Project 2: Digital I/O
MAE 5483
Charles O’Neill
1 October 2004

Introduction

This project’s purpose is to become familiar with digital input and output on a PIC micro-controller. Five programs demonstrate: digital LED output, digital button input, button debouncing, interrupts, and random number generation.

The micro-controller is a 28 pin DIP PIC16F876 manufactured by Microchip. The PIC voltage input is +5 volts DC via a µA7805 dc/dc voltage regulator. The compiler is the CCS C compiler (v. 3.207) for 14 bit PIC chips. C compilation occurs on a x86 based PC. Data transfer between the PIC and the PC is through a 9 pin serial cable. All programming and experiments were performed by Charles O’Neill.

Figure 1 shows the circuit schematic for this project. The red and yellow LEDs are active low and connected to pins C0 and C1. The green LED is active high and connected to pin C2. 470 Ohm resistors restrict the current to a maximum of 10 mA, within the PIC’s 25 mA maximum. The LED’s voltage drop improves the margin by even more. The external button is connected to pin B0, the external interrupt pin. Pushing the single pole momentary button shorts pin B0 to ground. Otherwise, a hold-up resistor maintains about 5 volts on pin B0. All 5 programs use this circuit.

![Figure 1: Schematic](image)
1 LED Binary Counting

The led_count.c program demonstrates digital output to Light Emitting Diodes (LEDs). The C code is given in the led_count.c Code Listings section (p.10). The objective is to sequentially count the numbers 0–7 on LEDs.

1.1 Main Loop

The heart of the led_count program is a while loop with a LED output function call. The loop continuously sets the LEDs based on the current Number value.

```c
while (1) {
    binary_led( Number ); // Timer0 Advances the Number
}
```

An internal interrupt timer is used for the delay between LED updates. The internal timer0 interrupts at 2 Hz to update the integer Number. The interrupt function configuration is based on the class notes.

```c
#define rtcc

void led_flipper(void) {
    if (--int_count == 0) {
        Number++; // Add to Number
        int_count = INTS_PER_SECOND/FREQUENCY; // Wait
    }
}
```

Using an internal timer requires calibration. A define statement gives the calibration constant INTS_PER_SECOND as 76.

\[
\text{INTS\_PER\_SECOND} = \frac{\text{Clock Rate}}{\text{Cycles per Op}} \cdot \frac{1}{\text{RTCC\_DIV\_256}} \cdot \frac{1}{\text{Max Int}}
\]

\[
= \frac{20,000,000}{4} \cdot \frac{1}{256} \cdot \frac{1}{256} = 76.29 \approx 76
\]

Eventually, the integer Number will overflow; however, overflow will not be significant because the LEDs only output 3 bits. Additionally, the integer overflow from 255 to 0 is equivalent to a 3bit overflow from 7 to 0.

1.2 LED Output

The digital output routine sets the LEDs based on the integer number. Because the circuit contains active low (Red, Yellow) and active high (Green) connected LEDs, an output routine for arbitrary color placement suggested using an inelegant solution. Each bit is picked-off with an AND mask. Both LED On and LED Off branches are needed. The routine is:

```c
/* Binary Encoded LED Output Routine */
```
void binary_led(int number) {
    /* RED LED state and Location */
    if (MASK_RED & number) {
        output_low(RED); // LED On
    } else {
        output_high(RED); // LED Off
    }

    /* YELLOW LED state and Location */
    if (MASK_YELLOW & number) {
        output_low(YELLOW); // LED On
    } else {
        output_high(YELLOW); // LED Off
    }

    /* GREEN LED State and Location */
    if (MASK_GREEN & number) {
        output_high(GREEN); // LED On
    } else {
        output_low(GREEN); // LED Off
    }
}

This output routine is general with respect to LED location and color, but poor in efficiency and expandability. Expanding the output to 8 outputs requires 8 if else statements. This routine does allow for simultaneous RS232 I/O on pin C6 and C7.

1.3 LED Output Routine Improvements

Homogeneous LED connections are much easier. For example, if all LEDs are continuously connected as active low, then the binary_led routine simplifies to:

```c
#define MASK_ALL 0x07
void binary_led(int number){
    output_c(~number & MASK_ALL);
}
```

The runtime savings is large: the revised binary_led routine has 11 machine operations, the original binaryled routine has 48 machine operations.

Additional masks for the active high bits allows for a binary_led routine with 12 operations. The routine begins by masking number with possible LED locations MASK_ALL. A bitwise complement converts all bits to active low. An XOR with a mask of the active high ports flips those specific bits to active high.

```c
#define MASK_ALL 0x07
#define MASK_ACTIVE_HIGH MASK_GREEN
```
void binary_led(int number){
    output_c( (˜number ˆMASK_ACTIVE_HIGH) & MASK_ALL);
}

This improved routine will be used for programs 1–4. The disadvantage of this routine is that the port locations must be continuous and the port cannot be shared with RS232 I/O.

2 Basic Button

This program investigates using external buttons for input. The C code is given in the led_button.c Code Listings section (p.11). The objective is to sequentially count the numbers 0–7 with about one advance per button press.

Operationally, the program is simple: poll for button B0 inside a while loop. The while loop waits for a button press. Next, another while loop waits for the button release. Finally, the integer number is advanced and displayed.

```c
while(1){
    while(input(BUTTON)); // While Wait for Button Press
    while(!input(BUTTON)); // While Wait for Button Release
    binary_led(++Number); // Advance Number
}
```

Unfortunately, button bounce was a rare event with this and even sloppier button handling codes. Eventually, a jumper wire across the open switch was required to consistently create button bounce. Bounce numbers become increasingly rare: 2 bounces are common, 4 bounces are rarely seen.

3 Debounced Button

This program experiments with button bounce. The C code is given in the led_button.c Code Listings section (p.13). The objective is to sequentially count the numbers 0–7 with only one advance per button press.

Adding delays after a known change in button state helps to alleviate button bounce. The following code uses a delay of DELAY milliseconds:

```c
while(1){
    while(input(BUTTON)); // While Wait for Button Press
    delay_ms(DELAY);
    while(!input(BUTTON)); // While Wait for Button Release
    delay_ms(DELAY);
    binary_led(++Number); // Advance Number
}
```

The following table shows a qualitative observation of the relationship between delays and the occurrence of bouncing.
<table>
<thead>
<tr>
<th>Delay [ms]</th>
<th>Occurrence</th>
<th>Noticeable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>common</td>
<td>no</td>
</tr>
<tr>
<td>20</td>
<td>common</td>
<td>no</td>
</tr>
<tr>
<td>40</td>
<td>rare</td>
<td>no</td>
</tr>
<tr>
<td>100</td>
<td>never</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Adding moderate delays —≈ 50ms per change in state— appears to remove bouncing without adversely inconveniencing the user.

4 External Interrupt

This program investigates external interrupts. The C code is given in the led\button.c Code Listings section (p.14).

The button is connected to pin B0, the external interrupt pin. The code required for an external interrupt is:

```c
#include <int.h>

void button_int(void){
    button_trigger = 1;    // Change Button State
}

enable_interrufts(INT_EXT);    // Port B Interrupt
enable_interrufts(GLOBAL);    // Turn on Interrupts
```

Interestingly, the compiler lists the available Interrupts for the 16F876 as:

- RTCC Timer 0 overflow (using RTCC name)
- RB Port B any change on B4-B7
- EXT External interrupt
- AD Analog to digital conversion complete
- TBE RS232 transmit buffer empty
- RDA RS232 receive data available
- TIMER1 Timer 1 overflow
- TIMER2 Timer 2 overflow
- CCP1 Capture or Compare on unit 1
- CCP2 Capture or Compare on unit 2
- SSP SPI or I2C activity
- BUSCOL Bus collision
- EEPROM Write complete
- TIMER0 Timer 0 overflow (using TIMER0 name)

INT_EXT is the only dedicated external interrupt on this chip; however, INT_RB can be used for this purpose. Adding an RB interrupt allowed for a touch-sensitive switch. Lightly touching a jumper wire connected to B7, interrupts the processor, flashes 100 times, and sleeps.

```c
/*
 * Interrupt on Port B pins 4 through 7
*/
```
Returning to INT.EXT, the program works correctly: one advance per button press. Debouncing the button was not needed; however, intentionally motion of the jumper wire could cause bouncing. Plus, the button handling code is cleaner than with raw port polling. Because the program is not blindly polling, the program is free to perform other tasks.

5 Digital Dice

This program investigates random number generation to digitally simulate dice. Light Emitting Diodes (LEDs) are used to binary encode the numbers (pips). A momentary button rolls the dice. The objective is to experiment with a random number generator.

5.1 Program Description

The C code is given in the dice.c Code Listings section (p.16). The die numbers are output on the LEDs. A internal timer interrupt continuously flips views between die #1 and die #2 at 0.5 second intervals using the flip switch.

\[
\text{binary\_led}\left(\\text{flip}\? \text{Die1: Die2 }\right);
\]

An external interrupt on button B0 rolls the dice. Assuming fair dice means an equal probability of numbers between 1 and 6. The rand() function with a srand() seed is used to generate random numbers. The program keeps 3 bits —for 8 total values— and rejects the numbers 0 and 7.

\[
\text{do}
\{
\quad \text{die1=\text{rand}() \& \text{MASKRAND};}
\}
\text{while(\text{Die1>MAXRAND} || \text{Die1<MINRAND});}
\]

\(^1\)Dice (plural) because a dictionary says: die: a small cube (as of ivory, bone, or plastic) marked distinctively on each face with one to six spots and used in pairs in various games and in gambling by being shaken and thrown to come to rest at random on a flat surface.
Keeping only 3 bits and then keeping only 6 values *theoretically* ensures equal number probability regardless of the random number generator’s range.

This dice simulator requires Money to operate — one dollar per roll! No Money... No Dice. When Money is zero, the LEDs strobe as an hypnotic attention getter. After all, attracting new customers\(^2\) is vital. A keyboard money manager credits the Wallet with every $ key press. The 0 key zeros out the Wallet. All other keys show the current balance. For long term storage, an EEPROM Wallet keeps track of Money. Unplugging the chip does not reset the Money balance. EEPROM reading and writing is through the read_eeprom() and write_eeprom() functions.

### 5.2 Operational Output

A typical dice session follows. The Wallet account originally has $0, but 3 dollars are added. Two dice rolls are made with the button, and the account is zeroed out. Next 5 dollars are added and 5 rolls are made. After the 3rd roll, the remaining Money is viewed. At the attempted 6th roll, the LEDs strobe.

```
Money: $0 $  
Money: $1 $  
Money: $2 $  
Money: $3 $  
Dice: 1 and 4  
Dice: 3 and 5   0  
Money: $0 $  
Money: $1 $  
Money: $2 $  
Money: $3 $  
Money: $4 $  
Money: $5 $  
Dice: 3 and 4  
Dice: 5 and 2  
Dice: 6 and 6   i  
Money: $2 $  
Dice: 5 and 4   i  
Money: $1 $  
Dice: 2 and 1   i  
Money: $0 $  
```

### 5.3 Random Number Distribution

The objective of this section is characterizing the random number generation in the dice.c program. The random number generator uses rand() as the number generator and srand() as the seed.

\(^2\)victims!
The first test is determining if the random number generator outputs and identical sequence for an identical seed. Figure 2 shows two equivalent rand() sequences, points and circles, with a seed of 1. The PICC manual says, “If srand is then called with same seed value, the sequence of random numbers shall be repeated.”

![Figure 2: Sequence of rand() with SEED=1](image)

The probability of each number 1–6 is equal, 16.6%, if the distribution is truly random. Figure 3 shows the convergence in each number’s probability as the number of samples is increased. A line denotes the truly random distribution of 16.6%. Different seeds appear to follow the trends seen above. The rand() number sequence appears to be mostly but not exactly random.

![Figure 3: Probability Convergence versus Seed](image)

The covariance function measures the connectivity of a sequence in time. For random processes, the covariance function should be the delta function at the sequence cycle length. From Matlab, Figure 4 show the covariance for a seed of 128. The covariance has a spike at the sample length 4000 as expected. The noise indicates a not-quite random process, but it appears that the random number generator did not repeat within 4000 points.
Conclusions

Five programs were created to become familiar with Digital Input and Output. The programs experimented with: digital LED output, digital button input, button debouncing, interrupts, and random number generation. Comparisons were made between various I/O routines and methods.
Code Listings

led_count.c

/*
 *    led_count.c    -- LED counting
 *    Outputs Binary Encoded LED Lights Sequentially Counting 0 to 7
 *    Charles O'Neill
 *    MAE 5483
 *    Project 2.1
 */

#include <16F876.h>
#include <stdlib.h>
#define delay(clock=2000000)
#define fuses HS, NOWDT
#define rs232(baud=19200, parity=N, xmit=PIN_C6, rcv=PIN_C7)
#define INTS_PER_SECOND 76 // Internal Timer Constant
    // 2E6 / (256 * 256 * 4)

#define RED 0x01 // active low at C0
#define YELLOW 0x02 // active low at C1
#define GREEN 0x04 // active high at C2
#define MASK_ALL 0x07 // All Output Ports
#define MASK_ACTIVE_HIGH GREEN // Active High Ports

#define FREQUENCY 2 // Number Frequency (Hz)

void binary_led(int number);

int Number=0;
int int_count=INTS_PER_SECOND/FREQUENCY; // Internal Timer
```c
// Timer Interrupt

#include <int.h>

void led_flipper(void) {
    if (--int_count == 0) {
        Number++;
        int_count = INTS_PER_SECOND/FREQUENCY;
    }
}

// Main Program for dice

void main() {

    // Initialize Interrupts
    setup_timer0(RTCC_INTERNAL | RTCC_DIV_256);
    enable_interruptions(INT_RTCC);
    enable_interruptions(GLOBAL);

    // Set the PIC's TriState I/O *
    binary_led(0);
    set_tris_c(0x08);

    // Loop Continuously *
    while(1) {
        // Set the LED Output *
        binary_led(Number);
    }

    // Binary Encoded LED Output Routine

    void binary_led(int number){
        // Complement to convert to active low. XOR with active high mask to flip
        // active high bits. Mask all active port bits.
        output_c (~number & MASK_ACTIVE_HIGH) & MASK_ALL;
    }

    led_button.c

    /*
     * led_button.c --- LED counting with an External Button
     *
     * Outputs Binary Encoded LED Lights Sequentially Counting 0 to 7
     * with an Input from Port B.
     */

    */
```
/*
 * Charles O' Neill
 * MAE 5483
 * Project 2.2
 */

/***************************************************************************/
* Default PIC Initialization
/***************************************************************************/
#include <16F876.h>
#define delay(clock=20000000)
#define HS_NOWDT
#define rs232(baud=19200, parity=N, xmit=PIN_C6, rcv=PIN_C7)

/***************************************************************************/
* Define LED Locations
/***************************************************************************/
#define RED 0x01
#define YELLOW 0x02
#define GREEN 0x04
#define MASK_ALL 0x07
#define MASK_ACTIVE_HIGH GREEN

/***************************************************************************/
* Define Button Location
/***************************************************************************/
#define BUTTON_PIN B0

/***************************************************************************/
* Function Prototypes
/***************************************************************************/
void binary_led(int number);

/***************************************************************************/
* Main Program for dice
/***************************************************************************/
void main(){

    /* Variables */
    int Number=0;

    /* Set the PIC's TriState I/O */
    binary_led(0);
    set_tris_b(0xff);
    set_tris_c(0x08);

    /* Loop Continuously */
    while(1){
        while(input(BUTTON));
        while(!input(BUTTON));
    }
}
// Advance Number

/* Binary Encoded LED Output Routine */

void binary_led(int number){
    // Complement to convert to active low. XOR with active high mask to flip
    // active high bits. Mask all active port bits.
    output_c (~number ^ MASK_ACTIVE_HIGH) & MASK_ALL;
}

led_button_debounce.c

/*
* led_button_debounce.c --- LED counting with an External Button
*
* Outputs Binary Encoded LED Lights Sequentially Counting 0 to 7
* with an Input from Port B.
*
* Charles O'Neill
* MAE 5483
* Project 2.3
*/

/* Default PIC Initialization */
.include <16F876.h>
#use delay (clock=20000000)
#fuses HS, NOWDT
#use rs232 (baud=19200, parity=N, xmit=PIN_C6, rcv=PIN_C7)

/*
* Define LED Locations */
#define RED 0x01 // active low at C0
#define YELLOW 0x02 // active low at C1
#define GREEN 0x04 // active high at C2
#define MASK_ALL 0x07 // All Output Ports
#define MASK_ACTIVE_HIGH GREEN // Active High Ports

/*
* Define Button Location */
#define BUTTON PIN_B0
#define DELAY 100

/ * Function Prototypes */
void  binary_led ( int  number );

/* Global Variables */
int  Number = 0 ;

/* Main Program for dice */
void  main (){
    /* Set the PIC’s TriState I/O */
    binary_led ( 0 );  // No Initial Output
    set_tris_b ( 0xff );  // Port B
    set_tris_c ( 0xc8 );  // Port C
    
    /* Loop Continuously */
    while ( 1 ){
        while ( input ( BUTTON ) );  // While Wait for Button Press
        delay_ms ( DELAY );
        while ( ! input ( BUTTON ) );  // While Wait for Button Release
        delay_ms ( DELAY );
        binary_led ( ++Number );  // Advance Number
    }
}

/* Binary Encoded LED Output Routine */
void  binary_led ( int  number ){
    // Complement to convert to active low. XOR with active high mask to flip
    // active high bits. Mask all active port bits.
    output_c ( ( ~ number ˆ MASK_ACTIVE_HIGH ) & MASK_ALL );
}

led_interrupt.c

/*  * led_interrupt.c —— LED counter with External Interrupt Advance  *
*  Outputs Binary Encoded LED Lights  *
*  Charles O’Neill  *
*  MAE 5483  *
Project 2.4

* Default PIC Initialization

```c
#include <16F876.h>
#include <stdlib.h>

use delay (clock=20000000)

use rs232 (baud=19200, parity=N, xmit=PIN_C6, rcv=PIN_C7)
```

* Define LED Locations

```c
#define RED 0x01 // active low at C0
#define YELLOW 0x02 // active low at C1
#define GREEN 0x04 // active high at C2
#define MASK_ALL 0x07 // All Output Ports
#define MASK_ACTIVE_HIGH GREEN // Active High Ports
```

* Function Prototypes

```c
void binary_led(int number);
```

* Global Variables

```c
short button_trigger = 0; // External Button Interrupt
```

* External Interrupt

```c
#define ext

void button_int(void){
    button_trigger = 1; // Change Button State
}
```

* Interrupt on Port B pins 4 through 7

```c
#define rb

void button_halt(void){
    int i;
    for(i=0; i<100; i++){
        binary_led(7); // Flash the LEDs
delay_ms(50);
        binary_led(0);
delay_ms(50);
    }
```
Main Program

```c
void main()
{
    /* Variables */
    int Number=0;

    /* Initialize Interrupts */
    enable_interrupts(INT_EXT);   // Port B Interrupt
    enable_interrupts(INT_RB);   // Port B 4-7 Interrupt
    enable_interrupts(GLOBAL);   // Turn on Interrupts

    /* Set the PIC's TriState I/O */
    binary_led(0);                 // No Initial Output
    set_tris_c(0xc0);             // Port C

    /* Loop Continuously */
    while(1){
        /* Advance LED Count if Button is Triggered*/
        if(button_trigger){
            binary_led(+Number);    // Advance the LED Count
            button_trigger =0;      // Reset Button Trigger
        }
    }
}

/* Binary Encoded LED Output Routine */
void binary_led(int number){
    // Complement to convert to active low. XOR with active high mask to flip
    // active high bits. Mask all active port bits.
    output_c (~number ^ MASK_ACTIVE_HIGH & MASK_ALL);
}
```

dice.c

/*
 * dice.c --- Virtual die simulator
 *
 * Outputs Binary Encoded LED Lights to Simulate a Pair of Dice
 */
/*
 *  Charles O' Neill
 *  MAE 5483
 *  Project 2.4
 */

/* Default PIC Initialization */
#include <16F876.h>
#include <stdlib.h>
#define delay (clock=20000000)
#define fuse HS_NOWDT
#define rs232 (baud=19200, parity=N, xmit=PIN_C6, rcv=PIN_C7)
#define INTS_PER_SECOND 76 // Internal Timer Constant
                // 2E6 / (256*256*4)

/* Define LED Locations */
#define RED PIN_C0 // active low at C0
#define YELLOW PIN_C1 // active low at C1
#define GREEN PIN_C2 // active high at C2
#define MASK_RED 0x04 // RED at bit 2
#define MASK_YELLOW 0x02 // RED at bit 1
#define MASK_GREEN 0x01 // RED at bit 0

/* Random Number Range */
#define SEED 4 // Random Number Seed and Limits
#define MIN_RAND 1 // see srand() and rand()
#define MAX_RAND 6
#define MASK_RAND 0x07 // Mask off 3 bits 0b00000111

/* Define WALLET EEPROM Location */
#define WALLET 10

/* Function Prototypes */
void binary_led(int number);

/* Global Variables */
short button_trigger =0; // External Button Interrupt
short flip =0; // LED Flip State
```c
int int_count = INTS_PER_SECOND / 2;  // Internal Timer

/* External Interrupt */

#define int_ext
void button_int(void){
    button_trigger = 1;  // Change Button State
}

/* Timer Interrupt */

#define int_rtcc
void led_flipper(void){
    if(--int_count==0){
        flip ^= 1;  // Flip LED State
        int_count = INTS_PER_SECOND / 2;  // Wait
    }
}

/* Main Program for dice */

void main(){
    /* Variables */
    int Die1=0, Die2=0, Money, Strobe_Rate;
    char Currency_Unit;

    /* Initialize Interrupts */
    setup_timer_0(RTCC_INTERNAL | RTCC_DIV_256);  // Timer0
    enable_interrupts(INT_RTCC);  // Timer0
    enable_interrupts(INT_EXT);  // Port B Interrupt
    enable_interrupts(GLOBAL);  // Turn on Interrupts

    /* Initialize Random Number Generator */
    srand(SEED);

    /* Set the PIC’s TriState I/O */
    binary_led(0);  // No Initial Output
    set_tris_b(0x0fe);  // Port B
    set_tris_c(0x0c8);  // Port C

    /* Load Money from Wallet */
    Money = read_eeprom(WALLET);
    printf("\n\rMoney: $%d", Money);

    /* Loop Continuously */
    while(1){
```

18
/ Roll Dice with Money */
if(button_trigger & Money > 0) {
    // Roll Dice
    write_eeprom(WALLET, Money--);
    button_trigger=0;
    delay_ms(10);
}

/* Die 1: Random Number */
do{
    die1=rand() & MASKRAND;
} while(Die1 > MAXRAND || Die1 < MINRAND);
/* Die 2: Random Number */
do{
    die2=rand() & MASKRAND;
} while(Die2 > MAXRAND || Die2 < MINRAND);

printf("\n\rDice:%d and %d", Die1, Die2);

/* No Money --- No Dice Roll */
} else if(button_trigger & Money <= 0) {
    binary_led(7);
    delay_ms(35);
    binary_led(0);
    delay_ms(35);
    Die1=Die2=0;
}

/* Set the LED Output */
binary_led((flip)? Die1 : Die2);
/* Keyboard Money Manager */
if(kbhit()){
    // If Keyboard is Active
    Currency_Unit=getc();
    printf("%c", Currency_Unit);
    switch(Currency_Unit){
        case '$':
            write_eeprom(WALLET, ++Money);
            button_trigger=0;
            break;
        case '0':
            write_eeprom(WALLET, Money=0);
            break;
        default:
            break;
    }
    printf("\n\rMoney: $%d", Money);
}
}
/ * Binary Encoded LED Output Routine */

void binary_led(int number){
    /* RED LED state and Location */
    if (MASK_RED & number) {
        output_low(RED);        // LED On
    } else {
        output_high(RED);        // LED Off
    }

    /* YELLOW LED state and Location */
    if (MASK_YELLOW & number) {
        output_low(YELLOW);    // LED On
    } else {
        output_high(YELLOW);    // LED Off
    }

    /* GREEN LED State and Location */
    if (MASK_GREEN & number) {
        output_high(GREEN);    // LED On
    } else {
        output_low(GREEN);    // LED Off
    }
}