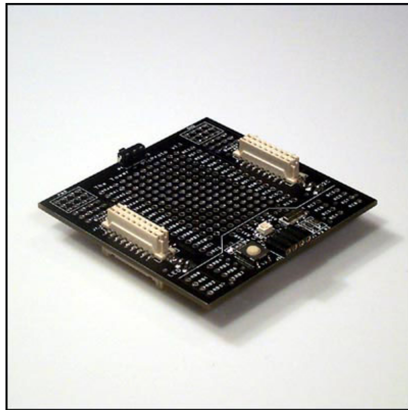
PicProto Layer

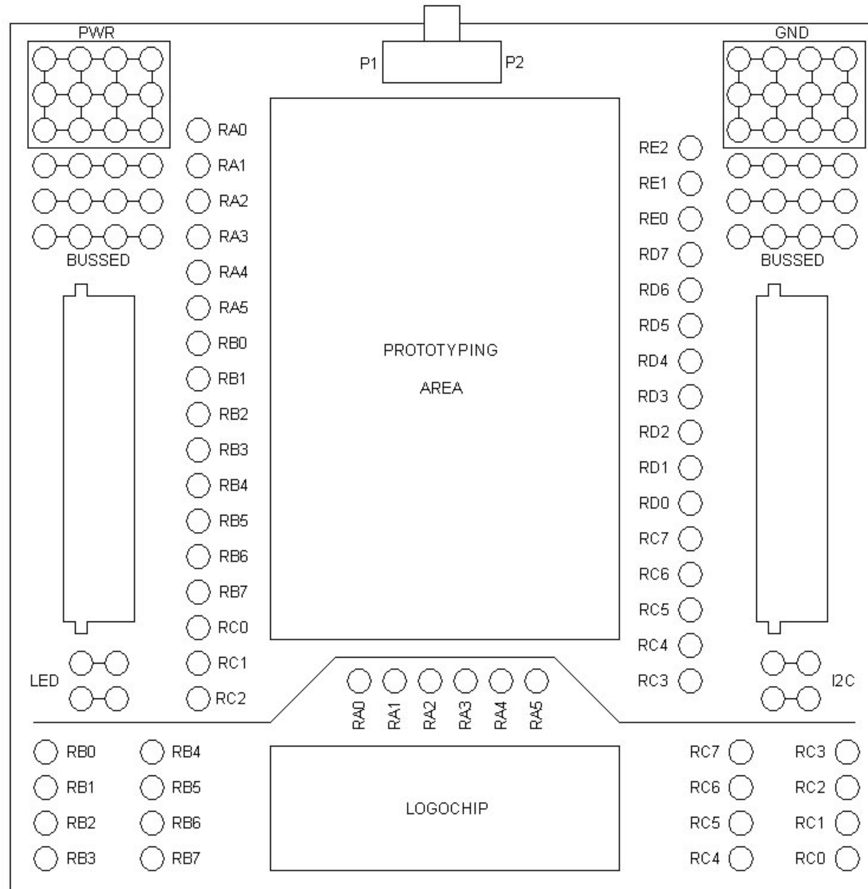


Description

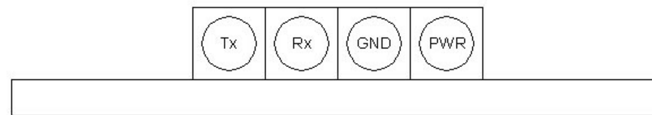
The PicProto layer is designed to give users the ability to make their own layers for the Tower system. In addition to the blank perforated board and tap-points for every I/O pin on the Foundation, this layer has a LogoChip on-board, which is another PIC Processor running the same virtual machine as the foundation. With its I/O pins accessible as well, and a programming header on-board, anyone can easily build layers that communicate directly with the Foundation via the I^2C protocol and add new functionality to the system as a whole.

Hardware Detail

The Proto layer has easy-to-access tap-points for all 33 I/O pins passed up from the PIC Foundation and all 22 I/O pins from the LogoChip, as well as common distribution busses, and two power and ground connection blocks. The PWR points can be switched between the primary and secondary power busses by the switch at the top of the board. The I^2C connection between the LogoChip and the Foundation can be jumpered if desired, as can the bicolor LED also on-board. An overview of the board is shown below:



The LogoChip 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.

Layer Code

Since the PicProto layer can be built up and configured in any way the users desires, there is no actually pre-existing layer code for it. All communications with anything built on the layer are done directly with the processor on the foundation.

In order to use the I^2C protocol to communicate with the Foundation, there are four new important functions that can be called from a Logo program. One allows you to check for new data, one gets data that has been received in the buffer, one puts data into the buffer, and the last one sets a flag saying that the buffer has been filled and is ready to be sent back.

It's important to first understand a bit about how the I^2C protocol is working on the Tower layers. When the Foundation sends out information over the bus, every layer checks to see if its address matches the one the Foundation wants to talk to.

If it matches, the next byte that comes from the Foundation tells the layer how many bytes it's going to be sending. The layer keeps grabbing bytes and putting them in the receive buffer until it's gotten the number that it was told it was receiving. All of that work is done in the background, but as soon as it's completed, a flag inside the chip is set saying that new data is present. The flag can be checked using the **new-i2c?** primitive like this:

```
print new-i2c?
> 1
```

When that function returns a value of "1", we can go ahead and start looking at the data that came in. To grab a byte out of the receive buffer, we use the **get-i2c** primitive. The primitive takes a single argument, which is the buffer location to get the data out of. Location 0 will have the number of arguments that were sent, not including itself, and the actual arguments will begin in location 1. For example, if we sent 3 arguments to the layer, location 0 would have a "3" in it, and the actual data would be in locations 1, 2, and 3. Let's see what the first argument is:

```
print get-i2c 1
> 14
```

It's that simple to get data from the Foundation. Once we have the data, we probably want to do something based on what arguments were sent. Often, we'll want to send data back to the Foundation.

To return values to the Foundation, all of the layers have a separate transmit buffer. The transmit buffer operates almost identically to the receive buffer, in that its first location contains the number of values

being sent back, followed by the values itself. To put data in the transmit buffer, we use the **put-i2c** primitive. Two arguments are needed for **put-i2c**, the buffer location, and the value to put there. Let's say we want to send back the number 23 to the Foundation. To do so, we can write:

```
put-i2c 1 23
```

We put the value in the first location of the buffer, but there's something we forgot. We need to tell the Foundation how many data values it's going to be getting. In this case, we're only sending a single byte back, so we should store a "1" in location 0 of the transmit buffer, like this:

```
put-i2c 0 1
```

But there's still one more important step. The I^2C protocol is master-driven, meaning that that layers don't send back data until the Foundation specifically asks for it. There could be a potential problem if the Foundation tried to grab data when the transmit buffer was only half-filled. To solve that problem, there's another flag inside the chip, which tells the Foundation whether or not it's ready to send back data.

The layer will just tell the Foundation that it's not ready, and to keep asking until the data is actually complete. After we've filled the buffer, we need to set that flag using the **i2c-ready** primitive. No arguments are needed, and it can just be called as follows:

```
i2c-ready
```

Now the Foundation is free to grab all of the data out of the buffer, and continue about its business. The flag will be automatically cleared as soon as the data has all been sent.

Examples of Use

To make a new layer for the Tower using the PicProto layer, two pieces of code need to be written. There's one part that runs on the PicProto layer itself, and another that has to run on the Foundation in order to talk to it.

For a simple example of communicating between layers and the Foundation, let's program the layer to take two arguments, add them together, and send them back to the Foundation. The code for the PicProto would look something like this:

```
on-startup [logochip-add]

to logochip-add
    loop
    [
        waituntil [new-i2c?]
        setn (get-i2c 1) + (get-i2c 2)
        put-i2c 1 n
        put-i2c 0 1
        i2c-ready
    ]
end
```

This function has a loop, sits at the beginning waiting for an I2C communication. As soon as one is received, we're ready to add. We set the global variable "n" equal to the sum of the first two arguments. We then store "n" in the first location of the transmit buffer, and store a "1" in location 0, telling the chip

that we're only sending back the one value. Finally, we set the ready-flag, and go back to the top of the loop to wait for another query.

The **on-startup** statement at the top just ensures that this code will start running when the PicProto layer is powered on. The layer will only respond to the Foundation if the program is running. Remember, you can't talk to a LogoChip or Foundation while it's running code, so you need to stop the program by pushing the white button before you can reprogram it with something new.

Now that we've got that half of the code taken care of, let's write the function that will run on the Foundation. This will be pretty much identical to what you'd find in the include files for any of the other layers, since it's using the exact same method to communicate with the PicProto layer as it does with any other. The add function we want to write should resemble this:

```
to add :n1 :n2
    i2c-start
    i2c-write-byte $1e
    i2c-write-byte 2
    i2c-write-byte :n1
    i2c-write-byte :n2
    i2c-stop
    i2c-start
    i2c-write-byte $1f
    waituntil [(i2c-read-byte 1) = 1]
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

This function opens communication to the PicProto layer, whose default address is $1e, and then sends our two arguments to be added. Remember, the "2" is just saying that two arguments are going to follow. After that, we need to open communication in the other direction by sending out the address $1f. This is where that ready-flag becomes important. That **waituntil** statement will sit there until it gets back a "1" from the layer, indicating that one byte is going to follow. Until the ready-flag has been set on the layer, **i2c-read-byte** will just keep receiving 255's. As soon as we know the data is ready, we grab it and store it, stop the I2C communication, and then output it to whatever function called this one. *(This function was written for the PIC Foundation. On other foundations, the code would differ slightly, due to the presence of local variables.)*

There is one very important note that we should mention. In most cases, you would never just sent the arguments to your function without a preceding byte for a dispatch routine. For example, we could make our first argument be either a "0" or a "1", where a "0" would indicate that it wanted to add the two numbers together, and a "1" would tell the layer to perform a subtraction instead. After that first byte, the two arguments themselves would be sent just as before. Even if a layer only performs one function, it's still a good idea to send a "0" or some other number first.

The reason why, is because the first argument carries special weight. If the first argument happens to be a 255, the layer enters addressing mode, in which the address of the board can be easily changed. While that's not a problem that can't be fixed, it can make things very frustrating to debug if your program keeps changing the address of the layer that it's trying to communicate with.