Pieter-Jan.com
Creativity In Automation & More

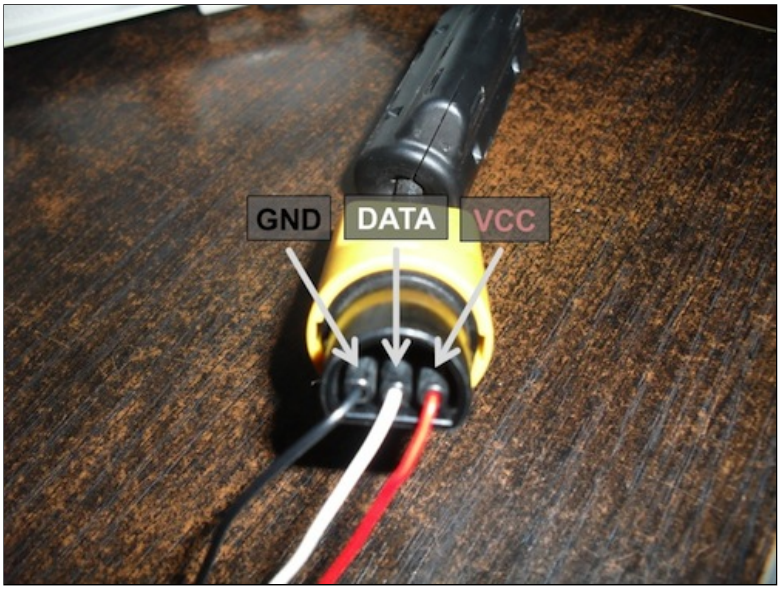
Reading Nintendo 64 controller with PIC microcontroller

Submitted by Pieter-Jan on Wed, 12/09/2012 - 08:37

I have a few old N64 controllers lying around and figured that it would be pretty cool to use them to control other things. In this article I will describe in detail every step I took to achieve this. I've used a PIC microcontroller, but it should not be too hard to port the code to any other architecture.

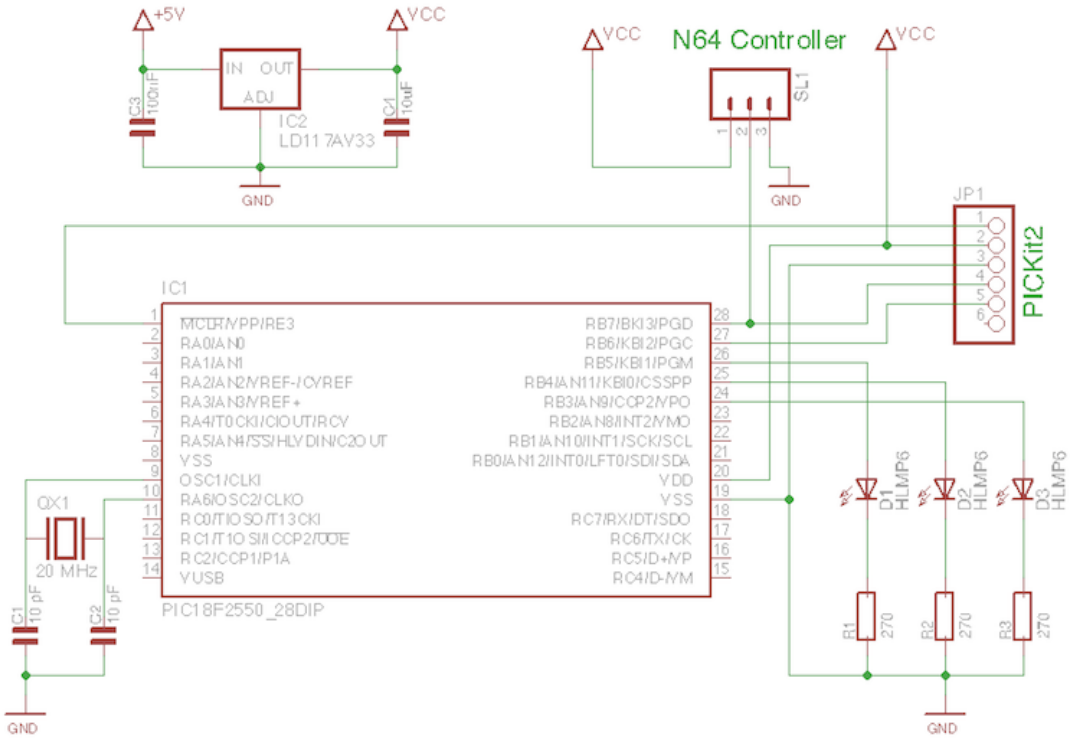
Connector

The N64 controller has three connections. From right to left with the round side up: Ground, Data and VCC (**3.3V**). Just plug some cables in the holes and you're ready to go.



Hardware

The circuit I created for reading out the controller is shown in the schematic below. It is based around a **PIC18F2550**. I chose a pic from the 18F range, because I am interested in the USB capabilities, as I might try to use the N64 controller with a computer some day (future article maybe?). I did however limit my oscillator frequency to 20 MHz, so that everything should work on a 16F pic as well if you don't need USB.

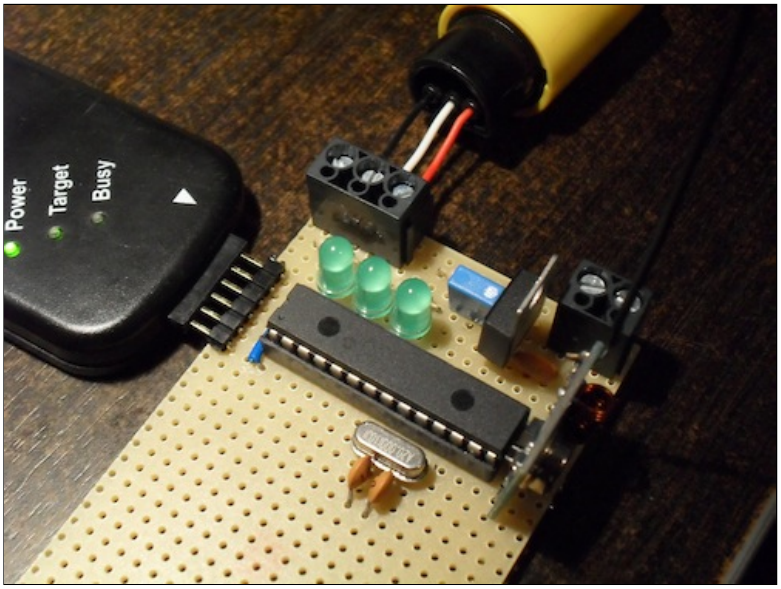


I've connected **three LED's** to RB5, RB4 and RB3. They will provide some visual feedback so we can see if everything works properly. A **3.3V voltage regulator** provides the voltage for the board. The maximum voltage the N64 controller can handle is 3.6V, so don't exceed that. If you use 5V you will risk frying your controller. The N64 controller is connected to the board with a simple screw connector. Here the supply voltage is given, and the data pin is connected to the RB7 pin of the PIC18F2550.

I've also connected a header (JP1) for the PICKit2. This allows me to program the PIC without having to take it out of the circuit (ICSP). At the moment it also provides the supply voltage to the board because I am to lazy to find a good power supply. Make sure you set the PICKit to 3.3V instead of the default 5V for the reason I mentioned earlier.

The image below shows the fully built circuit built on perfboard. The rectangular thing to the right of the microcontroller is a wireless RF

transmitter. I will talk about this in a next article, but it is of no importance to us now.

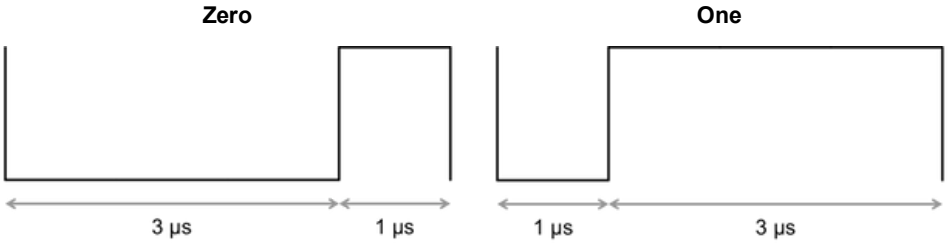


Interface

Now we're going to take a quick look at the communication with the Nintendo 64 controller. The interface looks a little bit like one wire, but is still different. It's a serial interface, which means that all the signals discussed here, will take place on one pin: the DATA (middle) pin of the connector (white wire in the pictures above).

Signal Shapes

The signal shapes used by the N64 controller for one and zero are shown below. As you can see these signals are **very fast** with a period of 4 uS and a resolution of 1 us! On a PIC microcontroller, one instruction cycle equals 4 oscillator cycles, which means that with a 20 MHz oscillator, **1 uS takes only 5 instruction cycles**! To generate these signals, we will need a low-level programming language like **assembly**, as C just won't cut it. I will give as much information as needed, so if this is your first experience with assembly, you will be able to follow easily. A good reference for the PIC instruction set can be found here: PIC Instructions.




```
movlw 0x00    5
movwf PORTB   1
nop           2
nop           3
nop           4
nop           5
nop           6
nop           7
nop           8
nop           9
nop           10
nop           11
nop           12
nop           13
nop           14
movlw 0x80    15
movwf PORTB   1
nop           2
nop           3
```

```
movlw 0x00    15
movwf PORTB   1
nop           2
nop           3
nop           4
movlw 0x80    5
movwf PORTB   1
nop           2
nop           3
nop           4
nop           5
nop           6
nop           7
nop           8
nop           9
nop           10
nop           11
nop           12
nop           13
```

```
nop          4                    nop          14
```

Below both signal shapes I have included a very simple program to generate the signal in asm (of course there are other possibilities e.g. with bsf and bcf). There are only three different instructions used:

- **movlw**: place the value that you indicate in the W register.
- **movwf**: place the value that is currently in the W register, in the register that you indicate.
- **nop**: do nothing.

It's easy to see that a combo of **movlw** with **movwf** can set a pin by modifying the PORTB register. As our N64 controller is connected to pin RB7, we set this pin high by sending the binary signal 0b10000000 to PORTB, which is 0x80 in hexadecimals. Ofcouse sending 0x00 makes the pin 0.

All three instructions we used take 1 instruction cycle (check the PIC 18F instruction set here). This means that it is very easy to count every step. As I said previously: 1 uS takes 5 instruction cycles. That means that, to generate a zero, we have to set pin RB7 low, wait for 15 instructions, and then set it high for 5 instructions. I have counted and indicated the instructions that the pin is high with a **bold** font, and the instructions that it is low with a gray font. You have to think about these code blocks as if a lot of them are cascaded after each other, so that the first instruction of every code block, is actually the last instruction of the previous block. This will make the timing exact.

Of course this code will only work with an oscillator frequency of 20 MHz on a PIC microcontroller. If you have another oscillator you can easily calculate how many instruction cycles you need for 1 uS though. If you are using another architecture (AVR, ARM, ...) you have to figure out how many oscillator cycles one instruction takes first (it is usually less than 4, which makes it easier to use C on those architectures).

Controller Data

The N64 controller data consists of a 32 bit (4 bytes), which gives you the status (on/off) of all buttons and the joystick position (2 bytes) and one stop bit. The array is built up like this:

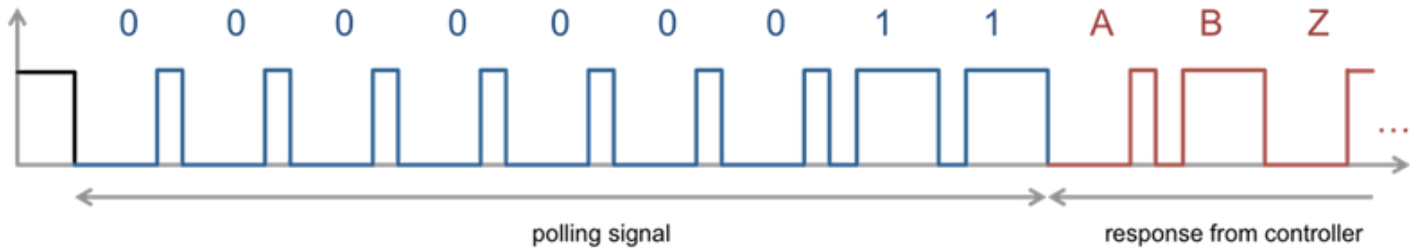
0	A
1	B
2	Z
3	Start
4	Up
5	Down
6	Left
7	Right
8	/
9	/
10	L
11	R
12	C-Up
13	C-Down
14	C-Left
15	C-Right
16-23	X-Axis
24-31	Y-Axis
32	Stop bit (1)

e.g. if the Z button is pressed, the 3th element of this array will read one etc.

The position of the joystick can be determined from the two bytes given by "X-Axis" and "Y-Axis".

Polling Signal

To receive the data array from the controller, a polling signal has to be sent first. This signal is: **ob000000011** (9 bits). After this signal is sent, the controller will respond with the 33 bit array (4 bytes + stop bit). This is shown in the picture below. The B-button is pushed so the second bit of the response is 1.



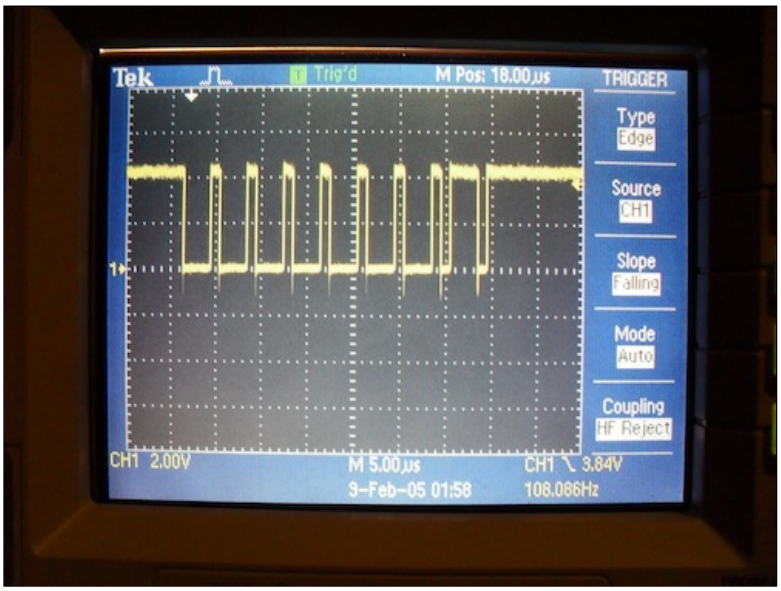
The assembly code used to generate the polling signal is shown below. Instead of copy pasting 7 zero blocks and 2 one blocks from above (which would work perfectly). I used the **decfsz** (decrement file, skip if zero) instruction to create loops. This instruction will decrement the register you indicate, and skip the next instruction if this register becomes zero. It takes two instruction cycles if it skips, else one! The registers d1, d2 and d3 can be defined in the beginning of the program using a cblock. After the decfsz instruction there is a **goto** instruction, which will move the program counter to the instruction after the word that you have indicated. It should be clear to see how a combo of these two instructions create a loop.

All instructions use one instruction cycle (4 oscillator cycles), except for: **goto**, which uses 2 instruction cycles. Of course **decfsz** also takes two instruction cycles when it skips, as mentioned earlier. Its not so hard to distinguish the two code blocks we saw earlier, only this time I tried to make them a little shorter by replacing some of the nop's with loops (zeroloop, oneloop). If you take the time to count the instruction cycles carefully for each loop (which I strongly recommend), you will see that there would be no difference if we would copy paste 7 zero's and 2 one's from the previous code.

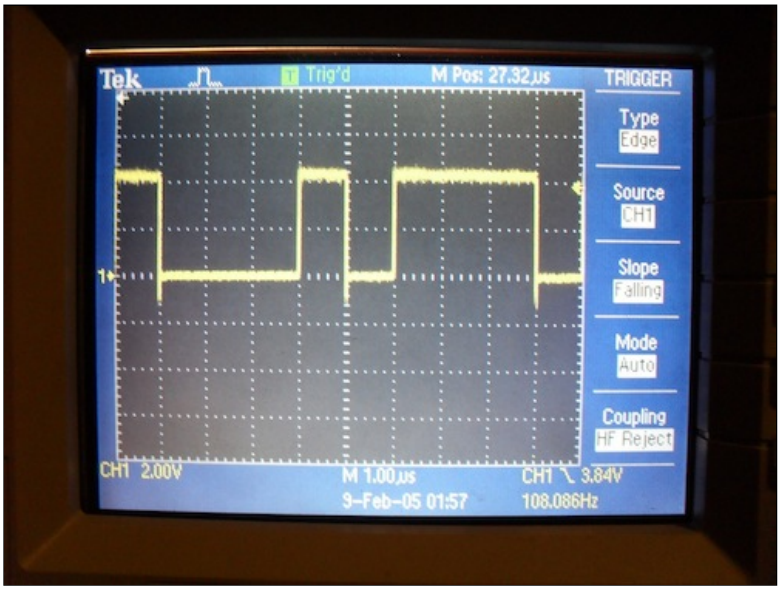
```
bcf TRISB, 7            ; make RB7 output

; 0b000000011
movlw 0x07              ; number of zeros that need to be sent
movwf d2

movlw 0x02              ; number of ones that need to be sent
movwf d3

zero
    movlw 0x00
    movwf PORTB
    movlw 0x03
    movwf d1
    zeroloop            ; 3 instruction cycles * 3 = 9
        decfsz d1, f
        goto zeroloop
    nop
    nop
    movlw 0x80
    movwf PORTB
        decfsz d2
    goto zero

one
    movlw 0x00
    movwf PORTB
    nop
    nop
        nop
    movlw 0x80
    movwf PORTB
    movlw 0x02
    movwf d1
    oneloop             ; 4 instruction cycles * 2 = 8
        nop
        decfsz d1, f
        goto oneloop
    decfsz d3
goto one
```

note: the pin RB7 is set to function as an output pin by setting the TRISB register (duh!).

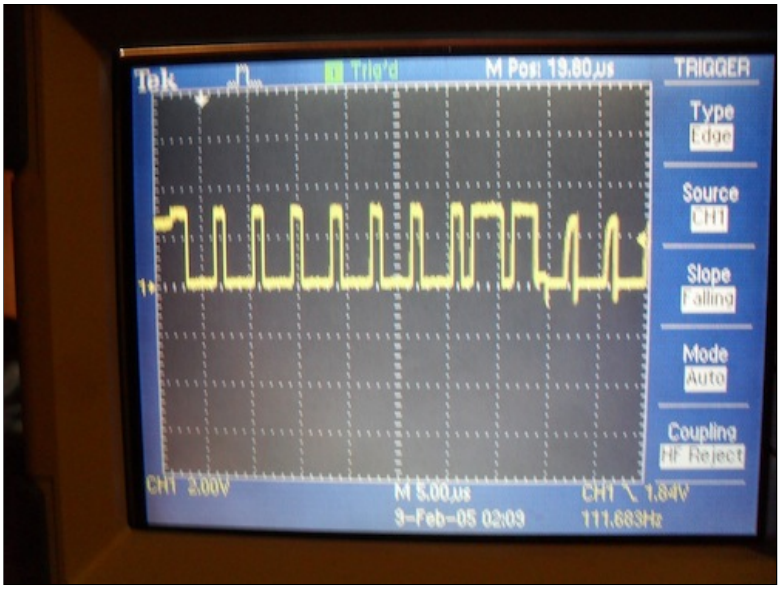
In the image below you can see a picture of an oscilloscope measuring the polling signal generated with the code above. You can clearly see the signal **ob00000011**.

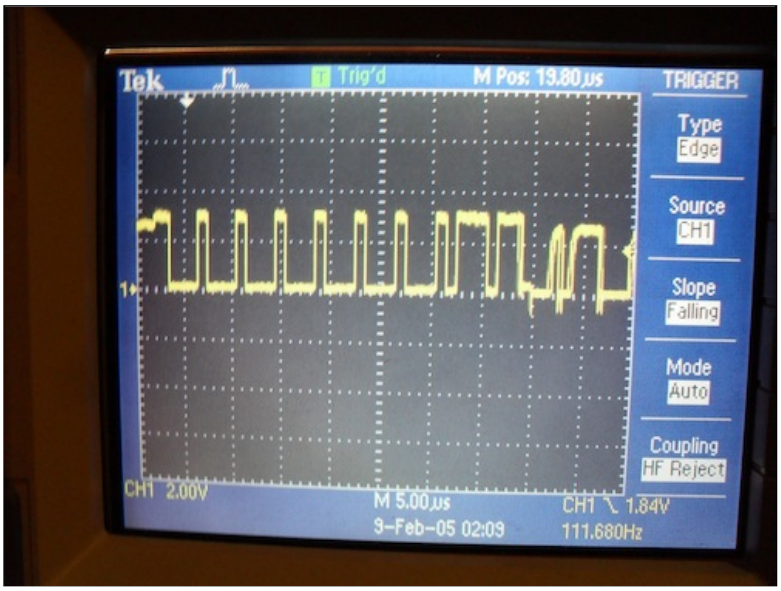


This is a close up of the 7th (0) and 8th (1) bit of the polling signal. The timebase is set to 1 uS. As you can see, the signal is extremely correct.



If you now connect the N64 controller, you should see that it responds immediately after the polling signal was sent. I have shown this in the pictures below. In the second picture the B button is pressed, so you can see that the second bit of the response is a 1. Once you attach the controller, the signal wil deform because of its impedance. This is normal and not really a problem for us.





Reading The Controller Data

Now we succeeded in triggering the Nintendo 64 controller so that it sends its data, we have to find a way to read & store this data properly. The easiest way to do this, is to wait until the middle of the signal (approx 2 uS) and then see if it is one or zero with the **btfss** (bit file, skip if set) instruction. This instruction will check the bit you indicate (e.g. "PORTB, 7" -> 7th bit of the PORTB register) and then skip the next instruction if this bit is one. It will take one instruction cycle, except if it skips when it will take 2 instruction cycles (it compensates automatically for the skipped instruction - nice!). Of course, we have to **switch the RB7 pin to an intput now** by setting it to 1 in the TRISB register.

Once we know if the bit was a 0 or a 1, we can store this somewhere in the RAM by using indirect addressing. The INDF0 register contains the value of the register pointed to by FSR0L, so if we increment FSR0L, we go to the next adress etc.

We can now read and store all values, so we will do this 33 times as there are 33 bits in the N64 array (see higher).

```
movlw 0x22              ; 0x21 = 33 bits
movwf d4                ; contains the number of buttons

movlw 0x20              ; set up the array adress for indirect adressing
movwf FSR0L

bsf TRISB, 7            ; RB7 as input
```

```
readLoop                ; read all 33 bits and store them on locations 0x20 etc.
    nop                 ; Wait
    nop
    nop
    nop
    nop
    nop
    nop
    movlw 0xFF          ; if bit is one
    btfss PORTB, 7      ; read the bit (9th instruction)
    movlw 0x00          ; if bit is zero
    movwf INDF0         ; Wait & store the value of the bit in the array
    incf FSR0L          ; Go to next adres in the array
    nop
    nop
    nop
    nop
    nop
    decfsz d4           ; end loop if all 33 bits are read
goto readLoop
```

note 1: the syntax for indirect addressing is a little bit different on 16F architectures. They use FSR and INDF.

note 2: this is definitely not the best way to read out a serial signal. A better way would be to re-sync the signal after every bit, by searching for a rising edge and then falling edge. This way you know for sure that you are at the beginning of the pulse form. I have tried to do this, but it seemed impossible with the slow pic (4 oscillator cycles / instruction cycle). You might succeed with this approach on an AVR though (1 oscillator cycle / instruction). Please let me know if you find a cleaner way! (This works perfectly though ...)

The Full Assembly (ASM) program

The full assembly program is given below. It blinks the leds RB3, RB4 and RB5 respectively when the buttons A, B & Z are pressed. I made use of the **mpasm (v5.46) compiler** which comes with the MPLAB X IDE. It's probably pretty easy to figure out what everything does if you look up the instructions on the webpage I referred to earlier, so I won't go into detail here.

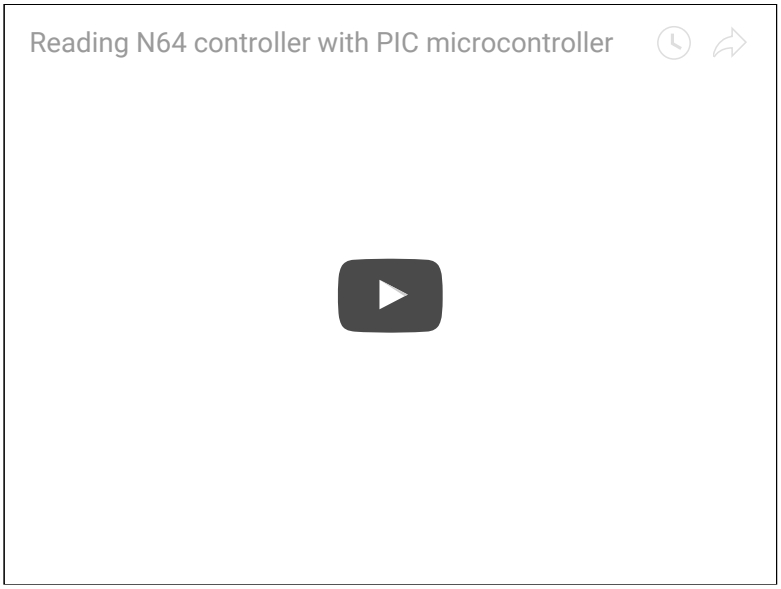
```
#include <P18F2550.INC>

CONFIG WDT = OFF        ; disable watchdog timer
CONFIG FOSC = HS        ; Oscillator crystal of 20MHz!
CONFIG LVP = OFF        ; Low-Voltage programming disabled
CONFIG DEBUG = OFF
CONFIG CP0 = OFF
CONFIG CP1 = OFF
CONFIG MCLRE = OFF

org 0

init
    cblock
        d1
        d2
        d3
        d4
    endc

    movlw 0x00
    movwf LATB
    movwf TRISB
    movwf PORTB

    movlw 0x80              ; initalise the line = HIGH
    movwf PORTB
```

```
Main
    ; Part 1 - Read the N64 controller and store the status values

    movlw 0x22              ; 0x21 = 33 bits
    movwf d4                ; contains the number of buttons

    movlw 0x20              ; set up the array adress
    movwf FSR0L

    bcf TRISB, 7            ; make RB7 output
    call poll               ; Send the polling signal (0b000000011)

    bsf TRISB, 7            ; RB7 as input  (is also last instruction of poll)

    readLoop                ; read all 33 bits and store them on locations 0x20 etc.

        nop                 ; Wait
        nop
        nop
        nop
        nop
        nop
        nop

        movlw 0xFF          ; if bit is one
        btfss PORTB, 7      ; read the bit (9th instruction)
        movlw 0x00          ; if bit is zero

        movwf INDF0         ; Wait & store the value of the bit in the array
        incf FSR0L          ; Go to next adres in the array

        nop
        nop
        nop
        nop
        nop

        decfsz d4           ; end loop if all 33 bits are read
    goto readLoop

    ; Blink some leds

    movlw 0x20              ; read A button (0x20) from array
    movwf FSR0L
    movlw 0x20              ; led RB5
    btfsc INDF0, 0
    iorwf LATB, PORTB

    incf FSR0L              ; read B button (0x21) from array
    movlw 0x10              ; led RB4
    btfsc INDF0, 0
    iorwf LATB, PORTB

    incf FSR0L              ; read Z button (0x22) from array
    movlw 0x08              ; led RB3
    btfsc INDF0, 0
    iorwf LATB, PORTB

    call delay
goto Main

poll                        ; 0b000000011
    movlw 0x07              ; number of zeros that need to be sent
    movwf d2
```

```
    movlw 0x02              ; number of ones that need to be sent
    movwf d3

    zero
        movlw 0x00
        movwf PORTB
        movlw 0x03
        movwf d1
        zeroloop            ; 9 instruction cycles
            decfsz d1
            goto zeroloop
        nop
        nop
        movlw 0x80
        movwf PORTB
        decfsz d2
        goto zero

    one
        movlw 0x00
        movwf PORTB
        nop
        nop
        nop
        movlw 0x80
        movwf PORTB
        movlw 0x02
        movwf d1
        oneloop             ; 8 instruction cycles
            nop
            decfsz d1
            goto oneloop
        decfsz d3
        goto one
return


delay   movlw   0x5f        ; +- 1500 uS = 1.5 ms
        movwf   d1
outer   movlw   0x60
        movwf   d2
inner   nop
        nop
        decfsz  d2
        goto    inner       ; Inner loop = 5 cycles = 1uS
        decfsz  d1
        goto    outer       ; outer loop = inner + 4 cycles
return

end
```

Video

Here is a small video of the result. I'm sorry that I didn't do anything cooler with it than blinking some leds, but that's for when I have some more time!



Tags:
Microcontrollers