luke Jaffe
Microprocessor-Based Design
Due 02/17/16

Pre-Lab 3

FreeRTOS

Pre 3.1) What is the resulting queueing policy when using the FreeRTOS queue with xQueueSend(), and xQueueReceive()?

This results in a First-In First-Out (FIFO) queueing policy.

Pre 3.2) What is the resulting queueing policy when using the FreeRTOS queue with xQueueSendToFront(), and xQueueReceive()?

This results in a Last-In First-Out (LIFO) queueing policy.

Pre 3.3) Under which conditions does a call to xQueueSend() block (i.e. does not terminate immediately)?

A call to xQueueSend() blocks if the queue is full.

Pre 3.4) What is the ISR safe alternative to send something to a queue from an ISR? Also reflect on why a separate queue send is needed.

The ISR safe alternative to send something to a queue from an ISR is the xQueueSendFromISR() function. When tasks not involved with ISRs want to exchange messages, they may do so using xQueueSend() since blocking will not cause a catastrophe. In fact, it may be desirable for some queues to be blocked in sending/receiving from queues, if they must send/receive before returning. Further, the xTicksToWait parameter allows a programmer to configure how long the task should bock waiting for space to become available in the queue. In this way, the delay can be configured to suit the priority of a given task.

Pre 3.5) Under which conditions does a call to xQueueReceive() block (i.e. does not return immediately)?

A call to xQueueReceive() blocks if the queue is empty.

Pre 3.6) For lab 3.3, the receive command should not block at all even if no elements are in the queue. Outline how to realize this.

Using the xQueueReceive() function, by setting the xTicksToWait parameter to 0, the function will return immediately if the queue is empty (and therefore never block).

C Programming

Pre 3.7) Compare keyword static can appear in different contexts. What is the meaning of defining a function local variable as static?

When a function local variable is defined as static, its value is retained between calls to the function.

PWM Generation

Pre 3.8) See Figure 2 in this assignment showing an example PWM output for value 5 in Table 1. What is the bit pattern (i.e. value for LD2, LD1, LD0) for each time step (0, 1, 2, 3) in Table 1?

Starting with LD0, each LED increases its duty cycle by 25% per time slice till 100%, then decreases in the same fashion back to 0%. The pattern begins for the next LED when its brightness reaches 75% on the decline i.e. when LED0 has brightness 75% (decreased from 100%), LED1 has brightness 25% (increased from 0%).

Pre 3.9) Table 1 defines the mapping between a value (0, 1, 2, …, 32) and individual LED brightness. The brightness will be controlled through PWM as described in the assignment (Lab 3.3). Write a function that computes the bit pattern for each time step (for all LEDs).

Below are the three iterations of my function (to produce something relatively terse). The key step was in finding the pattern which repeated every 4 values. The first attempt explicitly creates if else statements for each element of the pattern and assigns the necessary values to patternArray. The final version of the function (getPWMBitPattern3) is fully functional, short, and safe.

```
#include <stdio.h>
#include <stdlib.h>

// First attempt at writing the function, has a negative shift left which
produces undefined behavior
void getPWMBitPattern1(unsigned int value, unsigned char patternArray[4])
{
    int shift = (value/4) - 1;
    if (value % 4 == 0)
    {
        patternArray[0] = 0x01 << shift;
        patternArray[1] = 0x01 << shift;
        patternArray[2] = 0x01 << shift;
        patternArray[3] = 0x01 << shift;
    }
    else if (value % 4 == 1)
    {
        patternArray[0] = 0x03 << shift;
        patternArray[1] = 0x01 << shift;
        patternArray[2] = 0x01 << shift;
        patternArray[3] = 0x00 << shift;
    }
    else if (value % 4 == 2)
    {
        patternArray[0] = 0x03 << shift;
        patternArray[1] = 0x03 << shift;
```

```
        patternArray[2] = 0x00 << shift;
        patternArray[3] = 0x00 << shift;
    }
    else if (value % 4 == 3)
    {
        patternArray[0] = 0x03 << shift;
        patternArray[1] = 0x02 << shift;
        patternArray[2] = 0x02 << shift;
        patternArray[3] = 0x00 << shift;
    }
}

// Second attempt, pattern is condensed to shorten the code
void getPWMBitPattern2(unsigned int value, unsigned char patternArray[4])
{
    int s = (value / 4) - 1;
    unsigned int p = 2 * (value % 4);
    unsigned char r0 = 0b11111101;
    unsigned char r1 = 0b10110101;
    unsigned char r2 = 0b10000101;
    unsigned char r3 = 0b00000001;
    patternArray[0] = shift(((r0 >> p) & 0b11), s);
    patternArray[1] = shift(((r1 >> p) & 0b11), s);
    patternArray[2] = shift(((r2 >> p) & 0b11), s);
    patternArray[3] = shift(((r3 >> p) & 0b11), s);
}

// This function prevents a negative shift
unsigned char shift(unsigned char val, int mag)
{
    if (mag < 0)
        return val >> abs(mag);
    else
        return val << mag;
}

// Shorter version, uses safe shift function to avoid negative left shift
void getPWMBitPattern3(unsigned int value, unsigned char patternArray[4])
{
    int s = (value / 4) - 1;
    unsigned int p = 2 * (value % 4);
    patternArray[0] = shift(((0b11111101 >> p) & 0b11), s);
    patternArray[1] = shift(((0b10110101 >> p) & 0b11), s);
```

```
    patternArray[2] = shift(((0b10000101 >> p) & 0b11), s);
    patternArray[3] = shift(((0b00000001 >> p) & 0b11), s);
}

// This iterates through all values in the table to test the function (it
works)
int main()
{
    int value = 9;
    unsigned char patternArray[4];

    int i,j;
    for (j = 0; j <= 32; j++)
    {
        printf("\nValue: %d\n", j);
        getPWMBitPattern3(j, patternArray);
        for (i = 0; i < 4; i++)
            printf("%d: %d\n", i, patternArray[i]);
    }

    return 0;
}
```

Pre 3.10) Show the code to only turn on LD0 but leave all other bits untouched (using the appropriate MASK_DATA_*_* MMR).

We need to set the upper half of the register to ~(0x01 << 16) so that pin 1 is asserted, then we set the lower half to 0x01 so the bit it set high:

```
MASK_DATA_2_LSW = (~(0x01 << 16) | 0x01)
```

Pre 3.11) Extend function from Pre 3.9 to only modify GPIO pins when needed

For a given value, the function which sets the data register will cycle through the four elements of patternArray during one period. Therefore, we know the bits set in the previous iteration, and check which bits have changed with the XOR operation. Then, we invert these bits to create the mask, since 0 bits will cause a set. Since we assume the LEDs are off at the beginning of the period, the mask bits for the first element of patternArray are just the data bits inverted. The data bits will be what we calculated before. Finally, we shift the mask bits left 16 and OR with the data bits to get the value needed for MASK_DATA_2_LSW.

```
#include <stdio.h>
#include <stdlib.h>

unsigned char shift(unsigned char val, int mag)
{
    if (mag < 0)
        return val >> abs(mag);
    else
        return val << mag;
}

void getPWMChangeBitPattern(unsigned int value, unsigned int patternArray[4])
{
    int s = (value / 4) - 1;
    unsigned int p = 2 * (value % 4);

    unsigned int d0 = shift(((0b11111101 >> p) & 0b11), s);
    unsigned int d1 = shift(((0b10110101 >> p) & 0b11), s);
    unsigned int d2 = shift(((0b10000101 >> p) & 0b11), s);
    unsigned int d3 = shift(((0b00000001 >> p) & 0b11), s);

    // assume all LEDs are off at the beginning of a period
    unsigned int m0 = (~d0 << 16) & 0xFFFF0000;
    unsigned int m1 = (~(d1 ^ d0) << 16) & 0xFFFF0000;
    unsigned int m2 = (~(d2 ^ d1) << 16) & 0xFFFF0000;
    unsigned int m3 = (~(d3 ^ d2) << 16) & 0xFFFF0000;

    patternArray[0] = m0 | d0;
    patternArray[1] = m1 | d1;
    patternArray[2] = m2 | d2;
    patternArray[3] = m3 | d3;
}

int main()
{
    unsigned int value = 9;
    unsigned int patternArray[4];

    int i,j;
    for (j = 0; j <= 32; j++)
    {
        printf("\nValue: %d\n", j);
        getPWMChangeBitPattern(j, patternArray);
```

```
        for (i = 0; i < 4; i++)
            printf("%d: %x\n", i, patternArray[i]);
    }

    return 0;
}
```