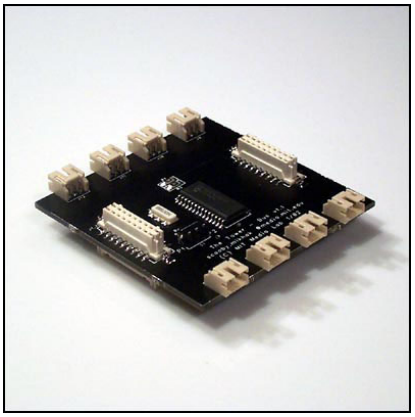
Cricket Bus Layer

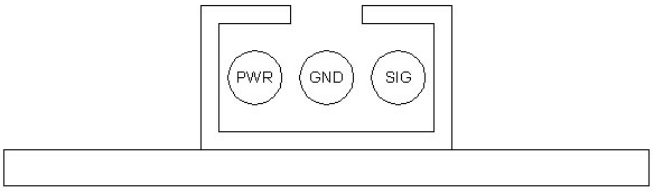


Description

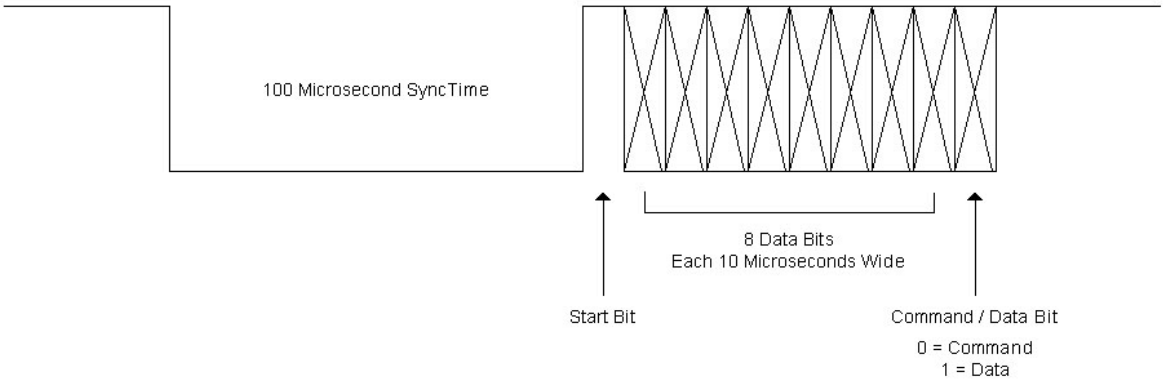
The Cricket Bus Layer has eight bus-ports, and is capable of communicating via a proprietary one-wire protocol with any device in the Media Lab's Cricket System, as well as with Crickets themselves.

Hardware Detail

The Cricket Bus layer uses a PIC to generate the tightly timed one-wire bus signals. The connector is a 3-pin JST connector, as found on all devices in the Cricket system. The pin configuration of the port is shown below.



The protocol is master-driven, and devices on the bus will only respond when directly spoken to. The bus protocol itself consists of eight, 10-microsecond long bits, preceded by a start bit and a 100-microsecond synchronization time to give all devices on the bus a chance to respond if they are in the middle of an operation. A diagram of the Cricket Bus Protocol is shown below:



It should be noted that all eight bus-ports on the layer are tied together. The presence of multiple ports is just to allow for ease of wiring. Since some bus devices consume a great deal of power, it is possible to power the Bus lines off of either the primary or secondary power bus, as selected by the switch on the layer.

Layer Code

The include file for the Bus layer contains a variety of functions for talking to some of the more commonly used Cricket bus devices. Currently, functions have been written for talking to the Hexadecimal Display, Optical Distance Sensor, Clap Sensor, and the Voice Recorder. Additionally, there is a function that allows users to set a delay time between bytes being sent out over the bus, since some bus devices are unable to keep up if data is sent at full speed. Even though functions have only been written for a handful of bus devices, upon inspection you can see how easy it is to write functions to talk to any device in the Cricket system should the need arise. *(All of these functions are written for the PIC Foundation. The include files used with other foundations differ slightly, due to the presence of local variables.)*

The **hex-display** function is used to display a value on a Hexidecimal Display bus device. It takes one argument, the number to display, which can be a value from 0 to 9999. Within the function, a total of four values are actually sent to the Cricket Bus layer. First, a "0" is sent, indicating that we wish to talk to a bus device, but are not expecting a response back from it. After that, we send the address of the bus device, in this case $70. Then, we send the value to display, split into high and low bytes, since the communication is only eight bits at a time.

```
to hex-display :n
    i2c-start
    i2c-write-byte $02
    i2c-write-byte 4
    i2c-write-byte 0
    i2c-write-byte $70
    i2c-write-byte :n / 256
    i2c-write-byte :n % 256
    i2c-stop
end
```

The **distance** function is used to take a reading from an Optical Distance Sensor bus device. It takes no, arguments, but sends a total of two values to the layer. First a "1" is sent, indicating that we want to talk to a bus device, and that we do expect a response back from it. Then the address is sent, which for the Optical Distance Sensor is $55. After the request is sent to perform the sensor reading, the result is read out of the layer's transmit buffer as a single byte. Actually two bytes are read out, but the first is ignored, as it is known to be a "1", representing the number of arguments to follow.

```
to distance
    i2c-start
    i2c-write-byte $02
    i2c-write-byte 2
    i2c-write-byte 1
    i2c-write-byte $55
    i2c-stop
    wait 2
    i2c-start
    i2c-write-byte $03
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The **clap** function is used to take a reading from an Optical Distance Sensor bus device. It takes no, arguments, but sends a total of three values to the layer. First a “1” is sent, indicating that we want to talk to a bus device, and that we do expect a response back from it. Then the address is sent, which for the Clap Sensor is $6a. Next, we send a $01, because that code is needed to ask the device if it’s heard a clap (or other loud noise), since the last time we asked it. After the request is sent to perform the sensor reading, the result is read out of the layer’s transmit buffer as a single byte. Actually two bytes are read out, but the first is ignored, as it is known to be a “1”, representing the number of arguments to follow.

```
to clap
    i2c-start
    i2c-write-byte $02
    i2c-write-byte 3
    i2c-write-byte 1
    i2c-write-byte $6a
    i2c-write-byte $01
    i2c-stop
    wait 2
    i2c-start
    i2c-write-byte $03
    ignore i2c-read-byte 1
    seti2c-byte i2c-read-byte 0
    i2c-stop
    output i2c-byte
end
```

The **voicerec-play** function is used to play a voice sample that has been previously recorded on the Voice Recorder bus device. It takes one argument, the sample number to play, which can be a value from 0 to 255. Within the function, a total of three values are actually sent to the Cricket Bus layer. First, a “0” is sent, indicating that we wish to talk to a bus device, but are not expecting a response back from it. After that, we send the address of the bus device, in this case $54. Then, we send the number of the sample to play, and playback should begin immediately.

```
to voicerec-play :n
    i2c-start
    i2c-write-byte $02
    i2c-write-byte 3
    i2c-write-byte 0
    i2c-write-byte $54
    i2c-write-byte :n
    i2c-stop
end
```

The **voicerec-clear** function is used to erase all of the voice samples that have been previously recorded on the Voice Recorder bus device. It takes no arguments, and within the function, a total of three values are actually sent to the Cricket Bus layer. First, a “0” is sent, indicating that we wish to talk to a bus device, but are not expecting a response back from it. After that, we send the address of the bus device, in this case $54. Then, we send a 255, which tells the bus device itself to reset its memory. The green LED on the Voice Recorder will flash to indicate that the erase command was successful. It is important to note that it is necessary to power-cycle the Voice Recorder itself after this command is sent, or the memory change will not actually take effect.

```
to voicerec-clear
    i2c-start
    i2c-write-byte $02
    i2c-write-byte 3
    i2c-write-byte 0
    i2c-write-byte $54
    i2c-write-byte 255
    i2c-stop
end
```

The **set-bus-delay** function is used to increase the delay between bytes sent out over the bus signal line. While not needed for any of the devices that functions have currently been written for, there are some devices in the system which cannot take data as quickly, and need to have their delay increased. By default, the delay between bytes is about 10 microseconds. This function takes a single argument, the desired time delay (in milliseconds), which can be a value from 0 to 255. Two bytes are sent to the actual layer, a "2" indicating that the delay time is being changed, immediately followed the new desired timing delay.

```
to set-bus-delay :delay
    i2c-start
    i2c-write-byte $02
    i2c-write-byte 2
    i2c-write-byte 2
    i2c-write-byte :delay
    i2c-stop
end
```

Examples of Use

To use the **Cricket Bus** layer, lets say that we've got all of the above-mentioned bus devices plugged into it. While that probably won't be true for the vast majority of the people using it, we'll look at examples of how to talk to each different device.

Talking to the Hexadecimal Display device is as simple as saying:

```
hex-display 1234
```

The number 1234 should now appear on the display itself. Let's now use the display to display a reading from the Optical Distance Sensor:

```
hex-display distance
```

You should see a single value appear on the display. That probably wasn't all that interesting, since it only happened once. Let's make it loop:

```
loop [hex-display distance wait 10]
```

Now, the Hexadecimal Display should be updating every second with a new value from the Optical Distance Sensor. While the wait statement in the loop is not necessary, it's in there so that we have a chance to actually read the number before it changes too quickly.

Now let's try using the Clap Sensor. We can use it just like we used the Optical Distance Sensor, and display its output on the Hexadecimal Display device like this:

```
loop [hex-display clap wait 10]
```

Like before, the display will update every second. In this case however, it will always be a "0" or a "1". A value of "0" means that it hasn't heard anything in the last second, and a "1" means that it has.

Let's add some sound to our project. For the sake of this example, assume that we have two voice samples recorded onto the Voice Recorder device. To record samples, you hold down the "Rec" button on the device itself and speak into the microphone. Let's say that the first sample is "Hello", and the second is "Goodbye". Here's how we tell the board to say "Hello":

```
voicerec-play 0
```

Now, let's do something a little more complicated. Let's check the distance sensor every 5 seconds, and if someone is close, we'll say "Hello", and if nobody is around, we'll say "Goodbye."

```
loop [if (distance < 100)[voicerec-play 0][voicerec-play 1] wait 50]
```

In this case, we set the threshold on the Optical Distance Sensor to be 100, but that is something that should usually be determined experimentally given the particular application.

Finally, just to show how to use it, let's change the delay between bytes being sent on the bus signal line:

```
set-bus-delay 10
```

There should now be a 10 ms delay between each byte sent over the bus. Everything that worked before should still work, it just might take a little longer to happen.