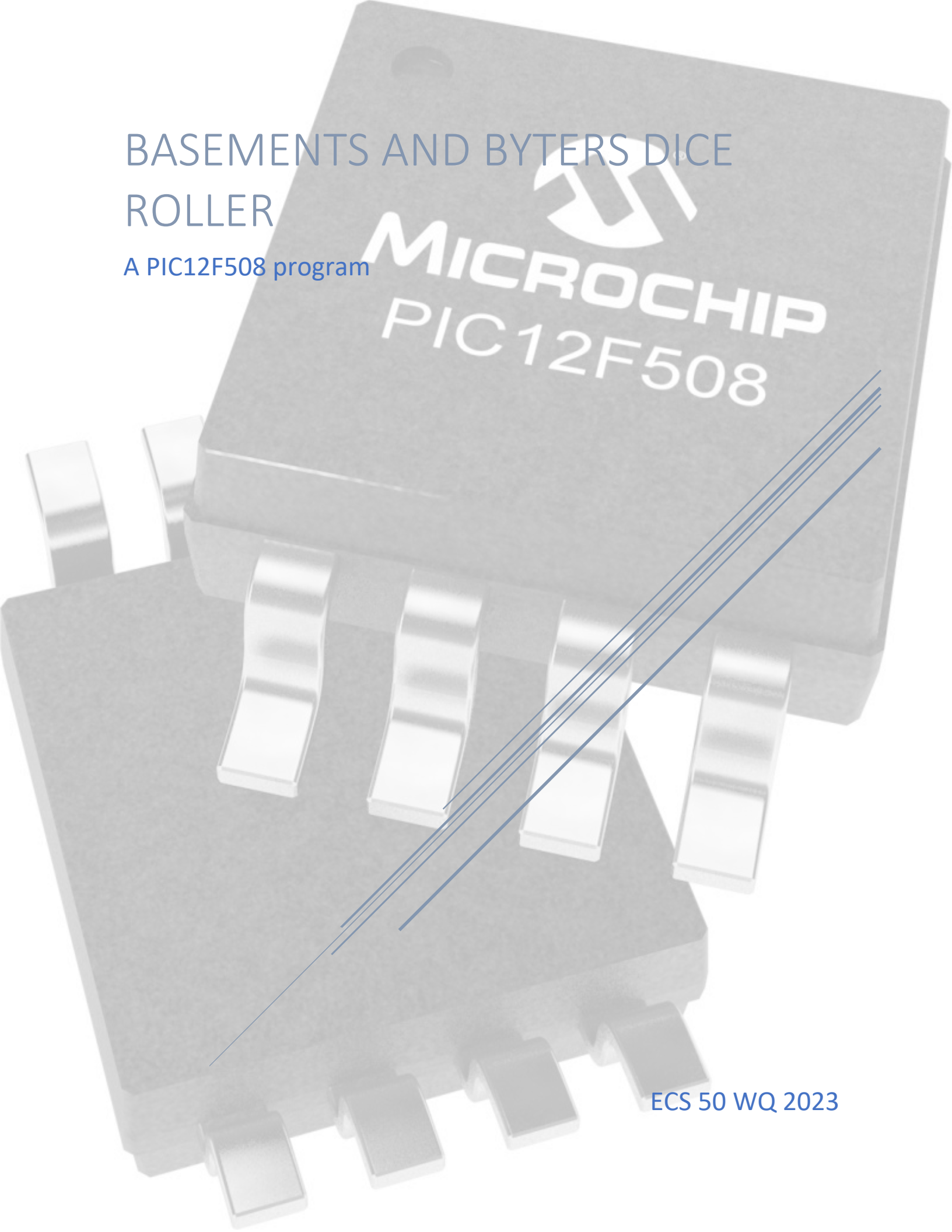
BASEMENTS AND BYTERS DICE ROLLER

A PIC12F508 program



ECS 50 WQ 2023

Introduction

Basement and Byters Dice Roller is a microcontroller program using Microchip's PIC12F508 and written in their instruction set found on page 57. Basement and Byters is a variation of the popular role-playing game Dungeons and Dragons, with similar gameplay. The dice used for BnB[1] however feature binary digits on their faces and each die has number of sides equal to a power of two. This device allows your group to digitally roll these dice—no physical dice needed! This document outlines how to use the device and gives basic knowledge of microcontrollers to those who are unfamiliar. The theory of operation and discussion section are intended for more experienced programmers/those with low-level hardware knowledge, however, these skills can be picked up easily using this website which includes the simulator that this program was built on for testing.

How to Use

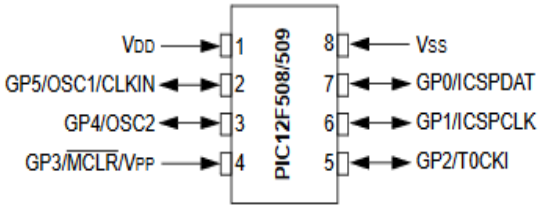


Figure 1 PIC12F508 schematic

The PIC12F508 features 8 pins with a detailed schematic in figure 1. GP0-5 are input/output[2] pins, colloquially known as I/O ports, meaning we receive data at input pins and produce output (in this case LEDs) at output pins. VDD and VSS deal with power to the microcontroller and the acronyms following the GP0-5 pins are out of scope for this document. In this program, we attach a switch to input pins GP3 and GP4 to provide input about the rolling and display the rolled results on the LEDs.

GP3 – When pressed (set), starts the random roll. The LEDs will begin blinking in an alternating pattern to indicate the roll has begun (equivalent to your hands shaking the dice prior to dropping them on the table). Press again to stop the roll and display on the LEDs. Note that BnB uses binary digits on each of its faces and we read binary digits from right to left rather than left to right. GP0's LED corresponds to the zeroth bit, GP1-the first, GP2-second, and GP5-third. Figure 2 shows an example of what a roll of 10 might look like.[3]

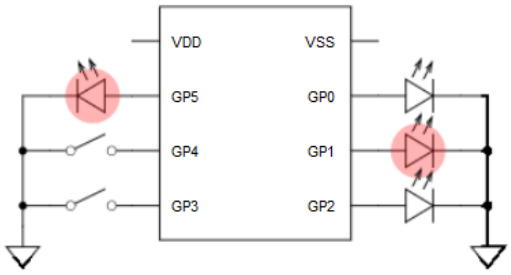


Figure 2 PIC12F508 displaying a dice roll of 10 (1010b)

GP4 – Selects method of rolling. When set, meaning switch in the up/open position and closing the current to ground[4], the device will roll 1 die with 16 sides, 0-15 (in binary), while 2 dice with 8 sides, 0-7, will be rolled when the switch is clear

[1] Basements and Byters
[2] GP3 is an input only pin in this microcontroller
[3] 1010b refers to 10's binary representation: 1010
[4] Ground in this instance is no/little voltage. Binary data is sent as voltage levels, with high voltage sent as '1's and low voltage sent as '0's. Connecting the ground sends low voltage, while closing the path to ground sends high voltage as there is an internal voltage in this microcontroller.

(down/closed position), and the sum will be outputted. Note that this does not begin the roll and is intended to be determined prior to the beginning of your roll when GP3 is pressed.

Theory of Operation

Let's describe how the program works. This section is not intended to explain what every instruction means. For that information, please refer to the datasheet linked on page 1. We initially set input at GP3 and GP4 with every other pin as output as well as load the LFSR register with an arbitrary seed in the origin. The reason we load the LFSR here is because we loop back to press_start after we display a roll and we do not want to load the initial seed again or else we would not generate a pseudorandom number. The LFSR register and inner workings will be described in detail later. We begin the loop with press_start which clears the GPIO to clear the roll we previously displayed as well as clear LEDs on startup which are initially on. We then loop until the user presses GP3 to begin rolling.

Label	Addr	Instructions (assembly)
origin	00h	movlw 00011000b
	01h	tris 6
	02h	movlw 0x79
	03h	movwf lfsr
press_start	04h	clrf GPIO
	05h	btfsc GPIO, GP3
	06h	goto press_start

Figure 3 Describes program setup

Figure 4 is more complex. We initially set GPIO to 00100010b to display the rolling sequence. Line 0Eh moves 00000101b to GPIO to reveal the alternating pattern. This comes a few operations down in order to display both parts for around the same time (give or take a couple clock cycles due to skipping instructions). Lines 09h – 0Dh generates a linear-feedback shift register. This will be discussed in more detail in the Discussion section, but in simple terms, this takes an arbitrary seed, right shifts it, then uses XOR instructions in order to get a seemingly 'random' 8-bit number. Note we bring LFSR back into the working register because we AND this 8-bit number with 111b when the user is not done rolling (when GP3 is clear). Thus, we are saving an 8-sided die after every loop of LFSR, even when GP4 is set. The reason for this will be discussed later, but it allows for easier implementation and prevents unnecessary looping. We then loop again to begin_roll to continue the LFSR sequence. When the user *is* ready to stop rolling and sets GP3, we continue to check_mode implemented below.

begin_roll	07h	movlw 00100010b
	08h	movwf GPIO
	09h	bcf STATUS,C
	0Ah	rrf lfsr, w
	0Bh	btfsc STATUS, C
	0Ch	xorlw 10111000b
	0Dh	movwf lfsr
	0Eh	movlw 00000101b
	0Fh	movwf GPIO
	10h	movf lfsr, w
end_roll	11h	btfsc GPIO, GP3
	12h	goto check_mode
save_small_die	13h	andlw 111b
	14h	movwf cur_roll
	15h	goto begin_roll

Figure 4 Describes how the device displays a rolling sequence, generates random numbers, and saves

We clear the GPIO to indicate that we are done rolling. We then either mask the lower 4 bits and save to our current roll register, or we mask the lower 3 bits and add to the current roll register which was previously filled with the previous roll of an 8-sided dice. We then display the roll simply by moving this roll to the GPIO. Remember however, we must include an inclusive-or on line 21h with 1 at the 5th bit slot because we display the 3rd bit at GP5. This is only done if the 3rd bit is set as to not display GP5 every time.

```
check_mode      16h   clrf GPIO
                17h   btfsc GPIO, GP4
                18h   goto one_die
                19h   goto second_dice
one_die         1Ah   andlw 1111b
                1Bh   movwf cur_roll
                1Ch   goto display_roll
second_dice     1Dh   andlw 111b
                1Eh   addwf cur_roll, f
display_roll    1Fh   movf cur_roll, w
                20h   btfsc cur_roll, 3
                21h   iorlw 00100000b
                22h   movwf GPIO
```

Figure 5 Describes how we display the dice depending on the mode at GP4

This device also saves the past 16 rolls into a circular buffer. This is implemented similarly to a stack with the difference being we reset the stack pointer to 0 when we reach the end. Registers FSR and INDF are useful here. We initially check if we are at the end of the stack by checking the zero bit in the STATUS register after the operation stack_ptr – stack_size. This checks if this subtraction yields a zero, thus stack_ptr is all the way through the stack and we then reset stack_ptr to 0 to get back to the top of the stack. We add stack_ptr and stack_start to get the correct address of the next spot in the stack and save it in FSR. We then move the current roll to INDF which points to the address in FSR which then stores the current roll correctly in the stack. We increment stack_ptr to progress through the stack on the next roll and wait for release of GP3 to go back to press_start and start the next roll.

```
stack           23h   movlw stack_size
                24h   subwf stack_ptr, w
                25h   btfsc STATUS, Z
                26h   movlw 0
                27h   btfsc STATUS, Z
                28h   movwf stack_ptr
                29h   movlw stack_start
                2Ah   addwf stack_ptr, w
                2Bh   movwf FSR
                2Ch   movf cur_roll, w
                2Dh   movwf INDF
                2Eh   incf stack_ptr, f
wait_for_release 2Fh  btfss GPIO, GP3
                30h   goto wait_for_release
                31h   goto press_start
```

Figure 6 Describes how we save rolls into a circular buffer and wait for user to roll once more

Discussion

LFSR

We used an implementation of LFSR created by Daryl Posnett, Professor at UC Davis. We start with an arbitrary seed, in this case 0x79. We first right shift the LFSR through the carry, with the left-most bit filled with zero. We then use XOR taps when the right-most bit (carry) is 1; the XOR taps we use are the 8th, 6th, 5th, and

Operation	7	6	5	4	3	2	1	0	Carry	Result after bit masking
Seed	0	1	1	1	1	0	0	1	0	
Right shift	0	0	1	1	1	1	0	0	1	
XOR Taps	1	0	0	0	0	1	0	0	0	4, 4
Right shift	0	1	0	0	0	0	1	0	0	

Figure 7 Demonstrates LFSR

4th bits, read right to left starting from zero. Refer to figure 7 for a demonstration with our arbitrary seed. This produces a cycle that refreshes after $2^k - 1$ times, where k is the number of bits. Since this LFSR generates an 8-bit number, this pattern repeats after 255 times. Thus, our random dice generator is not truly random, but is what we call 'pseudorandom.' The next number in the sequence can be predicted by knowing the seed, but without it, the rolls appear random in nature. Figure 8 plots the rolls we get after 4-bit masking and 3-bit masking. Note it is difficult to plot the results we get after rolling 2 dice because the user will loop through the LFSR a random number of times. However, by plotting the results we get after bit masking, we can see that each die roll is almost perfectly uniform, giving strong randomness when summing two of them. Figure 9 shows the last 16 values in the circular buffer. Note that we see repeating values from addresses 1Ah – 1Dh. This is common as the values in this example are storing every iteration of a new LFSR sequence. In reality, the user would not be pressing and releasing the switch at common intervals, thus increasing the randomness and making it difficult to predict upcoming rolls. Our implementation of 3-bit masking after every loop of LFSR makes it more difficult for the user to predict as well as prevents unnecessary looping. Attentive readers may notice that if the user is quick enough to stop the roll, it is possible to produce a sum of the current roll and 0. This however is extremely unlikely as the clock cycles on this device refresh much more quickly than on the simulator and this becomes a non-issue.

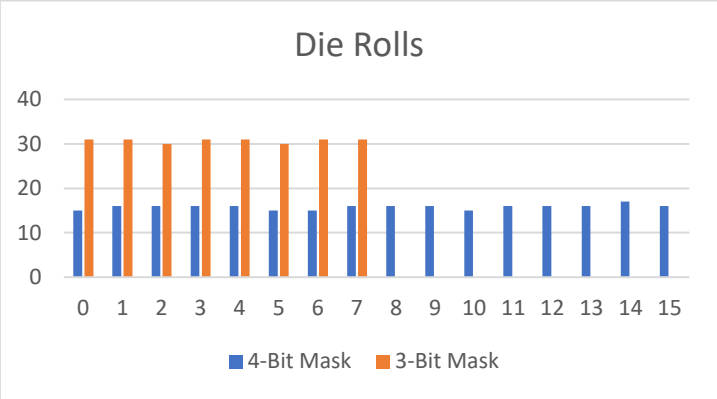


Figure 8 Plot of Die Rolls we collect after masking current LFSR numbers

10h	stack_start	04h
11h		02h
12h		01h
13h		08h
14h		04h
15h		0Ah
16h		05h
17h		02h
18h		09h
19h		04h
1Ah		0Ah
1Bh		05h
1Ch		0Ah
1Dh		05h
1Eh		02h
1Fh	stack_end	09h

Figure 9 Last 16 rolled values stored in circular buffer

Conclusion

This program is a great starter program for beginners eager to learn more about hardware and low-level programming. This program uses many paradigms in machine level code and successfully performs a task. The random numbers it generates are deterministic, however without knowing the seed, have strong randomness to its users.

Appendix

```
// Program that rolls dice for DnD variation Basements
// and Byters

// Standard constants
$f 1
$w 0
$stack_size 16

// Registers
@cur_roll 0x09
@lfsr 0x0A
@stack_ptr 0x0F
@stack_start 0x10
@stack_end 0x1f

:origin
      // Setup io ports
      movlw 00011000b
      tris 6

      // Load lfsr with a seed
      movlw 0x79
      movwf lfsr

:press_start
      // Clear LED outputs and proceed when GP3 is
      // pressed and clear cur_roll
      clrf GPIO
      btfsc GPIO, GP3
      goto press_start

:begin_roll
      // Display sequence on GPIO to inform user that
      // we are rolling
      movlw 00100010b
      movwf GPIO

      // First clear the carry so that we know that it
      // is zero. Now, shift the lfsr right putting
      // the LSB in carry and copying the lfsr into
      // the working register for more processing
      bcf STATUS,C
      rrf lfsr, w

      // If the carry (LSB) is set we want to xor the
      // lfsr with our xor taps: 8,6,5,4(0,2,3,4)
      btfsc STATUS, C
      xorlw 10111000b
      movwf lfsr // save the new lfsr

      // continue rolling display sequence
      movlw 00000101b
```

```
      movwf GPIO

      // retrieve pseudo-random 16 bit number again
      // since we moved rolling display bits above
      // into working
      movf lfsr, w

:end_roll
      // Check the mode if GP3 is set (user has
      // stopped rolling). Otherwise proceed forward
      // to save_small_die.
      btfsc GPIO, GP3
      goto check_mode

:save_small_die
      // Mask 3 bits to save 8 sided die even if user
      // doesn't intend to roll 2 8 sided dice. This
      // makes rolling 2 8 sided dice easier to
      // implement
      andlw 111b
      movwf cur_roll
      goto begin_roll

:check_mode
      // Clear GPIO to inform user that rolling has
      // stopped, then check the mode and proceed to
      // appropriate subroutine according to mode from
      // GP4
      clrf GPIO
      btfsc GPIO, GP4
      goto one_die
      goto second_dice

:one_die
      // When user wants one die, mask lower 4 bits
      // and save it so we can display properly
      andlw 1111b
      movwf cur_roll
      goto display_roll

:second_dice
      // When user wants two dice, mask lower 3 bits
      // and add with previous roll generated by LFSR
      // and by saveing 3 bit masks after every new
      // number generated from the lfsr
      andlw 111b
      addwf cur_roll, f

:display_roll
      // Because we can not bit test from working
      // register, we must read from cur_roll register
      // as well as bring to working to inclusive or
      // with 5th bit if 3rd bit is set. We then move
      // to GPIO. We must do this because GP5 is
```

```
      // outputting the 3rd bit.
      movf cur_roll, w
      btfsc cur_roll, 3
      iorlw 00100000b
      movwf GPIO

:stack
      // If stack_ptr - stack_size = 0 then we are at
      // end of stack as we have incremented the
      // pointer all the up through the stack.
      // Therefore we check zero bit of STATUS. If we
      // are at end of stack then reset stack_ptr back
      // to 0
      movlw stack_size
      subwf stack_ptr, w
      btfsc STATUS, Z
      movlw 0
      btfsc STATUS, Z
      movwf stack_ptr

      // add stack_ptr to stack_start to get correct
      // address to store current roll and store that
      // address in FSR
      movlw stack_start
      addwf stack_ptr, w
      movwf FSR

      // Store roll in correct location in circular
      // buffer by moving current roll into INDF.
      // Increment stack_ptr to continue down the
      // stack on next roll
      movf cur_roll, w
      movwf INDF
      incf stack_ptr, f

:wait_for_release
      // Wait until switch is released
      btfss GPIO, GP3
      goto wait_for_release
      goto press_start
```