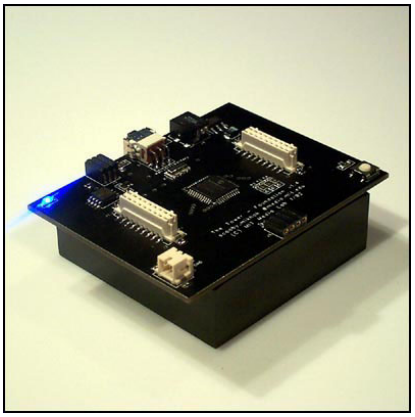
PIC Foundation

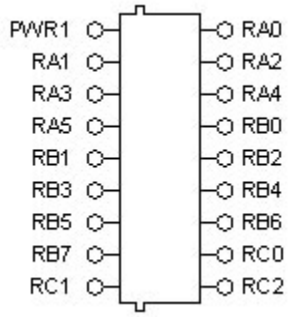


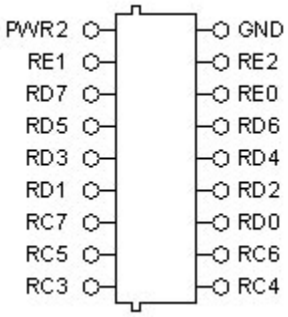
Description

The PIC Foundation is built around a Microchip PIC16F877 processor running at 8 MHz. The Foundation contains a serial programming header, as well as all of the power and support circuitry needed for basic operation. Every I/O pin from the processor is broken out to the main Tower connectors, and passed up and down the entire stack.

Hardware Detail

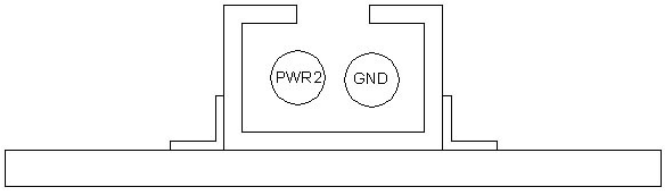
One of the key features of the Tower system, is that it can communicate with layers via the I2C protocol, yet at the same time any layer can have access to all of the I/O pins from the Foundation. To make this possible, all 33 I/O pins of the PIC16F877 are passed to every layer through the main Tower connectors. The connectors are 18-pin surface mount Hirose connectors. Each board has female connectors on the top, and male connectors on the bottom, so stacking layers is a simple process. In addition to the I/O pins, two power busses and a ground line are also available to every layer. The pin configuration for the connectors is detailed below:





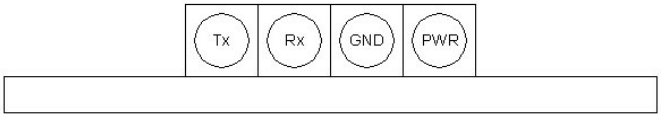
The Foundation itself as well as many of the layers, obtain their power only off of the primary power bus. The primary bus can be powered by either four AA batteries, or through a 1mm wall jack also located on the foundation. The PIC Foundation can accept either Alkaline or rechargeable batteries. As an alternative to batteries, the wall jack can be used. It must have a 1mm negative-tip connector, providing at least 6 volts, since it gets regulated down to a tight 5 volts, and needs a bit of overhead to do so.

Some layers that consume more power have the option of using either the primary of secondary bus. To put power on the secondary bus, a cable is needed that plugs into the 2-pin power connector on the Foundation. This connector is a standard Hirose 2-pin male surface mount header, with the pinout shown below:



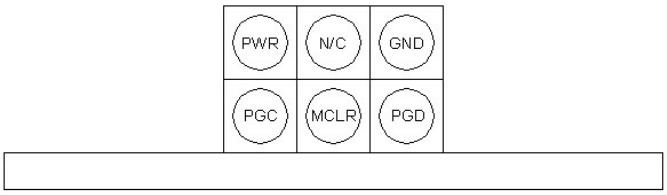
Power can be supplied by a standard bench power supply, or the Power module can be used to provide either more battery power, or another regulated 5-volt line if powered through a similar wall adapter. Both power busses are switched on and off using the same power switch, located at the top edge of the Foundation. The switch is three-way, with the middle position being “off”, and the two sides selecting between wall or battery power for the primary bus. Regardless of what source the primary bus is using, the secondary bus will always use whatever is provided to it through the connector, but can still be turned off by ensuring that the switch is in the middle position.

The Foundation itself can be programmed through the on-board serial-programming header when connected to a serial cable with an RS-232 module on the end. The pin configuration of the header on the board is shown below:



In addition to downloading Logo code, it is also possible to download Assembly code. To download assembly, the board must be powered on while the Button is depressed. At that point, Assembly code can be downloaded through the Tower Development Environment. After downloading new assembly code, it is necessary to power-cycle the layer before the new code will run.

In rare cases, it may be necessary to program the PIC on the Foundation with an actual hardware PIC programmer. If the serial-boot-loader program that handles assembly downloads becomes damaged, or if an assembly program needs to have precise control over the exact memory location of each instruction, it is necessary to program the chip directly through the programming header. The pin configuration of the 2mm programming header is shown here:



A special cable is needed to connect this header to the hardware PIC programmer. Not that since the MCLR pin will be pulled to a high voltage during programming, it has a no-connect pin directly across from it on the connector, to ensure that the high voltage will not damage the chip if the cable is accidentally connected upside down.

Foundation Code

For information on all of the primitives available in the virtual machine, please refer to the PIC Logo Language Reference document. There are however, three include files that pertain directly to functions on the foundation, which will be discussed here.

The **standard** include file contains the definitions for commonly used constants and global variables, as well as functions for driving output pins, performing A/D conversions, enabling Pulse-Width-Modulation, and looking up items in table entries.

The basic constant and global declarations are useful for standard arithmetic operations, as well as talking directly to frequently accessed registers such as the I/O ports. The declarations themselves are shown here:

```
constants
  [[porta 5][porta-ddr $85]
   [portb 6][portb-ddr $86]
   [portc 7][portc-ddr $87]
   [portd 8][portd-ddr $88]
   [porte 9][porte-ddr $89]
   [adcon0 $1f][adcon1 $9f]
      [adon 0][adgo 2]
   [adresh $1e][adresl $9e]
   [t2con $12][ccp1con $17][ccpr1l $15][pr2 $92]]

globals [nn i2c-byte n]
```

For controlling output signals on PIC pins, three functions have been written to simplify the process. While at its basic level, controlling outputs is as simple as just setting and clearing bits, an important step that users often forget, is that they need to first ensure that the pins are configured as outputs. All three procedures first set the pin to be an output, before talking to the pin itself. The **set**, **clear**, and **toggle** functions can be used to respectively turn-on, turn-off, and switch the state of output pins.

```
to set :chan :port
    clear-bit :chan (:port + $80)
    set-bit :chan :port
end

to clear :chan :port
    clear-bit :chan (:port + $80)
    clear-bit :chan :port
end

to toggle :chan :port
    clear-bit :chan (:port + $80)
    flip-bit :chan :port
end
```

The **ad** function is used to perform a 10-bit A/D conversion using one of the eight built in A/D pins on the PIC. The function takes a single argument ranging from 1 to 8, that corresponds to a fixed pin on the chip as given in the following table.

A/D Channel	Pin Number
0	A0
1	A1
2	A2
3	A3
4	A5
5	E0
6	E1
7	E2

The function itself is as follows:

```
to ad :chan
    write-reg adcon1 $80
    write-reg adcon0 ((:chan - 1) * 8) + $81
    set-bit adgo adcon0
    waituntil [not test-bit adgo adcon0]
    output ((read-reg adresh) * 256) + read-reg adresl
end
```

The **pwm** function is used to set up a Pulse-Width-Modulation signal on pin C2 of the chip. The function takes a single argument, a value from 0 to 255, and will create a fixed-frequency pulse-train with a positive duty cycle directly proportional to the argument given.

```
to pwm :val
    clear-bit 2 portc-ddr
    write-reg t2con 6
    write-reg pr2 100
    write-reg ccp1con 12
    write-reg ccpr1l :val
end
```

The **table-item** function is used in conjunction with the **table** primitive, to read indexed values out of a previously defined table. The function takes two arguments, the name of the table itself as defined with the **table** primitive, and the index location that is to be read. The first item in the table will be at location 0, and will then count up from there. The function looks up the value in the corresponding memory location, and outputs it.

```
to table-item :table :index
    output (lsh (read-prog-mem :table + (lsh :index 1)) 8)
            or (read-prog-mem :table + (lsh :index 1) + 1)
end
```

The **print** include file contains the basic functions needed for formatting and printing numbers, characters, and strings directly to the terminal.

The most basic print functions are used to print numbers directly to the terminal. The procedures **print** and **type** both just call the **print-number** procedure, putting a negative sign in front if the number is below zero. The main difference between **print** and **type** is that **print** inserts a carriage return after the number, by calling the **cr** function, which just sends the ASCII sequence needed to jump to the next line in the terminal. The **print-number** function itself breaks a large number into individual digits, and then calls **print-digit** to send the ASCII code need to print each individual character.

```
to print :n
    if (:n < 0) [put-serial 45 print-number 0 - :n cr stop]
    print-number :n
    cr
end

to type :n
    if :n < 0 [put-serial 45 print-number 0 - :n stop]
    print-number :n
end

to print-number :n
    if :n > 9999 [print-digit :n 10000]
    if :n > 999 [print-digit :n 1000]
    if :n > 99 [print-digit :n 100]
    if :n > 9 [print-digit :n 10]
    print-digit :n 1
end

to print-digit :n :d
    put-serial ((:n / :d) % 10) + 48
end

to cr
    put-serial 10 put-serial 13
end
```

Another print function has been defined which is useful for printing clock values, or any other case where a number below 10 should have a zero printed in front of it. The **print-clock** function does just that, by checking to see if the number is below the threshold, and if it is, printing a 0 before the actual number is printed.

```
to print-clock :n
    if :n < 0 [put-serial 45 print-number 0 - :n stop]
    ifelse (:n > 9)
        [print-number :n]
        [print-number 0 print-number :n]
end
```

In some cases, we want to print numbers in hexadecimal. .The **print-hex** function first prints a "$" character, and then splits the byte in half, and prints both the high and low nibbles using the **print-hex-digit** function, which simply prints the letters A through F corresponding to number 10 to 15.

```
to print-hex :n
    print-string "$
    print-hex-digit :n / 16
    print-hex-digit :n % 16
    cr
end

to print-hex-digit :n
    ifelse :n < 10
        [put-serial :n + 48]
        [put-serial :n + 55]
end
```

One last key function that is commonly used is **print-string**, which will print a character-based text string directly to the terminal. The procedure itself starts by pointing a variable to the beginning of the string, and then just sends out one character at a time until it reaches a value of "0" signifying the end of the string.

```
to print-string :n
    setnn :n
    loop
    [
        if (read-prog-mem nn) = 0 [stop]
        put-serial read-prog-mem nn
        setnn nn + 1
    ]
end
```

Finally, we have defined a **hi** function which just prints out a welcome string to the terminal. While not critically important for most programs, it serves as a good power-on test for the Tower, and can be called whenever you're feeling lonely and need a digital friend.

```
to hi
    print-string "|     Welcome to Tower Development Kit| cr
end
```

The **address** include file contains the basic functions needed for scanning the Tower, and for changing the address of any layer connected to it.

The **get-board-id** function is used to determine which specific board is located at a given address. The function itself takes a single argument, the address to query, which must be an even number between 0 and 254. The return value will be the hardware identifier, which can be looked up in the **get-board-name** function, also present in this file, to obtain the name of the actual board. The function sends two values to the actual layer, a "255" to enter address mode, and a "0" indicating that we want to obtain the hardware identifier. The identifier is then read back from the board, ignoring the first value back, which will just be a "1" indicating that a single byte is going to follow.

```
to get-board-id :address
    i2c-start
    i2c-write-byte :address
    i2c-write-byte 2
    i2c-write-byte 255
    i2c-write-byte 0
    i2c-stop
    i2c-start
    i2c-write-byte :address + 1
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The **set-board-address** function is used to change the address of any layer plugged into the stack. The function takes two arguments, the old address, and the new address, both of which can be values from 0 to 254. However, it should be noted that “0” is the general-call-address, meaning that no layer should actually have it as its address. If a “0” is sent as the argument for the new address, the board will return to its default address, which is always defined to be exactly twice its hardware identifier. The function itself sends three values to the layer, a “255” to enter address mode, a “1” indicating that we want to write a new address, and then the new address itself, which must be an even number from 0 to 254.

```
to set-board-address :oldaddress :newaddress
      i2c-start
      i2c-write-byte :oldaddress
      i2c-write-byte 3
      i2c-write-byte 255
      i2c-write-byte 1
      i2c-write-byte :newaddress
      i2c-stop
end
```

The **scan-tower** function is used to check every address on the Tower, and when a board is found, it prints a string to the terminal indicating that fact. Starting at address 0, each address is individually checked to see if a board is present, by the **i2c-check** primitive. If a board is present, the hardware identifier is obtained, and then looked up by the **get-board-name** function. The resulting string is then printed to the terminal, followed by a statement indicating the address at which it was found. This process then repeats until the entire address space has been checked.

```
to scan-tower
    setn 0
    repeat 128
    [
        i2c-start
        setnn i2c-check n
        i2c-stop
        if (nn = 1)
        [
            print-string get-board-name get-board-id n
            print-string "| layer located at address |
            type n cr
        ]
        setn n + 2
    ]
    print-string "|Done!|
    cr
end
```

The **get-board-name** function is used to look up the name of a board, given its hardware identifier. The name is output directly as a string to the calling function. If no match is found, the phrase “Unknown” is returned.

```
to get-board-name :n
    if (:n = 1) [output "Bus stop]
    if (:n = 2) [output "Servo stop]
    if (:n = 3) [output "IR stop]
    if (:n = 4) [output "Voice stop]
    if (:n = 5) [output "Sensor stop]
    if (:n = 6) [output "EEPROM stop]
    if (:n = 7) [output "Clock stop]
    if (:n = 8) [output "CompactFlash stop]
    if (:n = 9) [output "I2C stop]
    if (:n = 10) [output "Tricolor stop]
    if (:n = 11) [output "Display stop]
    if (:n = 12) [output "Freakbaby stop]
    if (:n = 13) [output "Motor stop]
    if (:n = 15) [output "PICProto stop]
    if (:n = 16) [output "Serial stop]
    if (:n = 17) [output "Elmo stop]
    output "Unknown
end
```

Examples of Use

Let’s start by saying “Hi” to our foundation:

```
hi
>     Welcome to Tower Development Kit
```

Let’s do a little simple math, just to show that everything is in working order:

```
print 3 + 5
> 8
```

How about we try out some basic pin I/O. For this example, we'll say that we have an LED connected to pin B0. Let's turn it on:

```
set 0 portb
```

Now let's make the LED flash by writing a simple loop:

```
loop [set 0 portb wait 10 clear 0 portb wait 10]
```

This code turns the pin on, waits a second, turns it off, waits another second, and then keeps repeating. Before we do anything else, make sure to press the white button to stop the program, or we won't be able to communicate with the Foundation.

Let's play around with addressing. We can start by scanning the Tower like this:

```
scan-tower
> Bus layer located at address 2
> IR layer located at address 6
> Sensor layer located at address 10
> EEPROM layer located at address 12
> Clock layer located at address 14
> CompactFlash layer located at address 16
> I2C layer located at address 18
> Tricolor layer located at address 20
> Display layer located at address 22
> Motor layer located at address 26
> PicProto layer located at address 30
> Servo layer located at address 58
> Done!
```

If we want to change the address of the Servo layer to "4", we can say the following:

```
set-board-address 58 4
```

Now let's scan the Tower again, just to make see that it worked:

```
scan-tower
> Bus layer located at address 2
> Servo layer located at address 4
> IR layer located at address 6
> Sensor layer located at address 10
> EEPROM layer located at address 12
> Clock layer located at address 14
> CompactFlash layer located at address 16
> I2C layer located at address 18
> Tricolor layer located at address 20
> Display layer located at address 22
> Motor layer located at address 26
> PicProto layer located at address 30
> Done!
```