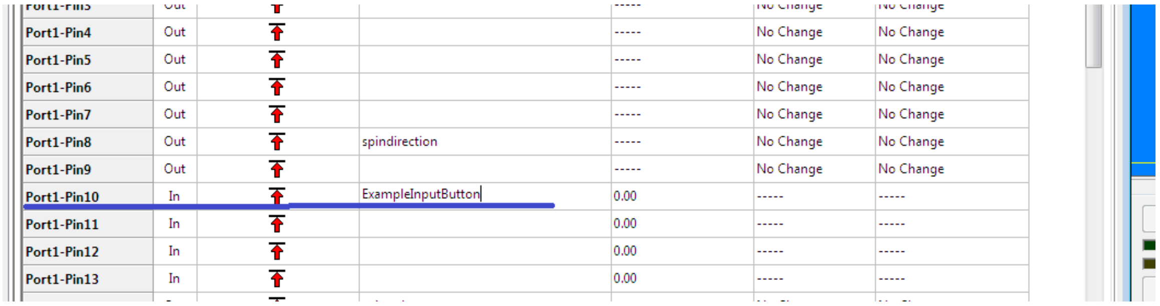
Using Mach4 Signals-Intro

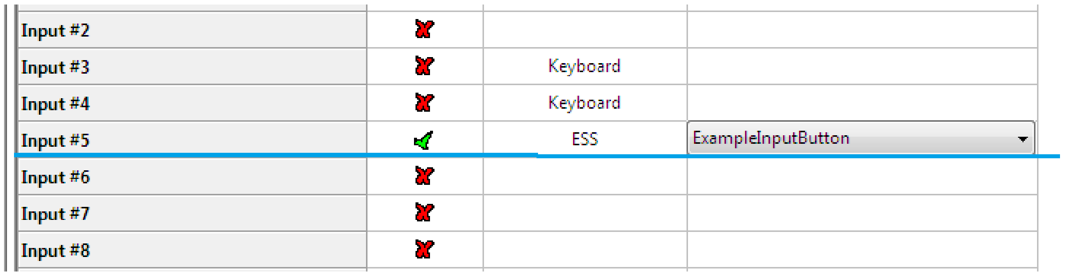
The overwhelming majority of new Mach4 users encounter the difficulty of 'How to monitor an external signal, be it a switch or a button and have Mach respond to it?'

The preliminary is to connect the switch or button to a spare input on your controller/BoB combination. Then using your controller plugin, and possibly the Mach Control plugin, to logically connect that input to one of Machs predefined signals. For this example I'm going to use ISIG_INPUT5.



Port1-Pin4	Out	↑		-----	No Change	No Change
Port1-Pin5	Out	↑		-----	No Change	No Change
Port1-Pin6	Out	↑		-----	No Change	No Change
Port1-Pin7	Out	↑		-----	No Change	No Change
Port1-Pin8	Out	↑	spindirection	-----	No Change	No Change
Port1-Pin9	Out	↑		-----	No Change	No Change
Port1-Pin10	In	↑	ExampleInputButton	0.00	-----	-----
Port1-Pin11	In	↑		0.00	-----	-----
Port1-Pin12	In	↑		0.00	-----	-----
Port1-Pin13	In	↑		0.00	-----	-----

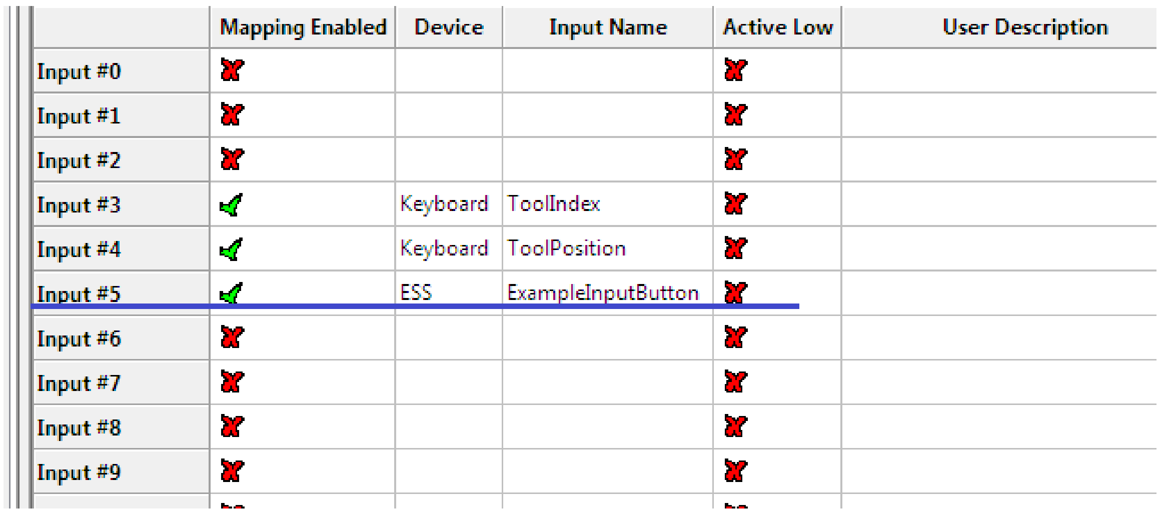
Pin Configuration Page (Ethernet SmoothStepper Plugin)



Input #2	✗		
Input #3	✗	Keyboard	
Input #4	✗	Keyboard	
Input #5	✓	ESS	ExampleInputButton
Input #6	✗		
Input #7	✗		
Input #8	✗		

Input Signal Page (Ethernet SmoothStepper Plugin)

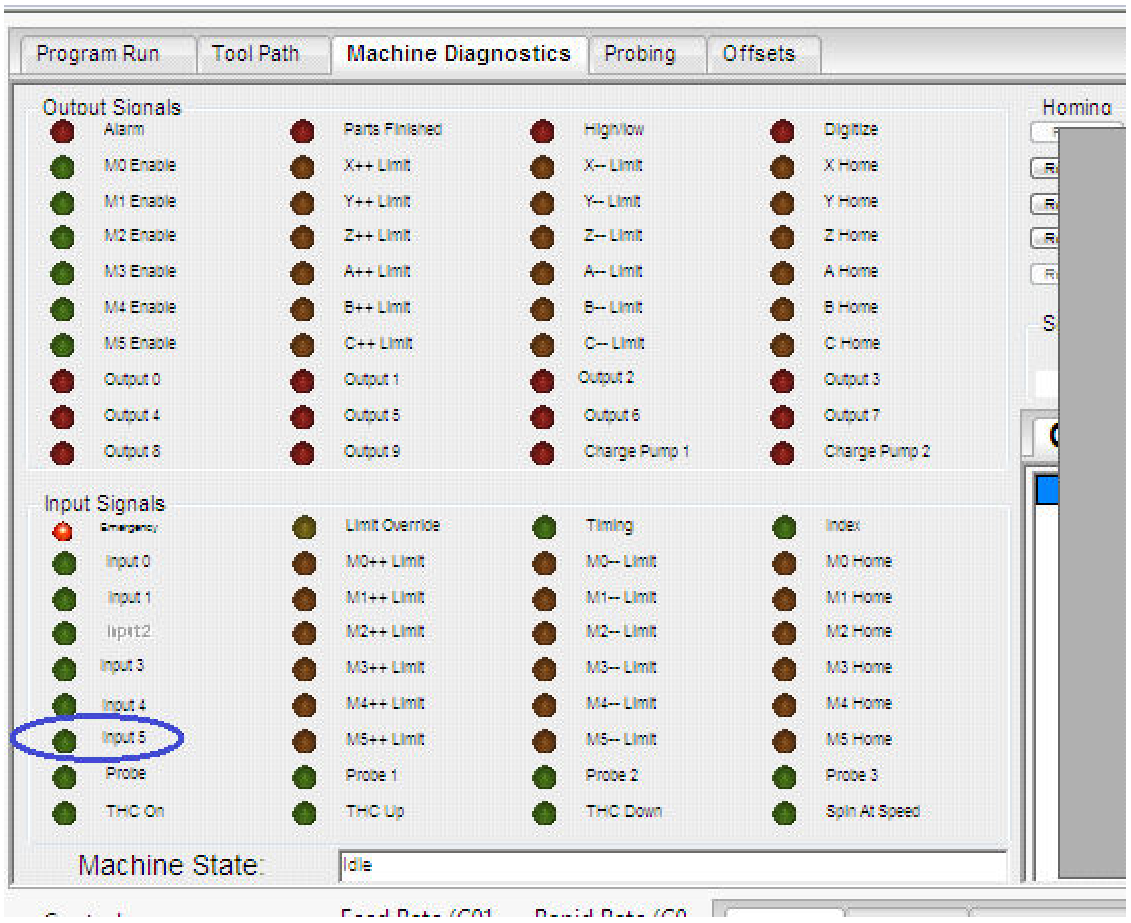
You should confirm that Machs Control plugin has been udated, the Ethernet SmoothStepper plugin automatically updates the Control plugin, but not all manufacturers plugins do so.



	Mapping Enabled	Device	Input Name	Active Low	User Description
Input #0	✗			✗	
Input #1	✗			✗	
Input #2	✗			✗	
Input #3	✓	Keyboard	ToolIndex	✗	
Input #4	✓	Keyboard	ToolPosition	✗	
Input #5	✓	ESS	ExampleInputButton	✗	
Input #6	✗			✗	
Input #7	✗			✗	
Input #8	✗			✗	
Input #9	✗			✗	

Mach Control Plugin, Input Signal Page

I elected to use Input#5 with malice aforethought because if you look at Mach4's Diagnostic page there are LEDs reflecting the status of some of Machs signals, Input#5 amongst them. Thus if you flick the switch or button you should see the inut LED alternately light then unlight.



Mach4 Diagnostic Page-Input Signal LEDs

This is a very convenient and useful means to check the switch/button and its wiring, the pin asignment of your controller/BoB and the logical connection of Machs signal (ISIG_INPUT5).

The prelimnary is complete....the question is 'how we get Mach to recognize our newly configured signal?'.   There are two methods of use. The first method is appealing because it is (comparatively) straight forward, but as we will see later inefficent and not preferred despite that it works.

The first method is to use the PLC script. The PLC script is a bunch of Lua code that runs every few milliseconds in Machs core. The code within the PLC script is deliberately kept lean and mean so that it executes smartly so that it does not 'hang Mach up'.

We can use it to monitor an input signal like our button.

```
local inst=mc.mcGetInstance()
local input5Handle=mc.mcSignalGetHandle(inst,mc.ISIG_INPUT5)
local input5State=mc.mcSignalGetState(input5Handle)
```

If you have never encountered Mach4/Lua code before there may be some features that look weird. I will explain a few of them. If you are already familiar with them skip over it.

The first is what does **local** at the beginning of each line mean? Lua is a self memory managed language. When you declare variables for use they get allocated memory. When they are no longer required (go out of scope) they will be 'Garbage Collected' and the memory released. When you declare a variable as **local** when it goes out of scope the Garbage Collector knows that its safe to remove the variable from memory. If that same variable is not delcared **local** then it will be a global variable available througout the Lua chunk in which it resides. That consumes memory that will not be garbage collected. Also each time the variable is referenced Lua has to stop and resolve and acess the reference which might be in another page of memory. The rule in Lua is variables should all be **local** unless you have a specific reason to have the variable treated as global.

The second question is 'What is **inst** in the code'. While it is not implemented yet it is intended that multiple copies of Mach4 can run at the same time. Thus each time you address Machs core you need to specify which of the several instances you mean. At the current time Mach has only one instance, instance=0, but you should get into the habit of interrogating Mach to get the current instance number and declare a local variable **inst** to store it.

'Why are all the function calls prefaced by **mc.mc.....**?' Lua has only one data structure, the table. It is very flexible despite its apparent simplicity. To access a member of a table you prepend the name of the table, a period, then the name of the table member or entry. For example: **myTable.first(.....)** acesses a table called **myTable** and addreses the function **first()** in that table. So the syntax of **mc.mcSignalGetHandle()** say, now becomes clear. All of Machs API functions are in a table called **mc** (Mach core) and the particular API function you have called is **mcSignalGetHandle()**. Note that **mc.ISIG_INPUT5** is just another call to the **mc** table but in this instance it is referencing a plain number (**ISIG_INPUT5**). This shows one of the strengths of Lua's table, it can have different types as entries in the table. As in this example one table entry is a function and another entry in the same table is a number, very clever!
It also is an example of the principle of a 'function as a first class value'. It means that a whole function can be bandied around as if it were just another number. This apparetly simple principle is a major part of what makes Lua so clever and will be used and reused throughout Lua.

'What is **mcSignalGetHandle(inst,mc.ISIG_INPUT5)** all about?'. In Lua, in fact probably all computer languages, a variable like **ISIG_INPUT5** has a specific location in memory and the computer wants to know the address or number of that location, the name we humans give to the data is immaterial to the computer. Thus the statement we are considering is aksing Lua to get the current memory location (or handle) of the variable **ISIG_INPUT5** and storing the address in the variable **input5Handle**.

So the last line of code:

local input5State=mc.mcSignalGetState(input5Handle)

means that the state (true or false/on or off) of the memory location described by the number **input5Handle** is assinged to the variable **input5State**. It will be

'0' if the signal (swith or button) is off or '1' if the signal (switch or button) is on.

We could now use the variable **input5State** to decide whether to turn a pump on say. This code in the PLC script will run every few milliseconds or hundreds of times a second. If this is the switch to turn on your coolant pump, you'll turn it on and leave it on until you part has finished. In the mean time the code in the PLC script has run many millons of times. Really we only need to do anything once when the switch is turned on and then again when the switch is turned off. Running the code millions of times just waiting for you to operate the switch is a waste of computing power.

This method of monitoring inputs is called 'polling'. It will be familiar to anyone versed in Mach3. Mach4 introduces a new and clever way to do the same thing WITHOUT all that unecessary code overhead. Its an elegant method but does take some thinking to work it out.

Mach has a whole bunch of signals, some you have seen like **ISIG_INPUT5** and others like **OSIG_OUTPUT43** for example. These signals are intended for input and output to Mach respectively. But they are by no means all of the signals Mach has or uses. Machs core for instance may use a signal which we never see to tell the motion controller to be ready to recieve an instruction from the trajectory planner. In short there could be hundreds or thousands of signals between Machs core and all the devices and their plugins in a second.

Each time ANY of those signals changes state, either turns on or off, Mach runs the Signal Script. Mach does not know what to do with any such change all it knows is the number of the signal called **sig** and its new state called **state**.

If for instance you turn on the switch Mach will detect that a signal has changed state, it has a signal number 6 and its state is true or '1'. **ISIG_INPUT5** is the human readable version of the number 6. Machs Signal Script will run:

```
if SigLib[sig] ~= nil then
       SigLib[sig](state);
end
```

Note this is the Signal Script as it ships from NFS. You can if you wish or require modify it.

This introduces a new variable **SigLib[sig],** called the Signal Library Table. Specifically the **if** statement asks 'is there an entry in the Signal Library Table that corresponds to **sig,** if so, that is, not equal to nil, then execute the code in the table at the entry **[sig]**. If not, that is to say there is no entry, we don't care about this signal then Mach goes back to whatever it was doing before it ran the Signal Script.

The Signal Library Table is a global table near the top of the screen load script and is visible throughout the Lua GUI chunk. We wish to put an entry into the table because we are interested in **sig**=6, we want it to turn on our pump.

```
-- Signal Library
---------------------------------------------------------------
SigLib = {
     [mc.OSIG_MACHINE_ENABLED] = function (state)
       machEnabled = state;
       ButtonEnable()
end,
[mc.OSIG_JOG_CONT] = function (state)
       if( state == 1) then
```

```
            scr.SetProperty('labJogMode', 'Label', 'Continuous');
            scr.SetProperty('txtJogInc', 'Bg Color', '#C0C0C0');--Light Grey
            scr.SetProperty('txtJogInc', 'Fg Color', '#808080');--Dark Grey
        end
end,
[mc.OSIG_JOG_INC] = function (state)
        if( state == 1) then
             scr.SetProperty('labJogMode', 'Label', 'Incremental');
             scr.SetProperty('txtJogInc', 'Bg Color', '#FFFFFF');--White
             scr.SetProperty('txtJogInc', 'Fg Color', '#000000');--Black
      end
end,
[mc.OSIG_JOG_MPG] = function (state)
        if( state == 1) then
             scr.SetProperty('labJogMode', 'Label', '');
             scr.SetProperty('txtJogInc', 'Bg Color', '#C0C0C0');--Light Grey
             scr.SetProperty('txtJogInc', 'Fg Color', '#808080');--Dark Grey
             --add the bits to grey jog buttons becasue buttons can't be MPGs
        end
end,
--M6 messagebox
[mc.OSIG_TOOL_CHANGE] = function (state)
        local selectedtool = mc.mcToolGetSelected(inst)
           local currenttool = mc.mcToolGetCurrent(inst)
           if (selectedtool ~= currenttool) then
             if( state == 1) then
```

```
                    mm.ToolChangeMsg("A tool change has been requested via M6.
Change your tool then press Cycle Start to continue!", "Tool Change Active!")
            end
      end
end
}
```

Note that this is the Signal Library, or close to,as it ships from NFS.

The first line:

SigLib = { is the opening statement of the defintion of the table **SigLib** as indicated with the opening **{** . Note also the very last line, **}** , that is the closing statement of the table definition with a series of comma separated entries.

The next lines are:

```
 [mc.OSIG_MACHINE_ENABLED] = function (state)
      machEnabled = state;
      ButtonEnable()
end,
```

This creates an entry in the Signal Library called **[mc.OSIG_MACHINE_ENABLED]** . Thus if the signal **OSIG_MACHINE_ENABLED** ever changes state then the function will run, in this case it upadtes a variable **machEnabled** and then runs the function **ButtonEnable()** defined elsewhere. Note also the comma **end,** after the function end statement, it signifies the end of this entry or member of the table.

The next lines introduce yet another entry, and when it is declared it will be closed with a trailing comma. Its very easy to forget the comma and you get some strange errors.

We want to create a new entry in the Signal Library Table:

```
[mc.ISIG_INPUT5]=function(state)
        if (state==1)then
                --run some code to turn   the pump on
        else
                --run some code to turn the pump off
        end
end,
```

So if the signal **ISIG_INPUT5** changes state then the function will run. Note that it tests the value of **state**, if its on (==1) then it will run the code required to turn on. It might be that we have just flicked the switch to off, and so **state** will be (==0) and thus we want to turn the pump off.

The only time the function will run is when the signal changes state, either we turn the switch on or later in the day turn it off again. Whereas the code in the PLC script will have run many millions of times.

It is an efficient and elegant way to monitor an input. It relies on several key principles of Lua.

1) Lua detects a change in any and all of it defined signals

2) Mach has a Signal Script that runs whenever a defined signal changes state

3) The Signal Script interrogates the Signal Library table (SigLib) for an entry or member that has the same signal number

4) If so the function executes, the function being treated as a first class value.