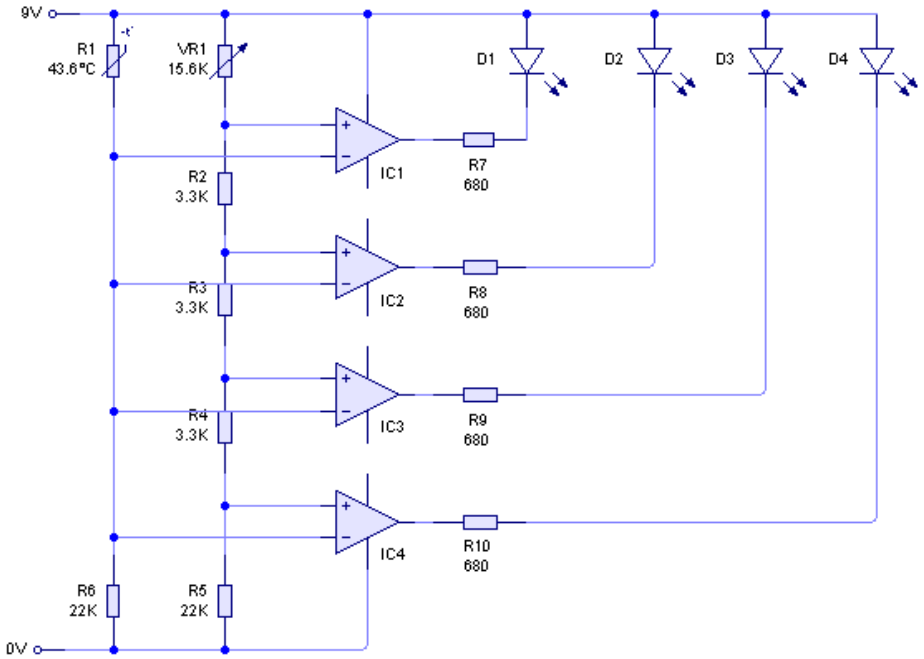
Tom Ladyman

Sutton Grammar

AS Systems and Control

PIC Project



This circuit is a very simple thermometer. As the thermistor (R1) heats up, it triggers different cascading op amps. This lights a series of different LEDs to show the temperature. For clarity of the temperature, I colour coded them on the breadboard.

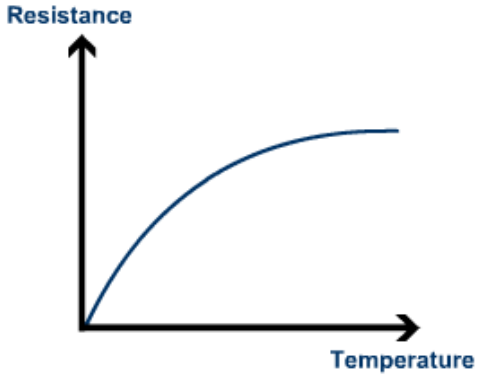
Although simple, the circuit's sensitivity can be adjusted by changing the variable resistor VR1. In the photo, I use a potentiometer wired as a variable resistor.

After trying to calibrate it to output any useful information, I quickly found the major flaws in the circuit. Firstly, the calibration is only guessed, as it is very difficult to produce an accurate and precise resistance with a variable resistor. Comparing the data to a precise stored value would be a much better solution.

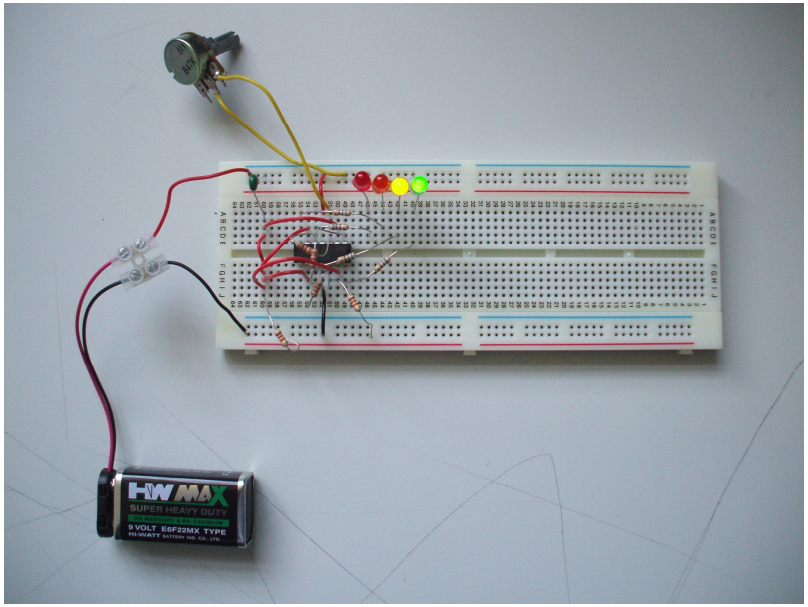
Also, the fixed resistors are fixed, they cannot be changed, and all of the cascading logic gates are calibrated together. It would therefore be impossible to light an LED at uneven intervals.

Again, because the circuit is mainly hardwired, it would be very difficult to change the order the LEDs come on in, or to add extra modules or functionality to the circuit.

My final criticism of the circuit is that the thermistor merely changes its resistance with a change in temperature, it is highly inaccurate. Additionally, thermistors have a non linear current-voltage trend. This means that it is not an Ohmic conductor and is even more inaccurate than first thought, as the changes in resistance are not linear and increases at a decreasing rate. This almost invalidates the readings at either end of the scale.

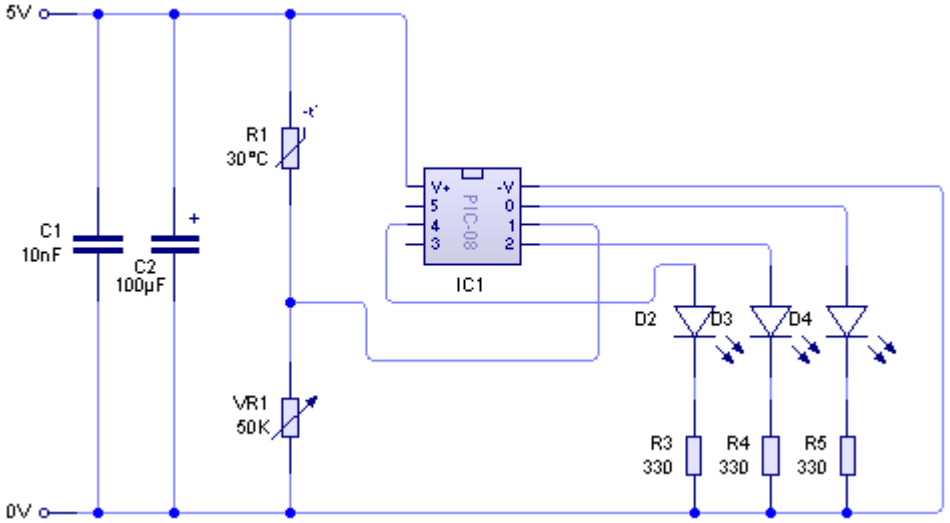


I decided to improve the circuit by making use of a microcontroller. By changing the program, I will be able to modify everything about the way it functions, build onto it and use more accurate sensors, if desired. It could also mean a huge increase in the accuracy of the circuit because the circuit can be much more easily calibrated or the extreme value filtered out or modified.



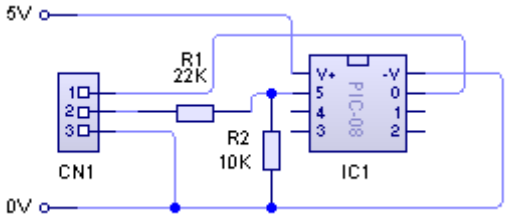
PICAxe PICs

Unlike op-amps, which behave in a set way, whatever, Peripheral Interface Controllers are programmable chips which can process information and make decisions. PICAxe chips are generally only used in education and rarely in industry. This is because the code is highly simplified, making the programs sluggish and often far larger than they need to be. The chips also need PICAxe tools to program with, which can be costly. Due to the simplification of the code, both efficiency and practically can be lost, as more complex tasks are sometimes harder to achieve. The circuit on the previous page acted as a sort of thermometer, the circuit on the right does the same, along with the program below.



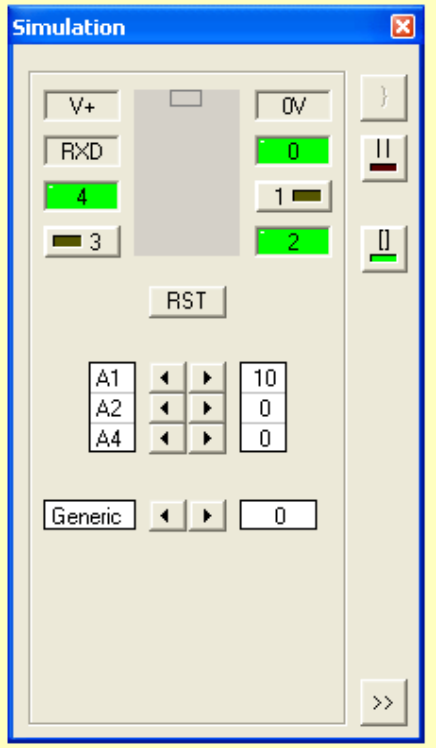

```
1       main:
2       let pins = 0
3       readadc 1, b1
4       if b1 < 5 then goto five
5       if b1 < 10 then goto ten
6       if b1 < 15 then goto fifteen
7       goto main
8       five:
9       high 4
10      goto main
11      ten:
12      high 4
13      high 2
14      goto main
15      fifteen:
16      high 4
17      high 2
18      high 0
19      goto main
```

I used the values of 5, 10 and 15 as arbitrary units to show the concept. On a real circuit, depending on the value of the variable, the values would be different. The program works by utilising the inbuilt analogue to digital converter to change the analogue value of the thermistor and variable resistor combination into a number. This number can then be processed and compared to turn the LEDs on in sequence. In actual fact, the LEDs strobe on and off very quickly, faster than the eye can see. This is to make the code shorter as, without it, there would need to be an extra set of routines to switch LEDs off when the temperature drops.



The program can be downloaded by using either a download board or a download socket (above).

Programming editor has a built in simulation program (right). Although this does not guarantee that the circuit will work in the real world, it helps eliminate many errors. The window shows the input on adc 1 as 10 which turns all three of the outputs on.



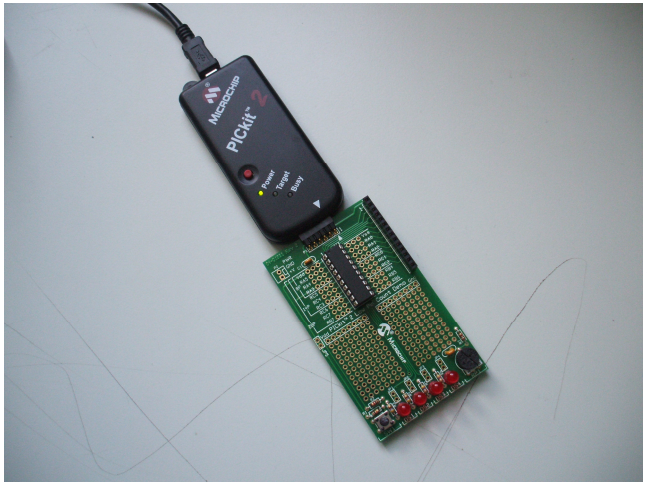
Microchip PICs

Microchip are one of the most popular manufactures of PICs, announcing the shipment of their 6th billionth PIC processor in February 2008. Every chip Microchip makes has a model specific code. This not only helps identify the chip, but also communicates important information about the chip. Every code starts with “PIC” and is followed by one of these numbers: (if 8 bit:) 10, 12, 14, 16, 17, 18, (if 16 bit:) 24, 30, 33, (if 32 bit:) 32. Although there are different families of PICs with different bits, they are all, in a way, 8 bit chips. This is because all of the data in their memory is based around an 8 bit byte, it is just the way they address these blocks of data which varies across the families. Following the family of chip, there comes a letter, either F, C or H. This shows the kind of memory they use, and, ultimately, how they can be erased. With a couple of exceptions, a C shows that the chip is not electronically erasable and must be exposed to UV light to remove the data, whereas an F shows that the chip is flash memory and can be electronically erased. The code for the individual variant of chip within the family comes next. An example of a full Microchip code is PIC16F84. This PIC is 8 bit and uses flash memory.

PICs are designed on the “Harvard Architecture” which separates the data and processes within the chip. This was originally used with very early computers, where the program was read from a tape, or other external data storage device, and the computer only processed the information. There are both advantages and disadvantages to this. It means that the data will never exceed a certain amount, causing the processing to slow down, and gaining continuity across different chips, but it also limits the amount of space available for the storage of the program. Furthermore, the processor and latency of the memory need to be equal, or else the speed of the processor is restricted by the speed it gains program data from the memory.

There are many different ways in which a PIC can be programmed; different languages as well as methods. Each one of them requires code to be compiled, and then ‘blown’ onto the PIC. Compilation on the simplest level, involves converting humanly readable data (such as C or Assembly code) into something which a computer or Microprocessor can understand. This results in a .hex file, which is a string of numbers. Each string refers to a specific instruction on the PIC. Although it is possible to write very efficient code by writing this ‘machine code’, it is very difficult to understand and troubleshoot, and, due to the power of modern compilers, is generally unnecessary. Many developers use simulation before blowing a PIC, so that they can test the code works without the constraint of hardware issues. Reprogrammable chips also have a limit before the capacitors inside them wear out, and some chips are not reprogrammable at all. The number of inbuilt instructions on a PIC varies depending on the quality of the chip. Cheaper ones can have 35 instructions, while better range ones can have around 80.

Whereas manufacturers such as PICAxe hide important information about how the chip works to simply develop code for it, Microchip PICs are just raw PICs. This means that they can be programmed in any language or any combination of languages. Inside the chip are file registers. This are directly related to the circuitry within the chip and are used to control inputs, outputs and processes within the chip. There are 16 useable registers (numbered in hex), 8 of them are predefined, while the remaining 8 can be used for any kind of data. From 00 to 07, they are: Indirect Address, TMR0, STATUS, FSR, PORTA, PORTB, PORTC. For simple use, the only three registers which matter are PORTA, PORTB and PORTC. These registers hold the data for the pins. For example, the pins of a PIC are labelled RA0-4 and then RB0-7 ending with RC0-7, depending on how many pins. The second letter in the pin name corresponds to the file register which holds the data on them. The data is simple a bit which shows whether the pin is high or low. Any pin can be used as either an input or an output by using the assembly command “TRIS” followed by A, B or C depending on which file register it belongs to. The data within the register is read in the same way for both inputs and outputs. For example, say PORTB contains 4 inputs and 4 outputs. Although the pins are handled differently, PORTB is still a byte, say: 10101001. The output pins, however, can be manipulated, while the inputs pins cannot, they can only be read. Many chips are too small to include PORTC, in which case, register 07 can be used for any purpose. File registers are an important element of PIC programming, in any language, as they directly interface with the input and output pins.

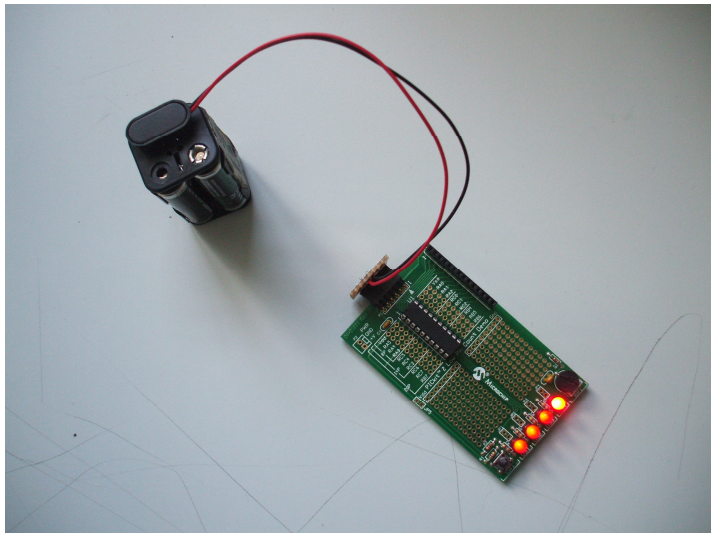


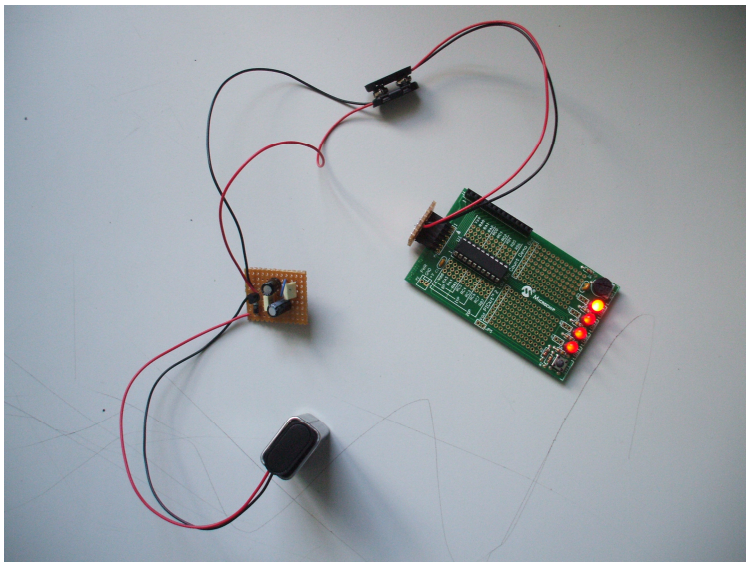
I started with the PICKit 2 Programmer, which came with a demo board and a PIC already assembled. The programmer was USB and interfaced with the MPLAB software made by Microchip without any configuration. Also, the programmer was self powered through the 5V output of a USB port. After writing a program, I compiled it and programmed it to the PIC. The program was supposed to make the LED's come

multimeter to measure the voltage and current across the circuit. The voltage was fine, but there wasn't enough current being drawn through the USB to make the LEDs turn on, so I soldered a header onto a piece of Veroboard and ran it from a 6V battery, and it worked as I had expected it to.

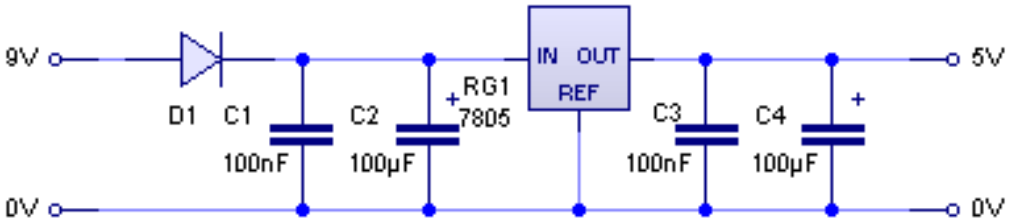
```
#include <htc.h>

void main (void) {
    TRISA = 1b111111;
    TRISB = 1b111111;
    TRISC = 0b000000;
    RC0 = 1;
    RC1 = 1;
    RC2 = 1;
    RC3 = 1;
}
```





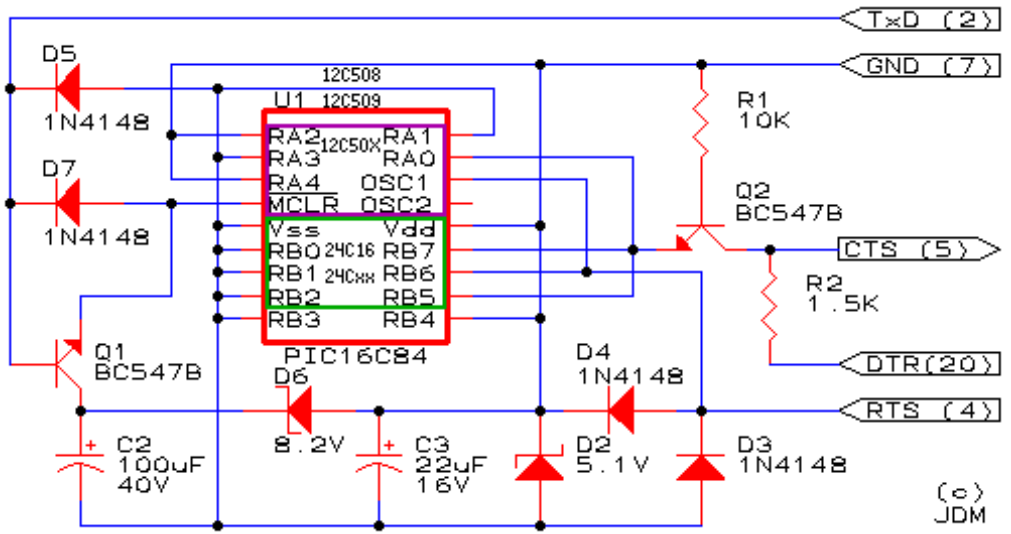
I made a voltage regulator on Veroboard so that I could run the circuit from a 9 Volt battery. The circuit includes a feedback diode as well as smoothing and decoupling capacitors on each side of the regulator. I soldered an extra battery clip to the output with the wires reversed, so that I can use the circuit like a normal battery.



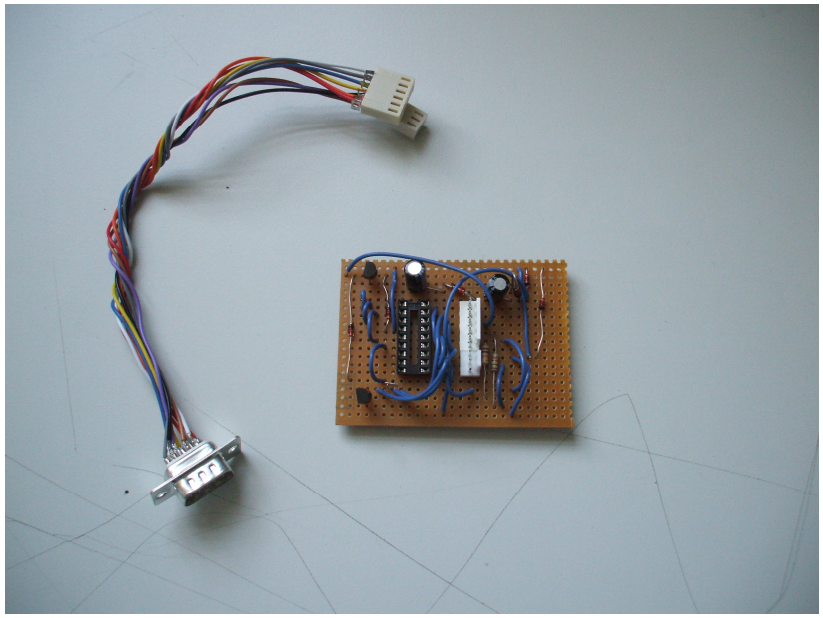
The JDM Programmer

The JDM programmer was designed by Jens Dyekjær Madsen and is one of the most popular programmers. It was originally specified to use with a 25 pin serial port, but I converted the pins to use with a 9 pin. This means that the connectors on the right hand side of the circuit diagram are different: TxD becomes pin 3, GND, pin 5, CTS, 8, DTR, 4 and RTS, 7. The circuit has no power supply as it is powered from the serial port, however, the RS-232 standard defines no dedicated pins for power supply and can output around 12 volts. Generally, newer computers are not compliant to this standard, and USB to serial port adaptors never are. When the voltage outputted from the serial port is too low, such as on many modern computers, the circuit will not function.

On the creator's website, I found an explanation for how the circuit functions, which I have elaborated upon and added information about how the serial port functions.



The logic for the serial port data transfer is reversed so that a 1 corresponds to a negative voltage and a 0 becomes a positive one. This means that the chip is driven on a negative voltage and the ground pin is not actually 0V, but a negative value. An external power supply can not be used, therefore, because it will short out the computer. There are current limiters built into the RS-232 standard which means that current limiting resistors are not needed in the circuit. For each mA of current which is drawn through the port, the voltage drops between 1 and 2 volts. Power for the circuit is provided through the RTS port, and the voltage to the clock is limited by D3 and D4. These are necessary because PICs typically run on 4.5-5.5 volts, and a 12v output from the serial port will easy blow the PIC, making it useless. Q2 acts as an amplifier for the output of the chip which is read by the computer. Because the voltage has been dropped for the chip, it must be amplified to comply with RS-232. R2 is a pull up resistor, but isn't completely necessary. When DTR is high, Q2 also limits the voltage to Vdd to about 0.7V. When DTR is low, Q2 inverts and creates a resistance with R1. In this case, the amplification is around 5, giving an effective resistance of 10K/5 = 2K. This reduces the chips current draw and keeps the voltage from the serial port high, because of the large voltage drop per mA. As DTR turns from low to high, the voltage spikes. This is only important for eeprom, as it insures the chip isn't in test mode. Q1 controls the voltage at MCLR, this resets the chip. C2 is charged by TxD through Q1 and the voltage across that is limited by the zeners, D6 and D2. This means that the voltage across the capacitor is approximately 13.3V. When TxD is high, the voltage on MCLR will not exceed the voltage across C2. This capacitor also decouples Vss and Vdd through D6, but only when the voltage across is around 13V. This is one reason it is important to use a high voltage serial port. If the voltage across C2 is 8V, then turning RTS and DTR will spike the voltage and control the power. Holding TxD, DTR and RTS high for half a second or so will cause C2 to drop to 8V. D5 is used to limit the voltage on TxD and also to power eeprom when DTR and RTS are both high. Also, when TxD is low, it keeps MCLR higher than –0.2V and the diode D7 pulls the voltage at MCLR low. Both of the signals going to Vss and Vdd need to be negative to power the chip at the highest possible current. This is possible because the potential difference between the two voltages is what drives the circuit. It explains why the ground is actually a negative voltage, rather than a real ground. If the signals are positive, C2 powers the chip. When the data is transferred to program the chip, RTS and DTR switch between positive and negative output. Because the data is reversed, when a 0 is transferred, the voltage is high. This means that there is a higher voltage supplied to the chip from DTR and RTS must be made low to compensate.



This was my first attempt at building the JDM programmer.
It has a lot of errors because I built it straight from the circuit diagram without a stripboard diagram. It highlights the danger of badly built programmers, because it will permanently break any PIC inserted into it.

Programming

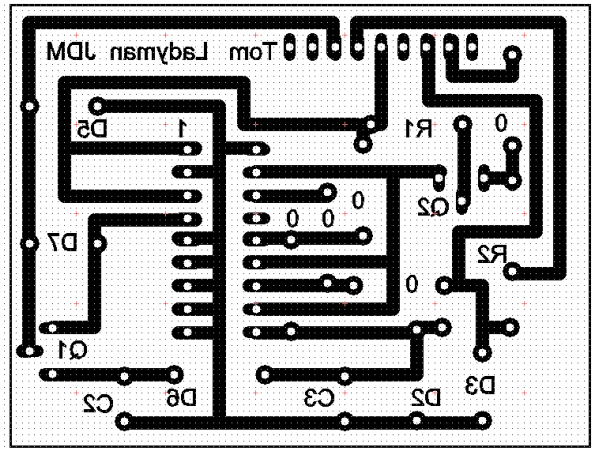
This is a very basic program which turns on an LED and leaves three unlit. Then, when a switch is pressed, the originally lit LED turns off, and the other two light up.

The program is written in C, which must then be compiled into hex for the chip to understand it.

The opening statement is an include statement which includes the “htc.h” header file. This is a compiler specific file containing information about the chip I have specified to use. The next line is the main function. This is a loop in which all of the main program resides. I then set the inputs and outputs on PORTA and PORTB. RA* is the identifier for PORTA pins, and RB* is the same for PORTB. Next I turn two LEDs off and one on and check whether input 3 on PORTB is high or low. If it is low, the program reloops and if it is high, the LEDs lit change.

```
#include <htc.h>

void main (void) {
TRISA = 0b000000;
TRISB = 0b011000;
RA0 = 0;
RA1 = 1;
RB7 = 0;
if(RB3 == 0){
RA0 = 1;
RA1 = 0;
RB7 = 1;
}
```

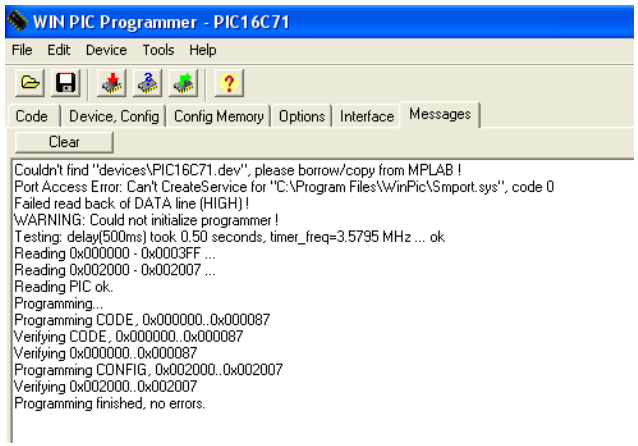


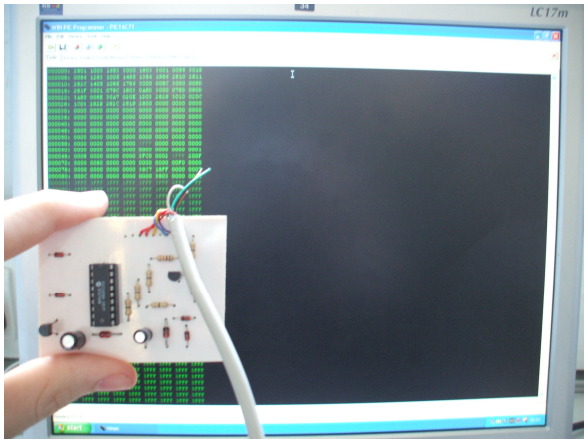
I used a piece of software called “WinPic” to program the PIC through the JDM programmer. This program works with .hex files, which are compiled versions of C code. Hex is a base 16 number system (see table), and the numbers correspond to an instruction built into the chip.
The PCB layout on the left is of the programmer. I tried to keep the layout as close to the circuit diagram as possible, which meant I had to use many link wires (0 Ohm resistors).
An example of a .hex file is on the middle left, in bold text. It is just about visible in the photograph of WinPic running (green text).

Base 10	Binary	Hex
0	00000000	00
1	00000001	01
2	00000010	02
3	00000011	03
4	00000100	04
5	00000101	05
6	00000110	06
7	00000111	07
8	00001000	08
9	00001001	09
10	00001010	0A
11	00001011	0B
12	00001100	0C
13	00001101	0D
14	00001110	0E
15	00001111	0F

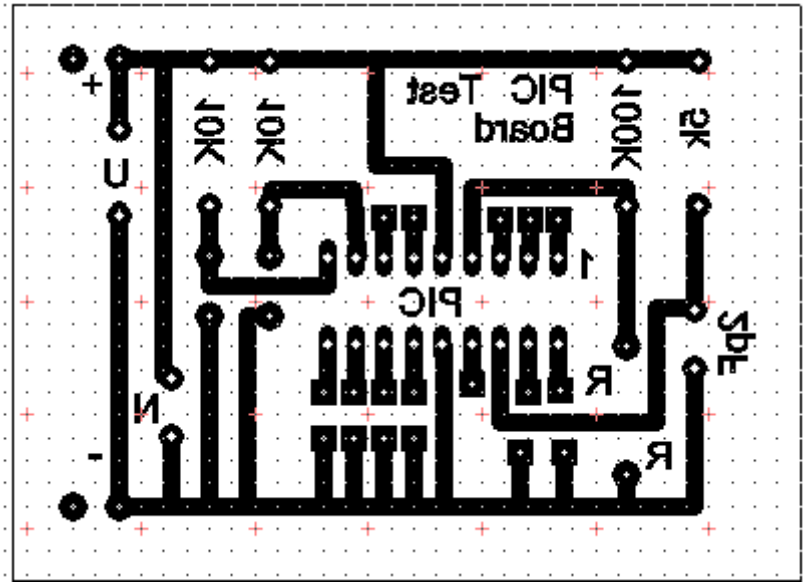
:10000000012803108316003003180130850018300D2
:1000100086008312051085148613861910281128 6E
:100020002C28051485108617003080000308D00B 8
:100030001F2801308C0703188D0A00308D070D08 2A
:10004000803A8E00A7300E02031D282810300C02 C3
:0A005000031C2B282C281928002877
:00000001FF

WinPic has options for many of the popular programmers, including JDM. It also includes specific modes for certain PICs by default and, with a datasheet, it is possible to add support for extra chips. It also supports eeprom programming.
The software can read chips and it has a programmer tester built in, although I found, it is not very accurate. It can blank chips as well (only flash chips) and can control the voltage out of the serial port (up to 13 volts).
The default setting for the PIC16F84 and JDM on COM port work perfectly, however.

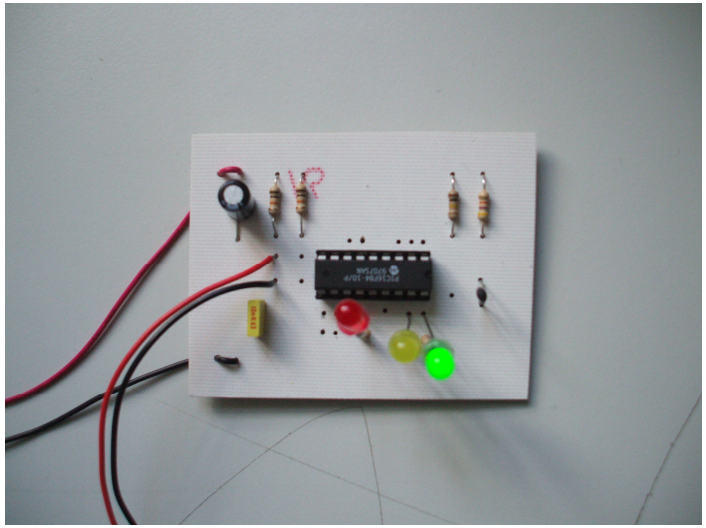




The errors generated in the Message window (bottom left) are before I plugged the circuit into the serial port: “WARNING: Could not initialize programmer !”. After this, it shows the reading of the chip, then the programming. After it has done this, the software verifies that the program has been blown to the chip correctly, and outputs “Programming finished, no errors”.



This is a PCB I made as a test board for PICs. I can add extra components to the inputs and outputs when I need to. The chip doesn't have an in built clock so I needed to use an external one. As this was only a testing board, a resistor-capacitor timing system was accurate enough. I used a 5K resistor and 2pF capacitor in series. Each instruction takes 4 cycles of the clock to execute. This means that when using a 4Mhz crystal or resonator, the chip processes one instruction every microsecond.
On the PCB design, I connected the positive pin of the chip the negative rail and the negative to positive, so I broke the tracks by cutting them with a knife, and soldered two link wires across the bottom. The picture of the working PCB shows a program where, one LED is one and when the switch is pressed (pulling the input low, not high), it turns off, and the other two come on. It is a very simple concept but it shows the concept of programming in C.



The clock speed of a chip is very important because, unlike PICAxe, C doesn't have any built in functions for timing. This means that to delay an event, the chip needs to process a certain amount of useless code. For example, on a chip with a 4Mhz crystal, it would be necessary to add in 1 million lines of code to produce a delay of one second. This is easily done by creating a delay function, for example:

```
void delay (int x) {
int delay = 0; //variable to hold the amount of microseconds
delay = x*1000000; //convert the amount of seconds to delay to microseconds
while(delay > 0 ){ //while there are microseconds to count...
delay--;  //take one microsecond off the total
}
}
```

Although this function does create a time delay, it does have limitations. It means that the chip won't do anything else while the delay is happening, as the delay requires the PICs processing power. Also, it is not completely accurate, as the start of the function requires processing before the loop begins. For each delay, there will be 2 or 3 added microseconds, although this could easily be compensated for by manipulating the variable "delay" before the loop begins.