Mach4 CNC Controller Lua Scripting Guide 10/26/22

Table of Contents

Introduction

The purpose of this manual is to teach the basics of scripting in Mach4 using the Lua interface. Scripts add functionality to Mach4 by providing the user with an interface to create custom M codes, macros to control tool changers and other custom accessories, create programming wizards, custom button functions, and much more. This manual will provide some basic programming knowledge as it pertains to creating scripts for Mach4. For more advanced Lua programming there are a variety of resources available online and in print.

Before You Begin

Any machine tool is potentially dangerous. Computer controlled machines are potentially more dangerous than manual ones because, for example, a computer is quite prepared to rotate an 8" unbalanced cast iron four-jaw chuck at 3000 rpm, to plunge a panel-fielding router cutter deep into a piece of oak, or to mill away the clamps holding your work to the table. Because we do not know the details of your machine or local conditions we can accept no responsibility for the performance of any machine or any damage or injury caused by its use. It is your responsibility to ensure that you understand the implications of what you design and build and to comply with any legislation and codes of practice applicable to your country or state. If you are in any doubt, be sure to seek guidance from a professionally qualified expert rather than risk injury to yourself or to others.

What is Mach4?

Mach4 is software that operates on a personal computer to create a powerful and cost efficient CNC controller. It makes up one small piece of a computer numerical control (CNC) machine. Machines can range from basic mills and lathes to wood routers, plasma cutters, multi axis machining centers, quilting machines, anything requiring motion control. The system is capable of interpreting multiple programming languages, the default and most common being G code, to provide instructions for machine movement and other functions. These instructions are passed to an external motion device which in turn controls all the inputs and output signals and motion.

Mach4 is designed to be flexible and adaptable to a wide variety of machines. Part of this flexibility is the ability for hardware and software developers to create addons or plugins for Mach4 to expand its capabilities. Addons are small programs installed into the Mach4 directory that give Mach the ability to talk to hardware devices such as motion controllers and pendants, communicate with other software, add additional wizards or conversational machining functions, or anything a developer can dream up. Addons to Mach4 are so diverse it would be impossible to cover them in this manual. The developer should provide detailed information on the installation, configuration and use of their addon or plugin.

What is a Mach4 Script?

Scripts in Mach4 are written in the Lua programming language. Users and OEMs can create scripts to accomplish any number of tasks, limited only by the programmer's imagination. There are four types of scripts in Mach4: M codes, modules, screen, and panels. M codes, screen scripts and panel scripts are each separate containers for code. They cannot interact with each other directly. For example, a variable or function defined in an M code cannot be used in a script on the screen. However, modules provide a place to put code that can be accessed by the others. An M code or screen script can call and use functions and variables from a loaded module. Registers are a powerful way to pass data between different processes in Mach4 and can be used as a bridge between script types.

Script Editor

Mach4 includes a built in script editor. The editor can be found in the 'Operator' menu as 'Edit/Debug Scripts' (see Figure 2-1). Selecting 'Edit/Debut Scripts' will open a window to select the script to be edited. By default the 'Macros' folder for the current profile will be shown. Select and open a file and it will be opened in the editor.

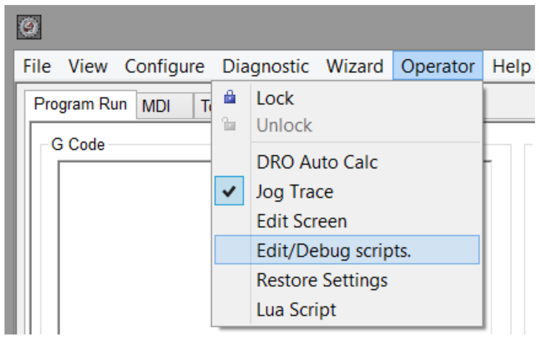
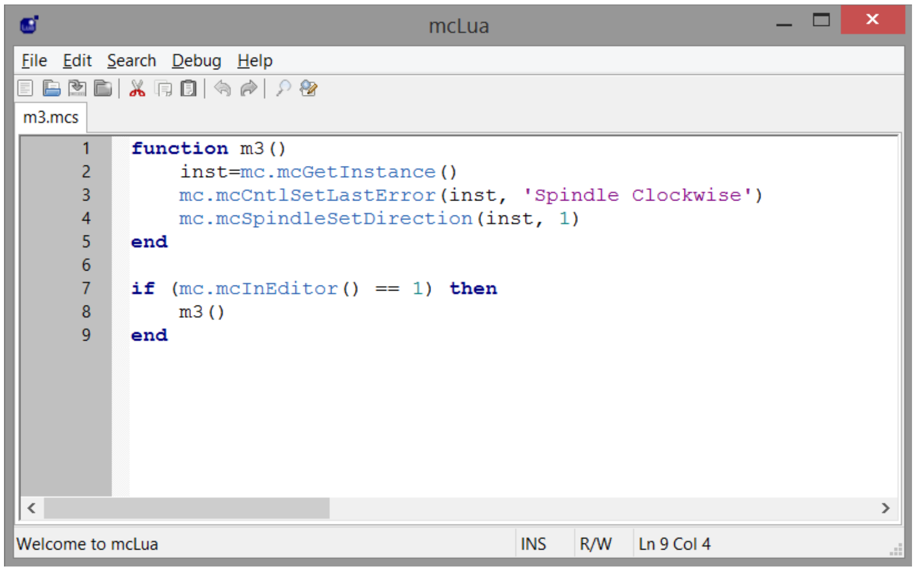


Figure 21: Operator Menu

The script editor is essentially a fancy text editor, with some features specific to programming scripts and macros for Mach4.



```
function m3()
    inst=mc.mcGetInstance()
    mc.mcCntlSetLastError(inst, 'Spindle Clockwise')
    mc.mcSpindleSetDirection(inst, 1)
end

if (mc.mcInEditor() == 1) then
    m3()
end
```

Figure 22: Script Editor

File Menu

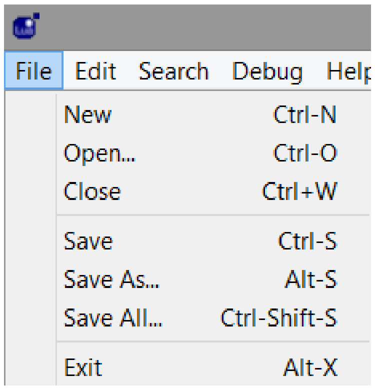


Figure 23: Script editor file menu.

The file menu contains the controls “New”, “Open”, “Close”, “Save” and “Exit”. Selecting new will open a blank file for creating a new script. Open will open a window to allow the user to find and select an existing script to edit. The close option will close the currently active file. Save allows the user to save the script, there is also a “Save As” option that will can save the document as a new name or in a new location. Exit simply closes the editor and any open files.

Edit Menu

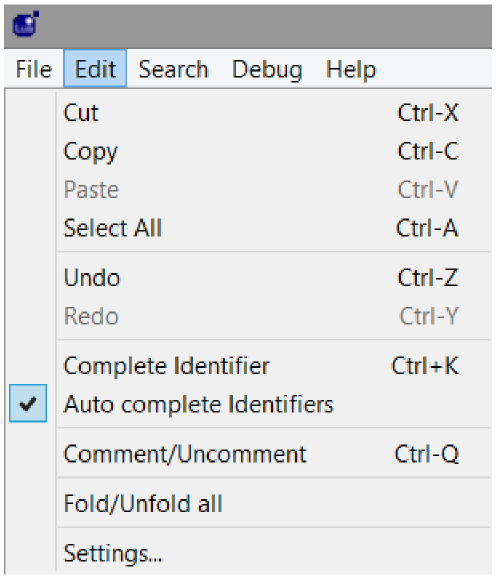
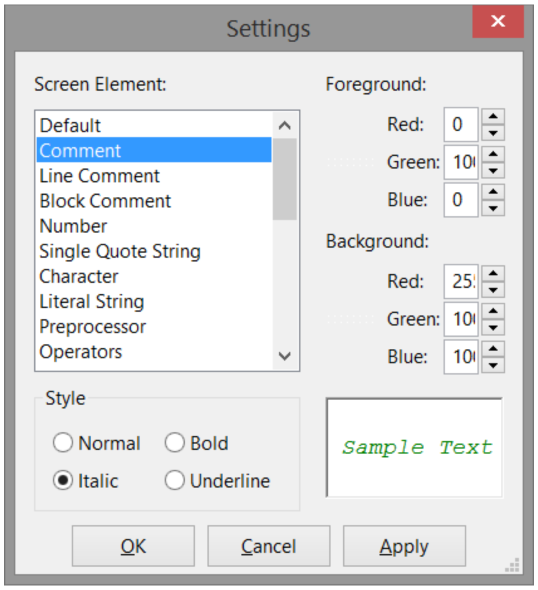


Figure 24: script editor edit menu.

The edit menu contains the typical Cut, Copy, Paste, Select All, Undo/Redo controls as well as a couple specific to the script editor. When Auto Complete Identifiers is checked the auto complete window will display when typing text in the editor. If the text being typed partially matches known commands then the window will display the possible options. When Auto Complete Identifiers is not checked the auto complete window will not be displayed automatically, but it can be manually shown with the Complete Identifier (Ctrl+K) command. Comment/Uncomment will either add the “- -“ characters to selected text to change it to a comment, or remove the “- -“.

Settings displays window that allows the operator to change the color and style of different types of text in the program.


Settings

Screen Element:

Default
Comment
Line Comment
Block Comment
Number
Single Quote String
Character
Literal String
Preprocessor
Operators

Foreground:

Red: 0
Green: 10(
Blue: 0

Background:

Red: 25:
Green: 10(
Blue: 10(

Style

Normal [✓ Italic] Bold Underline

Sample Text

Figure 25: Script editor settings.

Search Menu

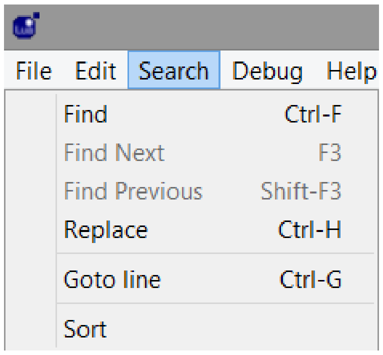


Figure 26: Script editor search menu

In the search menu the user can find commands to search for and replace text, jump to specific line numbers and sort lines in alphanumerical order. Numbers first, 0 to 9, then letters, A to Z.

Debug Menu

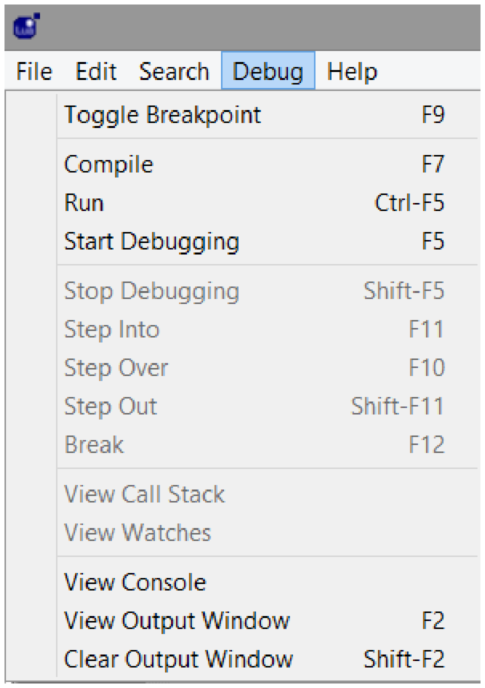


Figure 27: Script editor debug menu.

The debug menu contains the controls for running and debugging programs. Compile compiles the current program into a .mcc file. Run will execute the script.

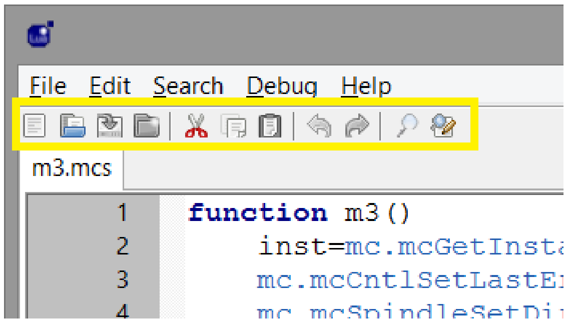
Start Debugging will start the debugger. Debugging helps in diagnosing errors in the program. There are a couple ways to run through the program in the debugging mode. The Start Debugging command will change to Continue when in the debugging mode. When the debugger starts the program will pause at the beginning and wait for input from the user. Continue will run through the complete program. The other controls, Step Into, Step Over, and Step Out allow the user to step through the script line by line. Step Into and Step Over differ in the way functions are executed. Step Into will make a single step into the function. Step Over will execute the entire function and pause at the end. If a function is stepped into, the Step Out command will execute the remaining portion of the function and pause at the end.

The Console shows error messages and debugging data while running programs.

Help Menu

The Help Menu displays the About information.

Toolbar



```
function m3()
    inst=mc.mcGetInsta
    mc.mcCntlSetLastE
    mc.mcSpindleSetDi
```

Figure 28: Script editor toolbar

The toolbar in the script editor contains some of the most common controls. From left to right:

- Create new document
- Open existing document
- Save current document
- Save all open documents
- Cut
- Copy
- Paste
- Undo
- Redo
- Search for text
- Find and replace text

Types of Scripts

Scripts are divided into 4 types: M codes, screen, panel and modules. This chapter will discuss the differences in and interactions between these types.

M codes

M codes, or miscellaneous functions, are used to create additional functionality in a machine. They are specified in a G code program or in the MDI mode. The functions can range from turning on and off coolant to changing tools, to custom code to engrave a serial number. With the ability to script custom macro M codes the possible functions are as diverse and varied as the machines and operators running them. M code scripts use a .mcs file extension.

Scriptable M codes

Scripts cannot be written for all M codes, some have functions that are defined in Mach4 and will only perform that function. Below is a list of the M codes and how they interact with user scripts and internal functions.

User Scriptable/No Internal Function	User Scriptable And Internal Function	Internal Function Only
M6	M3 to M5	M00 to M02
M10 to M45	M7-M9	M46 to M48
M50 to M61	M30	M62 to M65
M66 to M95	M47	M96 to M99
M100 and up		

M codes in the column “User Scriptable/No Internal Function” are completely open to user scripts. There is not function associated to them in Mach4.

M codes in the column “User Scriptable And Internal Function” have internal functions in Mach4, but also allow user scripts. These codes are further divided into those that call the function internal to Mach4 OR a user script and those that run both. M3 to M5 and M7 to M9 are codes that control the spindle and coolant functions. If there is no user script for these codes they turn on/off their respective signals as defined inside Mach4. However, if a user script is present, the script will be run instead. This gives the user the power to create custom codes for custom spindle and coolant applications, but if the machine simply needs to turn on/off an output, no programming is required.

M30 and M47 are both codes that show up at the end of a program. As they are required to end and/or rewind the G code execution their internal functions cannot be ignored. However, it is useful to have a script execute at the end of a program, a parts counter for instance. For this reason both codes will execute a user script if it is present. After executing the script the M30 and M47 will execute the internal function of ending/rewinding the program.

The last column is “Internal Function Only.” M codes found in this column will only execute their internal functions and will NOT execute a user script, even if one is present.

M Code Macro Folder

Using custom M codes requires the scripts to be located in the ‘Macros’ folder located in the folder for the desired profile. Navigate to the Mach4 root directory, usually located on the C drive, open the ‘Profiles’ folder, then open the folder of the desired profile. The name of the folder will be the name of the profile, ‘Mach4Mill’ for example. The ‘Macros’ folder will be located in this profile folder.

Every script file in this folder will be compiled into one file. If a custom M code is desired it must have an associated file in this folder named in the format M3.mcs, replace the 3 with whatever M code is desired. The format of the script is important as well. Because all the files get compiled into one, each M code must be its own function.

```
function m3()
    inst=mc.mcGetInstance()
    mc.mcCntlSetLastError(inst, 'Spindle Clockwise')
    mc.mcSpindleSetDirection(inst, 1)
end

if (mc.mcInEditor() == 1) then
    m3()
```

```
end
```

Above is an example of a custom script for the M code M3. The name of this file is m3.mcs (Note the lower case file name). Reading through the script the main chunk is the function m3() (Note the lower case function name). This is the function that will be called when an M3 is commanded in a G code file or MDI. The second part of this script is for debugging purposes. When the script is open in the editor nothing would happen when it was run unless there was some code to call the m3() function. However, if there was simply and m3() line to call it, the M code would be executed as soon as Mach4 loads. The if statement checks to see if the script is open in the editor, if it is, then the m3() will be executed. Otherwise the function will need to be called from a G code program or MDI command.

Again, it is important to note that the M code script file names needs to be lower case as well as the functions names inside the modules. Mixing in upper case letters will not work.

Screen Scripts

The screen contains scripts than run on load and unload and a plc script. Certain screen elements (such as buttons, panels, DROs, and tabs) can also execute user defined scripts. These scripts call all be set in the screen editor (see the Mach4 customization manual for more information of customizing the screen).

Screen Load Script

The screen load script runs when the screen is loaded. This is a useful tool for loading saved settings or data, setting a start-up state, initializing controls, etc.

Global functions that will be used in other scripts on the screen can also be run in the screen load script. All scripts in the screen (with the exception of panels) will have access to global functions and variables that are defined in the screen load script. This can reduce the amount of programming for the user.

A word of caution: The screen load script runs while the screen is being loaded. When trying to set the state of screen elements use care, sometimes the target element has not been loaded when the screen load script runs. If data it to be set on the screen, it is usually best to do that in the first run of the PLC script.

Screen Unload Script

The screen unload script runs when the screen is unloaded. This can be a useful tool for saving settings or data. Registers are a perfect example as their values are not saved by Mach4 on exit. The best place to store data with a profile is in the .ini file. Mach4 provides an easy way to do this from a script with the mc.mcProfileWriteString(inst, section, key, value) command. With this command it is possible to write any data to the .ini file to be saved and reloaded later. The register's section shows specific examples for saving and loading registers from the .ini file.

PLC Script

The PLC script continuously runs at an interval set in the screen. By default the PLC scripts run on a 50 millisecond interval. Although this is a script and not a ladder type program, it does provide a similar functionality to a PLC, hence the name. This script can monitor the state of signals and inputs and outputs and react very quickly. A common use for the PLC script is showing errors or faults from external devices such as servo drives and VFDs.

The first run of the PLC script is also the best place to set data on the screen for the first time. This ensures that the target element on the screen exists before data is being set. Running a section of the code on the first run is easy to accomplish with a simple counter in the PLC script.

```
count = count + 1
```

With this count variable at the top of the PLC program it will count up by one every time the PLC script runs. So, on the first run count will equal 1. So an if statement can run certain code only on the first run.

Signal Script

The signal script is an all new concept in Mach4. This script is an event handler that can be used to perform actions in response to state changes of signals in the signal library. Signals are internal triggers for events in Mach4 and are not to be confused with external inputs and outputs. Some signals are completely internal and some are used to connect to external inputs and outputs. The signal script can connect all signals to an action.

How does it work? A change in state of a signal is considered an event, on every event the signal script runs. In the signal script two variables are used to determine which signal triggered the event and what its new state is. The variable “sig” is the internal ID number of the signal, and the variable “state” is the state of the signal after the event.

Now we know which signal triggered the event and what its state is, but we don’t know if it is the signal we want. To do this we need a way of comparing the ID numbers of the signal we want to perform an action and the signal that triggered the event. This means knowing the ID number of the signal we want. Mach4 makes this easy on us by providing a complete set of signal definitions. They are all in the format: mc.OSIG_MACHINE_ENABLED, mc.ISIG_INPUT0, etc.

One application of this is to connect physical buttons on a control panel to actions in Mach4. Let’s make a simple cycle start button that is setup on input 1. Since we will be using input 1, the ID number we’ll be looking for is mc.ISIG_INPUT1. In the signal script we could have this code:

```
if (sig == mc.ISIG_INPUT1) then
    local inst = mc.mcGetInstance()
    mc.mcCntlCycleStart(inst)
end
```

Now, this would work, however remember that the signal script runs for every event. The events are when the signal changes state. So this code would command the cycle start when the button is pressed, and again when it is released. To avoid this we can look for the state to be what we want as well. That would lead us to:

```
if (sig == mc.ISIG_INPUT1) and (state == 1) then
    local inst = mc.mcGetInstance()
    mc.mcCntlCycleStart(inst)
end
```

```
SignalTable = {
    [mc.ISIG_INPUT1] = function (on_off)
        if (on_off == 1) then
            mc.mcCntlCycleStart(inst)
        end
    end
}
```

This code would look for the input 1 signal to change to an active state. When the signal changes to the off state the cycle start will not run. This is relatively simple, but if when connecting a lot of signals the script can get very complex, and checking many if statements can bog things down, slowing the reaction to state changes. To make things run more efficiently we can use a table for all the signals we want to use, and index that table from the signal script. The best place to create the table is in the screen load script, where it will be loaded when Mach4 is started and can then be accessed by the signal script, or any other script in the screen. The table in the screen load script could look something like this:

This table contains a function with a name matching the ID number of signal connected to input 1. That function will command a cycle start when the state of the signal is equal to 1, or the signal is active. Now that we have a table we need some code to index it in the signal script.

This code, which lives in the signal script, will look into the table, SignalTable, for an entry matching the signal ID number that is stored in the variable sig. If there is no entry nothing is done. If there is, then the state is passed into the function and the desired action is performed.

This seems more complex than the if statements at first glance, but it really isn't. The only code that will be in the signal script is that shown above. The signal table will grow as functions are added in, but it is no more difficult or complex than the many if statements that would be required. And, as said before, the table is much faster to index and thus far more efficient and reactive to events.

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

Panels

Although panels are located on the screen, they deserve their own section as they are a separate entity. Unlike all the other screen elements they do not share the same global space, and thus cannot access functions or variables in the screen load script like buttons and other controls can. However, they can load and utilize modules just as any other script in Mach4 can.

Panels are simply and environment to run a self-contained Lua program. Elements of a panel are not defined in the screen designer, they are defined in the code contained in the panel. The easiest way to create an interface in a panel is to use a form designer capable of outputting Lua code, wxFormBuilder for example.

The mouse wheel as MPG code is a great example of how panels can be used. A video tutorial about this code can be found here: https://www.youtube.com/watch?v=MRyaRQwhYWk. A link to the code is in the description.

Wizards are another example of what can be shown in a panel. Mach4 comes with an example bolt hole circle wizard that can be displayed in its own dedicated frame or in a panel on the Mach4 screen. The code for this wizard can be found in the "Wizards" folder in the Mach4 directory on your computer, the file name is "BoltHoleCircle.mcs".

If you try to display and scale an image in a panel on the screen you may find that the images only scales to the correct size after resizing the screen. If you run into this issue the way to fix it is the get the parent of the panel, and catch the OnIdle event from it. In the event, resize the images and then use a flag to only run your code in the OnIdle event once. This will prevent you're code from creating an unneeded update loop of resizing the images inside the Idle event.

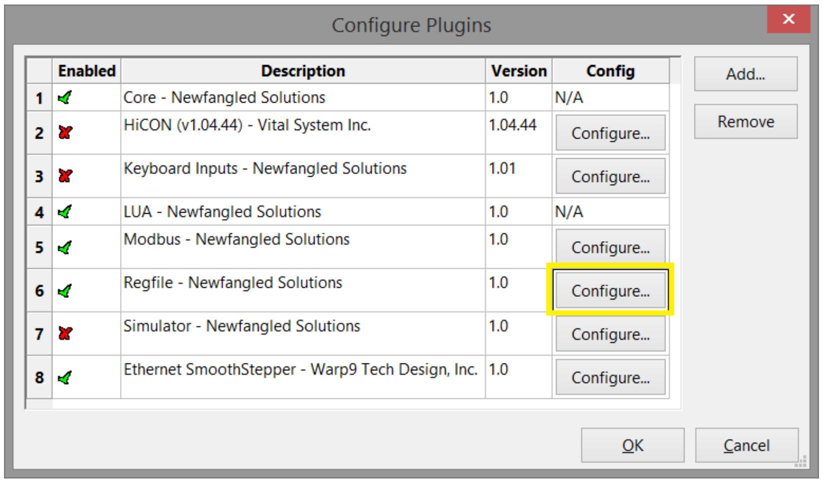
```
mcParent = mcLuaPanelParent:GetParent()
FirstTime = true
mcParent:Connect( wx.wxEVT_IDLE, function(event)
    if FirstTime == true then
        FirstTime = false
        --Run Resizing code
    end
    event:Skip()
end)
```

Registers

Registers are a very powerful tool in Mach4. They are completely user definable and can be accessed from anywhere in Mach4. Scripts can use them to record and save data or to transfer data to another script or communicate with a plugin. Registers can contain numbers or strings.

Creating Registers

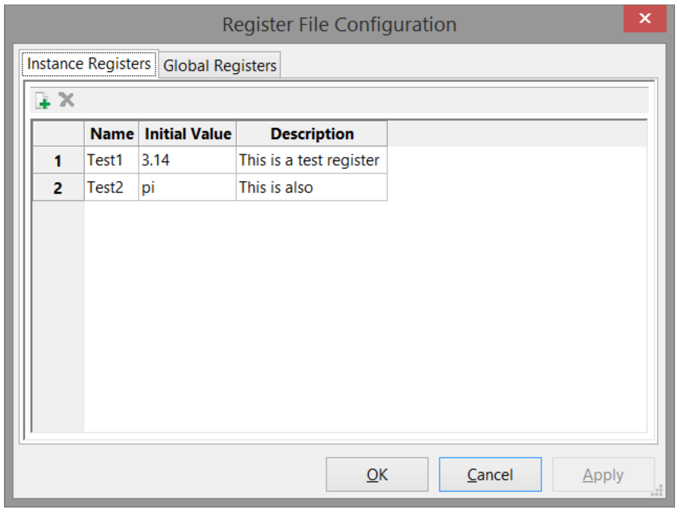
Registers are created in the Regfile plugin. Select "Plugins…" from the "Configure" menu to display the plugin configuration window, figure 4-1.



	Enabled	Description	Version	Config
1	✓	Core - Newfangled Solutions	1.0	N/A
2	✗	HiCON (v1.04.44) - Vital System Inc.	1.04.44	Configure...
3	✗	Keyboard Inputs - Newfangled Solutions	1.01	Configure...
4	✓	LUA - Newfangled Solutions	1.0	N/A
5	✓	Modbus - Newfangled Solutions	1.0	Configure...
6	✓	Regfile - Newfangled Solutions	1.0	Configure...
7	✗	Simulator - Newfangled Solutions	1.0	Configure...
8	✓	Ethernet SmoothStepper - Warp9 Tech Design, Inc.	1.0	Configure...

Figure 49: Configure plugins window.

In the configure plugins window find the Regfile plugin, row 6 in this case, and click on the "Configure…" button. This will open the register configuration window, figure 4-2.



	Name	Initial Value	Description
1	Test1	3.14	This is a test register
2	Test2	pi	This is also

Figure 410: Register file configuration window.

In this window new registers can be created as well as assigned initial, or default, values. To add a register click on the icon with the green plus sign (top left corner of the tab). A new row will be added, simply give it a name, an initial value, and a description. The name will be used to look up the register, so use something simple. The description is optional but accurate descriptions are certainly beneficial in the long run.

The initial value is the value that will be assigned to the register when Mach4 loads. The registers are not saved by default when Mach4 is closed. If a register, or many, needs to be saved on exit the Screen Unload script is a great place to do this. Also, to load the saved value use the Screen Load script. Saving registers to and loading them from the Machine.ini file will be covered in the next sections.

Viewing Registers

The Regfile configuration window shows the registers and their default values. A diagnostics window is provided to show the registers and their values in real time. The diagnostics window can be found by selecting “Regfile” from the “Diagnostic” menu in Mach4.

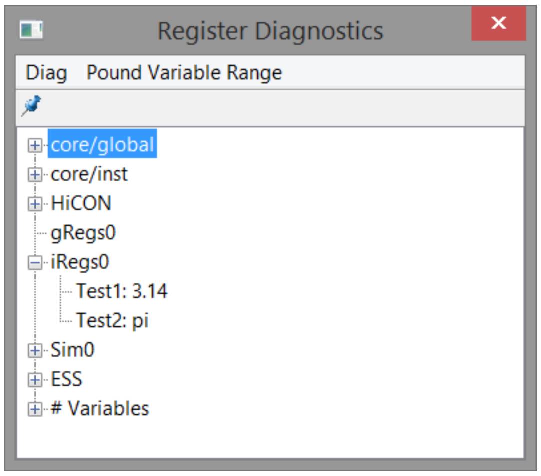


Figure 411: Register diagnostics window.

The diagnostics window displays all the registers in Mach4 and also provides a display for the pound variables. If the “iRegs0” category is expanded we will see the registers and associated values from the previous section.

The register diagnostics window is not limited to only viewing register values, they can be changed as well. Double click on any value and an input window will pop up to allow the user to change the value.

Using Registers in Scripts

There are several commands available for working with registers, for clarity and simplicity we will be using only the necessary few here:

- hreg = mc.mcRegGetHandle(inst, path)
- val = mc.mcRegGetValue(hreg)
- string = mc.mcRegGetValueString(hreg)
- mc.mcRegSetValue(hreg)
- mc.mcRegSetValueString(hreg)

The first step in using registers in scripts is to get the handle. The handle is basically an ID number assigned to the register internally in Mach. There is now what to know this except to ask Mach for it. The mcRegGetHandle function returns the ID or handle. There are two arguments required for this function, the current instance number of Mach we are working in and the path of the register. The instance can be found using the mc.GetInstance function in the format:

```
inst = mc.mcGetInstance()
```

The path is the register type followed by the register name. For example, the path for the first register in figure 4-2 is “iRegs0/Test1.” So to get the handle of the Test1 register the code would look like:

```
inst = mc.mcGetInstance()
hreg = mc.mcRegGetHandle(inst, "iRegs0/Test1")
```

Now to get the value. Here we are discussing two ways to retrieve the value, as a number or as a string. If the register’s value is an unknown type then use mcRegGetValueString. Using mcRegGetValue will result in an error if the register contains a string, but mcRegGetValueString can read a number or a string as a string. The only catch there is that math cannot be performed on a string, even if it is a sting of only numbers. Strings containing only numbers can be converted to a number format by using the tonumber(“string”) command. Let’s use mcRegGetValueString to retrieve the value of Test1 and then convert it to a number for computation later. Ofcourse this only works for values that are numbers, the tonumber(“string”) function will create an error if the string contains more than just numbers.

```
inst = mc.mcGetInstance()
hreg = mc.mcRegGetHandle(inst, "iRegs0/Test1")
val = mc.mcRegGetValueString(hreg)
val = tonumber(val)
```

It would save a line of code to use the mcRegGetValue(hreg) function, but there is a reason for using the string method shown above. Every time a register is called it requires two lines of code, as well as the current instance. There is an easier way, create our own function. Remember that all the contents of the macros folder are rolled into one large file, and the screen has access to all the elements of the screen, and the modules can be loaded and used everywhere. With the standard method shown above this code would have to be used every time register information was needed, in every M code, every button, every script in Mach4. Making a function to call the register is much easier and can make the code a lot cleaner. Let's use a button on the screen as an example. We would like a button to get the value of a register and display it in a message box to the user. Or first step will to be to create a function in the screen load script to get and return the value of the register. This function can be used by our new button and any other button we decide to add to the screen. In the screen load script we will make a function as shown below.

```
function GetRegister(regname)
    local inst = mc.mcGetInstance()
    local hreg = mc.mcRegGetHandle(inst, string.format("iRegs0/%s", regname))
    return mc.mcRegGetValueString(hreg)
end
```

With this function in the screen load script, our button can call it. To get the value of the "Test1" register and display it in a message box the button script will be:

```
regval = GetRegister("Test1")
wx.wxMessageBox(regval)
```

Now with that function created, any time we want to retrieve the value of a register we simple call the GetRegister(regname) function with the name of the register to get. In the screen of course. Try adding this function to the screen load script then create a button on the screen to use the GetRegister function.

A similar process can be used in the macros. For example a file can be created named "macrofunctions" that contains the GetRegister function and any others that will be frequently used. The functions in this file can be accessed by any of the other macros.

Another approach is to use a module. The module can contain these functions then be loaded by the screen load script or in a macro. The functions contained in the module can be used anywhere it is loaded.

A function for writing to registers is also useful, and if very similar to the GetRegister function above.

```
function WriteRegister(regname, regvalue)
    local inst = mc.mcGetInstance()
    local hreg = mc.mcRegGetHandle(inst, string.format("iRegs0/%s", regname))
    mc.mcRegSetValueString(hreg, tostring(regvalue))
end
```

Using WriteRegister is also very simple.

```
WriteRegister("Test1", 12)
```

The example will write a value of 12 to the instance register "Test1". Again, this function does everything as a string, this way strings or numbers can be sent and set to the registers. If a number is passed into the function it will be converted to a string with the tostring function.

Saving Registers to Machine.ini

Since register values are not saved when Mach is closed it is necessary to create code to save any desired registers. The easiest and most convenient place is in the Machine.ini file. One place for a register save script is in the screen unload script. This script runs when the screen is unloaded/Mach4 is closed.

Another place to save registers is in the code immediately following when they are set. The best time and place to save registers to the .ini file will vary depending on the application.

To save a register to the .ini file use mc.mcProfileWriteString(). To use this we need to define a section in the .ini file to place the data, specify a name for the data and then the value. For this example let's assume the register 'Test1' equals 6.

```
local inst = mc.mcGetInstance()

local hreg = mc.mcRegGetHandle(inst, "iRegs0/Test1")
local val = mc.mcRegGetValue(hreg)
val = tostring(val)

mc.mcProfileWriteString(inst, "Registers", "Test1", val)
```

This little bit of code gets the value of register 'Test1' and then writes it to the Machine.ini file in a section labeled as "Registers" with a name "Test1". The Machine.ini file will contain a section like this:

[Registers]
Test1 = 6

If the section "Registers" does not exist it will be created, if it does the key "Test1" will be added to it. Writing data to the Machine.ini file is very useful for saving registers, but that is not the only application for it. Any data can be saved for use later, custom screens may have configuration data that needs to be stored, wizards may have information to store for next time, etc.

Loading Registers From Machine.ini

After data is saved to the Machine.ini file it is very useful to retrieve it. For registers, this means that they can be initialized with saved values when Mach4 starts.

Like with saving to the Machine.ini file there is a function for loading from it, mc.mcProfileGetString(). For example, to get the previously saved value for "Test1" and set it to the register of the same name the code could be:

```
local inst = mc.mcGetInstance()

local val = mc.mcProfileGetString(inst, "Registers", "Test1", "nf")

local hreg = mc.mcRegGetHandle(inst, "iRegs0/Test1")
mc.mcRegSetValueString(hreg, val)
```

In this code the variable 'val' is set to the value found in the Machine.ini file under the section "Registers" and key "Test1". If there is no value found, 'val' will be set to "nf", the default value we defined in the mc.mcProfileGetString() function. This default value can be defined as anything, it will only be used if there is no value found in the Machine.ini file. A unique value here can be useful to perform a specific action if the key does not exist in the .ini file or if a simple default value is desired.

Examples

The following examples are to provide some guidance on the creation of scripts. They are not specific to any one machine and are not guaranteed to work on any machine.

Using Signals

Signals are internal triggers for events in Mach4. They can be tied to external I/O, or internal events such as running a Gcode file, jogging, feed hold, soft limit state, enabled state, etc. A common use for signals in scripts will be to read the state of inputs, and set outputs active or inactive. Turning on/off coolant, clamping fixtures, and special spindle functions, are all examples of when signals might be used in a script. Here is a quick script for turning the spindle on, in the forward direction, Mach signal OSIG_SPINDLEFWD.

```lua
local inst = mc.mcGetInstance()
local hsig = mc.mcSignalGetHandle(inst, mc.OSIG_SPINDLEFWD)
local spinstate = mc.mcSignalGetState(hsig)
if (spindstate == 0) then
    mc.mcSignalSetState(hsig, 1)
else
    mc.mcSignalSetState(hsig, 0)
end
```

To use a signal we first need to know the handle, or Mach's internal reference ID for the signal. Once we know the handle, the state can be read and/or written to. By reading the state of OSIG_SPINDLEFWD we know if the spindle is on in the forward direction or not. We can then decide what to do based on that information. This script will turn the spindle on in the forward direction if it is currently not running in forward, and will stop it if it is running in forward. This is a very simple example, but it shows the basics of how to use signals.

Reading Data from an External File

Lua scripts are also capable of reading data from an external file, a .csv for example. This can be extremely useful for providing a table of shapes for a wizard, or materials, positions for a tool changer, etc. Having data in an external file can make it easier to edit by providing a formatted Excel spreadsheet for example, or make it more difficult by making the file read only.

For an example let's make a .csv file that contains X, Y and Z positions of tools in a rack style tool changer, we'll call it "ToolChangePositions.csv". This is what it will look like:

Looking at this table we can see that tool number 1 is at the position X3.5, Y2, Z-10. If we can read this data into a table in a script it can be used to find the tool position in a tool change routine. Below is what that script could look like.

For this a table makes the most sense for storing the tool position data. We can start by defining the table TC_Positions.

Then we can find the .csv file and store the path to a variable, in this case CSVPath.

Next we open the file to read the data. Because we are going to have multiple pieces of data, tool number and three positions, for each entry we will create a table for each entry within the TCPositions table. The variable ToolNum sets the ID of each entry, or line in the .csv. So we start with ToolNum = 0, the header information will land here. Then we increment ToolNum by 1 and run it again, so all the data from the tool number 1 line in the .csv will be in the table TCPositions[1] table and so on. This loop will run until there is no more data in the .csv.

Just before incrementing ToolNum, we can set a max tool number in TCPositions so we know how many tools have been defined. We will call this TCPositions["Max"]. The value of this will be equal to the last tool number entered. So in this example TCPositions["Max"] = 6.

Now the position data can be used in a tool change script. The following lines show how to read and use the data. This example script checks to make sure the selected tool number is greater than 0 but less than or equal to the maximum. So if a user tried to select tool number 10 they would get the "ERROR: Tool number out of range!" message. If the selected tool number is within the valid range then the tool position data will be displayed in the message bar on the Mach4 screen.

```
local TC_Positions = {}
local inst = mc.mcGetInstance()

local CSVPath = wx.wxGetCwd() .. "\\Profiles\\YourProfile\\Modules\\ToolChangePositions.csv"

ToolNum = 0;
--[[
Open the file and read out the data
--]]
 io.input(io.open(CSVPath,"r"))
local line;
for line in io.lines(CSVPath) do
    tkz = wx.wxStringTokenizer(line, ",");
    TC_Positions[ToolNum] = {}-- make a blank table in the positions table to hold the tool data
    local token = tkz:GetNextToken();

    TC_Positions[ToolNum] ["Tool_Number"] = token;
    TC_Positions[ToolNum] ["X_Position"] = tkz:GetNextToken();
    TC_Positions[ToolNum] ["Y_Position"] = tkz:GetNextToken();
    TC_Positions[ToolNum] ["Z_Position"] = tkz:GetNextToken();
        TC_Positions["Max"] = ToolNum --Set the max tool number
    ToolNum = ToolNum + 1 --Increment the tool number
end
    io.close()

--Read tool data
local SelectedToolNum = 1
local MaxToolNum = TC_Positions["Max"]

if (SelectedToolNum <= MaxToolNum) and (SelectedToolNum > 0) then
    local Num = TC_Positions[SelectedToolNum].Tool_Number
    local XPos = TC_Positions[SelectedToolNum].X_Position
    local YPos = TC_Positions[SelectedToolNum].Y_Position
    local ZPos = TC_Positions[SelectedToolNum].Z_Position

    mc.mcCntlSetLastError(inst, string.format("Tool: %.0f | X: %.3f | Y: %.3f | Z: %.3f", Num,
XPos, YPos, ZPos))
else
    mc.mcCntlSetLastError(inst, "ERROR: Tool number out of range!")
end
```

Modules

Modules are scripts that can be accessed from any other script in Mach4. This is useful if you have functions or data that could be used in multiple types of scripts all throughout your Mach interface. This could be reading and storing data for wizards, commonly used functions, or all of the special scripts and functions used in the screen. For developers putting special or custom scripts in a module has the benefit of being able to compile it so pieces of the code cannot be copied and reused somewhere. Screen buttons that require lengthy scripts can be simple function calls to code in a compiled module.

For an example, the tool change position script from the previous section could be converted into a module to be used by the tool change routine. This will simplify the tool change script itself and also allow access to the data in other scripts.

The first step will be making some slight changes to the tool change positions script from the previous example. It will be saved in the Modules folder as ToolChangePositions.lua.

```
local TC_Positions = {}
local inst = mc.mcGetInstance()

local CSVPath = wx.wxGetCwd() .. "\\Profiles\\YourProfile\\Modules\\ToolChangePositions.csv"

ToolNum = 0;
--[[
Open the file and read out the data
--]]
 io.input(io.open(CSVPath,"r"))
local line;
for line in io.lines(CSVPath) do
    tkz = wx.wxStringTokenizer(line, ",");
    TC_Positions[ToolNum] = {}-- make a blank table in the positions table to hold the tool data
    local token = tkz:GetNextToken();
    TC_Positions[ToolNum] ["Tool_Number"] = token;
    TC_Positions[ToolNum] ["X_Position"] = tkz:GetNextToken();
    TC_Positions[ToolNum] ["Y_Position"] = tkz:GetNextToken();
    TC_Positions[ToolNum] ["Z_Position"] = tkz:GetNextToken();
        TC_Positions["Max"] = ToolNum
    ToolNum = ToolNum + 1;
end
    io.close()
function TC_Positions.GetToolData(SelectedToolNum)
    local MaxToolNum = TC_Positions["Max"]
    if (SelectedToolNum <= MaxToolNum) and (SelectedToolNum > 0) then
        return TC_Positions[SelectedToolNum]
    else
        return nil
    end
end
return TC_Positions
```

Reading through this new script we can see that it is mostly the same. The difference is an addition of a function, TC_Positions.GetToolData(SelectedToolNum), to return the data associated with the desired tool number back to the main program. This module can stay in the modules folder, and could also be compiled and saved as a .mcc file instead of the .lua or .mcs formats. Any of the three formats are acceptable. A compiled .mcc file has the benefit and drawback of not being editable or even viewable.

The function TC_Positions.GetToolData(SelectedToolNum) checks the selected tool number against the max tool number and zero. If the selected tool number is in the valid range then the data is returned using the return command: *return TC_Positions[SelectedToolNum]*. If the selected tool number is outside the valid range then "nil" is returned. In the main program we can use this difference in returned value to check if the tool number was valid or not.

The last line in the module is *return TC_Positions*. This line sends the table TC_Positions, with all of its contents, back to the script that loaded the module.

Now for the main script, this is what would appear in the tool change script to get the tool change positions.

```
local inst = mc.mcGetInstance()
package.path = wx.wxGetCwd() .. "\\Profiles\\YourProfile\\Modules\\?.lua;"
if(package.loaded.ToolChangePositions == nil) then
    tcp = require "ToolChangePositions"
end

local SelectedTool = mc.mcToolGetSelected(inst)

ToolData = tcp.GetToolData(SelectedTool)

if (ToolData ~= nil) then
    Num = ToolData.Tool_Number
    XPos = ToolData.X_Position
    YPos = ToolData.Y_Position
    ZPos = ToolData.Z_Position
    mc.mcCntlSetLastError(inst, string.format("Tool: %.0f | X: %.3f | Y: %.3f | Z: %.3f", Num,
XPos, YPos, ZPos))
else
    mc.mcCntlSetLastError(inst, "ERROR: Tool number out of range!")
end
```

The first part of this script set the file path for the module, or package, to load. The "?.lua" is like a wild car. It's looking for any file with the extension .lua, if you're module is compiled this would need to be changed to .mcc. The following if statement checks to see if the desired module, in this case ToolChangePositions, is loaded to the variable "tcp". Whatever is returned at the end of the module, in our module the table "TC_Positions" is returned on the last line, is now contained in the variable tcp.

Now tool data can be retrieved by using the TC_Positions.GetToolData(SelectedTool) function. To call it from the main script replace the variable name in the module with the new variable name that the module was loaded to in the main script, tcp. So to get tool data for a selected tool we would call the function tcp.GetToolData(SelctedTool). In this example script the command mc.mcToolGetSelected(inst) is used to get the currently selected tool from Mach, the last T number commanded in a program or in MDI. So in the case of a tool change; if T4 M6 is commanded in a program then calling mc.mcToolGetSelected(inst) in the M6 macro will return a value of 4. Passing this into our GetToolData function will return the position values for the selected tool provided that it is within the acceptable range.

Now we can check to see if we got valid data back and use it if we did. If the function returns nil, then we know that the tool number was outside the acceptable range and we can error and/or abort the process.

Tool Change

In the previous examples we've been building up to a tool change macro for a rack style tool changer. It's time to put it all together. We will use the module for getting the tool position data, and the main script from the last example will be the bones of our tool change script.

A couple thoughts before we write the script. 1.) A tool change routine should only execute if the tool actually needs changing, there is no sense doing anything if the tool we want to change to is already in the spindle. 2.) We will be moving the machine around and changing modal states and feedrates. It is a good idea to store the state of the machine prior to changing anything so it can be returned to that state when the tool change is complete. This helps to avoid accidents from a machine unknowingly being changed into incremental mode when it is expected to be in absolute, bad feedrates being used because they weren't specified after a tool change, or any of a host of other possible issues. Best to avoid such problems.

Let's create an m6 tool change script to use our module and change tools.

```
local inst = mc.mcGetInstance()
package.path = wx.wxGetCwd() .. "\\Profiles\\YourProfile\\Modules\\?.lua;"
if(package.loaded.ToolChangePositions == nil) then
    tcp = require "ToolChangePositions"
end

function m6()
    ------ Get and compare next and current tools ------
    local SelectedTool = mc.mcToolGetSelected(inst)
    local CurrentTool = mc.mcToolGetCurrent(inst)
    if (SelectedTool == CurrentTool) then
        mc.mcCntlSetLastError(inst, "Next tool = Current tool")
        do return end
    end
    ------ Get current state ------
    local CurFeed = mc.mcCntlGetPoundVar(inst, 2134)
    local CurFeedMode = mc.mcCntlGetPoundVar(inst, 4001)
    local CurAbsMode = mc.mcCntlGetPoundVar(inst, 4003)

    ------ Get position data for current tool ------
    ToolData = tcp.GetToolData(CurrentTool)
    if (ToolData ~= nil) then
        Num1 = ToolData.Tool_Number
        XPos1 = ToolData.X_Position
        YPos1 = ToolData.Y_Position
        ZPos1 = ToolData.Z_Position
    else
        mc.mcCntlEStop(inst)
         mc.mcCntlSetLastError(inst, "ERROR: Tool number out of range!")
        do return end
    end

    ------ Get position data for next tool ------
    ToolData = tcp.GetToolData(SelectedTool)
    if (ToolData ~= nil) then
        Num2 = ToolData.Tool_Number
        XPos2 = ToolData.X_Position
        YPos2 = ToolData.Y_Position
        ZPos2 = ToolData.Z_Position
    else
        mc.mcCntlEStop(inst)
         mc.mcCntlSetLastError(inst, "ERROR: Tool number out of range!")
        do return end
    end

    ------ Move to current tool change position ------
    local GCode = ""
    GCode = GCode .. "G00 G90 G53 Z0.0\n"
    GCode = GCode .. string.format("G00 G90 G53 X%.4f Y%.4f\n", XPos1, YPos1)
    GCode = GCode .. string.format("G00 G90 G53 Z%.4f\n", ZPos1 + 1.0)
    GCode = GCode .. string.format("G01 G90 G53 Z%.4f F15.0\n", ZPos1)
    mc.mcCntlGcodeExecuteWait(inst, GCode)

    ------ Release drawbar ------
    local DrawBarOut = mc.OSIG_OUTPUT4
    local hsig = mc.mcSignalGetHandle(inst, DrawBarOut)
    mc.mcSignalSetState(hsig, 1)

    ------ Move to next tool change position ------
    GCode = ""
    GCode = GCode .. string.format("G01 G90 G53 Z%.4f\n F15.0", ZPos1 + 1.0)
    GCode = GCode .. "G00 G90 G53 Z0.0\n"
    GCode = GCode .. string.format("G00 G90 G53 X%.4f Y%.4f\n", XPos2, YPos2)
    GCode = GCode .. string.format("G00 G90 G53 Z%.4f\n", ZPos2 + 1.0)
    GCode = GCode .. string.format("G01 G90 G53 Z%.4f F15.0\n", ZPos2)
    mc.mcCntlGcodeExecuteWait(inst, GCode)

    ------ Clamp drawbar ------
    mc.mcSignalSetState(hsig, 0)

    ------ Move Z to home position ------
    mc.mcCntlGcodeExecuteWait(inst, "G00 G90 G53 Z0.0\n")

    ------ Reset state ------
    mc.mcCntlSetPoundVar(inst, 2134, CurFeed)
    mc.mcCntlSetPoundVar(inst, 4001, CurFeedMode)
    mc.mcCntlSetPoundVar(inst, 4003, CurAbsMode)

    ------ Set new tool ------
    mc.mcToolSetCurrent(inst, SelectedTool)
    mc.mcCntlSetLastError(inst, string.format("Tool change - Tool: %.0f", SelectedTool))

end

if (mc.mcInEditor() == 1) then
    m6()
end
```

Each section of the tool change macro is labeled to make it easier to follow. We've combined all the code examples from the previous examples to arrive here, with a couple new additions. Let's walk through it.

The first section of the code loads the module we will be using to look up the tool positions. This is positioned outside the m6() function so that it can be available to all M codes without having to call it in every macro it is used in.

Next is the m6() function. Recall that an M code requires a function of the same name to call. In this case an M6 in the program will call the m6() function.

Now, inside the function is the meat of the tool change. The first section, "Get and compare next and current tools," gets the next tool and current tool and compares them to make sure a tool change should happen. If the current tool in the spindle is the tool to be changed to exit the script.

Previously we talked about getting the current state of the machine prior to changing anything so we can set it back the way we found it when the script exits. The next section, "Get current state," retrieves that data for us and saves it for later.

The next two sections, "Get position data…," does just what it says. This is the section of the script that accesses the "ToolChangePositions" module that we created. These two sections function just like the code we developed in the module example. After getting the tool position data it is checked to make sure a valid selection was made. This error also needs to stop the machine or it will just keep running the program when the script is exited, this is big potential for a crash. In this example and E-stop command is given when an invalid tool is selected to make sure the machine is incapable of continuing when the script is exited. This way if the selection is invalid we exit the script with an error rather than get part way thru a tool change and then error for bad data.

If the tool selection is valid for both tools we move ahead to the next sections which perform the actual tool change motion. The "Move to current tool change position," executes a short G code program to move the machine to the spot in the rack for the tool currently in the spindle.

The "Release drawbar" section does just that. Using the code from the example in section 5.1 we can make a bit of code to activate the output that controls the drawbar.

After releasing the tool the sections, "Move to next tool change position", "Clamp drawbar", and "Move to Z home position" continue the rest of the machine motion to pick up the next tool and return home.

After all the motion of actually changing tools, we can "Reset state" back to what it was before entering the macro. The last thing to do is reset the current tool to the new tool in the "Set new tool" section. That ends the m6() function.

Below that is an if statement to check if the script is open in an editor. As discussed before this is for debugging, so when in the editor the function will be called.

Automatic Tool Height Setting

There are several ways to create an automatic tool height setting process. It could be written in G code or in a Lua script. Since this is a Lua scripting manual we'll show one way to set tools using a Lua script. This example puts the auto tool height setting script in an M code for easy access from anywhere in Mach. It can be executed from a button or in a program. The following program is an example of an M code for automatically setting the height of a tool. There are a number of necessary parameters that are stored in registers, so it is necessary to get those values before performing the operation. This script follows the same format as the tool change macro example in section 5.4.

A brief run through: As before the first step in the script is getting and defining necessary variables and the current state of the machine. Register values are retrieved using the GetRegister() function created in section 4.3. It is not defined in this script so it must be defined in an M code header file or a loaded module to allow access to it.

The next sections calculate the position on the touch off sensor and use a G31 probing move the tool into the sensor. The state of the sensor's signal is checked before movement to make sure it is not already active, and the checked again after it is touched to make sure that it was indeed contacted. This is all error checking and although it is not required it is strongly suggested.

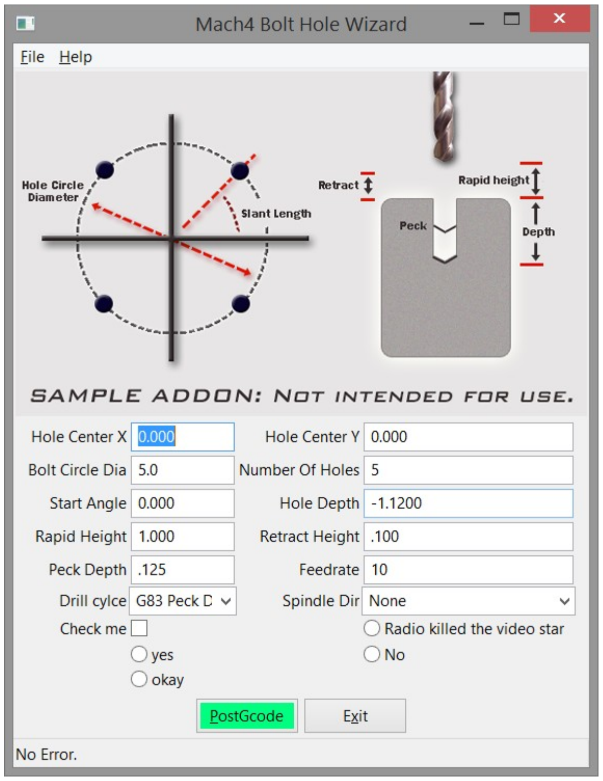
Following the actually tool touch off movement, the offset length is calculated and set and then everything returned to its previous state.

```lua
-----------------------------------------------------------------------------
-- Auto Tool Setting Macro
-----------------------------------------------------------------------------
--[[
    Requires the following instance registers to be defined
    TS_XPos-----------X position of probe (machine position)
    TS_YPos-----------Y position of probe (machine position)
    TS_Type-----------Offset type (1 or 2)
    TS_TouchPos-------Z position of touch off surface (machine position)
    TS_ProbeH---------Height of probe above touch off surface
    TS_DefaultL-------Default tool length guess
    TS_Retract--------Retract distance after probe touch

    Offset Type 1-----Length of tool from gauge line to tip
    Offset Type 2-----Distance from tip of tool to the touch position
]]
--The function GetRegister() must be defined for use by macros
function m1005()
    local inst = mc.mcGetInstance()

    ------------- Define Vars -------------
    local ProbeSignal = mc.ISIG_DIGITIZE

    ------------- Get current state -------------
    local CurTool = mc.mcToolGetCurrent(inst)
    local CurHNum = mc.mcCntlGetPoundVar(inst, 2032)
    local CurFeed = mc.mcCntlGetPoundVar(inst, 2134)
    local CurZOffset = mc.mcCntlGetPoundVar(inst, 4102)
    local CurFeedMode = mc.mcCntlGetPoundVar(inst, 4001)
    local CurAbsMode = mc.mcCntlGetPoundVar(inst, 4003)

    ------------- Get touch off parameters -------------
    local Xpos = GetRegister("TS_XPos", 1)
    local Ypos = GetRegister("TS_YPos", 1)
    local OffsetType = GetRegister("TS_Type", 1)
    local TouchPos = GetRegister("TS_TouchPos", 1)
    local ProbeHeight = GetRegister("TS_ProbeH", 1)
    local RetractDistance = GetRegister("TS_Retract", 1)
    local ToolLengthGuess = GetRegister("TS_DefaultL", 1)

    ------------- Check Probe -------------
    local hsig = mc.mcSignalGetHandle(inst, ProbeSignal)
    local ProbeState = mc.mcSignalGetState(hsig)
    if (ProbeState == true) then
        mc.mcCntlSetLastError(inst, "ERROR: Probe signal is activated")
        do return end
    end

    ------------- Calculations for Gcode -------------
    local StartHeight = TouchPos + ProbeHeight + ToolLengthGuess + .5

    ------------- Generate GCode -------------
    AutoToolSetGCode = ""
    AutoToolSetGCode = AutoToolSetGCode .. "G00 G80 G40 G49 G90\n"
    AutoToolSetGCode = AutoToolSetGCode .. "G00 G53 Z0.0\n"
    AutoToolSetGCode = AutoToolSetGCode .. string.format("G00 G53 X%.4f Y%.4f\n", Xpos, Ypos)
    AutoToolSetGCode = AutoToolSetGCode .. string.format("G00 G53 Z%.4f\n", StartHeight)
    AutoToolSetGCode = AutoToolSetGCode .. "G91 G31 Z-2.0 F25.\n"

    mc.mcCntlGcodeExecuteWait(inst, AutoToolSetGCode)

    --Check probe contact
    ProbeState = mc.mcSignalGetState(hsig)
    if (ProbeState ~= 1) then
        mc.mcCntlSetLastError(inst, "ERROR: No contact with probe")
        mc.mcCntlGcodeExecuteWait(inst, "G0 G90 G53 Z0.0\n")
        do return end
    end

    AutoToolSetGCode = ""
    AutoToolSetGCode = AutoToolSetGCode .. string.format("G91 G00 Z%.4f\n", RetractDistance)
    AutoToolSetGCode = AutoToolSetGCode .. "G91 G31 Z-1.0 F10.\n"

    mc.mcCntlGcodeExecuteWait(inst, AutoToolSetGCode)

    --Check probe contact
    ProbeState = mc.mcSignalGetState(hsig)
    if (ProbeState ~= 1) then
        mc.mcCntlSetLastError(inst, "ERROR: No contact with probe")
        mc.mcCntlGcodeExecuteWait(inst, "G0 G90 G53 Z0.0\n")
        do return end
    end

    AutoToolSetGCode = ""
    AutoToolSetGCode = AutoToolSetGCode .. "G90 G00 G53 Z0.0\n"

    mc.mcCntlGcodeExecuteWait(inst, AutoToolSetGCode)

    ------------- Get touch position and set offset -------------
    local ZProbed = mc.mcCntlGetPoundVar(inst, 5063)
    local ZOffset = ZProbed - ProbeHeight + CurZOffset
    if (OffsetType == 1) then
        ZOffset = math.abs(TouchPos - ZOffset)
    end

    mc.mcToolSetData(inst, mc.MTOOL_MILL_HEIGHT, CurTool, ZOffset)
    mc.mcCntlSetLastError(inst, string.format("Auto tool setting complete, Offset = %.4f",
ZOffset))

    ------------- Set previous state -------------
    mc.mcCntlSetPoundVar(inst, 2134, CurFeed)
    mc.mcCntlSetPoundVar(inst, 4001, CurFeedMode)
    mc.mcCntlSetPoundVar(inst, 4003, CurAbsMode)

end

if (mc.mcInEditor() == 1) then
    m1005()
end
```

Wizards

Wizards are tools that can be created and used to perform common function. In Mach4 a wizard is usually a graphical interface for creating G code files, very similar to conversation programming on some machines. Lua gives the programmer the ability to create a wizard for anything; hole patterns, facing, engraving, spirals, loops, play music. Included with Mach4 is a bolt circle wizard designed to show users how wizards can be built. It is written to be easy for a user to modify to create their own wizard.

Wizard scripts should be located in the “Wizards” folder in the Mach4 root directory. The “Load Wizards” button on the default Mach4 screen will load the scripts located here. When the “BoltHoleCircle” wizard is run it will open a window:



Hole Center X	0.000	Hole Center Y	0.000
Bolt Circle Dia	5.0	Number Of Holes	5
Start Angle	0.000	Hole Depth	-1.1200
Rapid Height	1.000	Retract Height	.100
Peck Depth	.125	Feedrate	10
Drill cylce	G83 Peck D	Spindle Dir	None
Check me	☐		○ Radio killed the video star
	○ yes		○ No
	○ okay		

No Error.

Figure 512: Bolt Hole Circle Wizard

This window is created using the functions and variables laid out in the script. There is a lot going on is this script to create the window layout and setup the buttons. Because a lot of the programming for the window itself is out of the scope of this manual, we will stick to discussing how and where to modify this to create another wizard. The only parts to worry about are the functions: Setupinputs(), SaveSettings(), GenGcode(), and variables “m_iniName” and “m_image”. Following is the code to create the bolt circle wizard:

```
-----------------------------------------------------------------------------
-- Name:        BoltHolelua
-- Author:      B Barker
-- Modified by:
-- Created:     08/03/2013
-- Copyright:   (c) 2013 Newfangled Solutions. All rights reserved.
-- Licence:     BSD license
-----------------------------------------------------------------------------

function GetNextID()
    m_id = m_id+1
    return m_id
end
--global var to hold the frame
mainframe = nil
panel = nil
m_id = 0
m_iniName = "CircleHolePat"


ID_GENGCODE_BUT   = GetNextID()
ID_CLOSE_BUTTON   = GetNextID()
m_image = wx.wxGetCwd() .. "\\Wizards\\HolesNew.png"

function Setupinputs()
 --Add all the inputs
    local val
    m_center_x = AddInputControl("Hole Center X",nil)
    m_center_x:SetValue( mc.mcProfileGetString(0 , tostring(m_iniName), "Xcenter", "0.000") )
    m_center_y = AddInputControl("Hole Center Y", nil)
    m_center_y:SetValue( mc.mcProfileGetString(0 , tostring(m_iniName), "Ycenter", "0.000") )
    m_circle_dia = AddInputControl("Bolt Circle Dia", nil)
    m_circle_dia:SetValue( mc.mcProfileGetString(0 , tostring(m_iniName), "Dia", "5.0") )
    m_NumHoles = AddInputControl("Number Of Holes", nil)
    m_NumHoles:SetValue( mc.mcProfileGetString(0 , tostring(m_iniName), "Holes", "5") )
    m_StartAngle = AddInputControl("Start Angle", nil)
    m_StartAngle:SetValue( mc.mcProfileGetString(0 , tostring(m_iniName), "Startangle", "0.000")
)
    m_z_depth = AddInputControl("Hole Depth", nil)
    m_z_depth:SetValue( mc.mcProfileGetString(0 , tostring(m_iniName), "Depth", "-1.1200") )
    m_rapid_height = AddInputControl("Rapid Height", nil)
    m_rapid_height:SetValue( mc.mcProfileGetString(0 , tostring(m_iniName), "RapidHeight",
"1.000") )
    m_retract_height = AddInputControl("Retract Height", nil)
    m_retract_height:SetValue( mc.mcProfileGetString(0 , tostring(m_iniName), "RetractHeight",
".100") )
    m_peck_depth = AddInputControl("Peck Depth", nil)
    m_peck_depth:SetValue( mc.mcProfileGetString(0 , tostring(m_iniName), "PeckDepth", ".125") )
    m_feedrate = AddInputControl("Feedrate", nil)
    m_feedrate:SetValue( mc.mcProfileGetString(0 , tostring(m_iniName), "Feed", "10") )
    m_cycletype, ID_CYCLE_TYPE = AddSelectControl("Drill cylce", {"G81 Single pass", "G83 Peck
Drill", "G73 High speed Peck"}, ID_CYCLE_TYPE)
    local val =  mc.mcProfileGetString(0 , tostring(m_iniName), "Cycle", "0")
    m_cycletype:SetSelection(tonumber(val))
    m_spindle = AddSelectControl("Spindle Dir", {"None", "CW", "CCW"}, nil)

    m_Test = AddCheckControl("Check me")

    m_Test2 = AddRadioControl("Radio killed the video star")
AddRadioControl("yes")
AddRadioControl("No ")
AddRadioControl("okay")

end
function SaveSettings()

    mc.mcProfileWriteString(0 , tostring(m_iniName), "Xcenter", tostring(m_center_x:GetValue()))
    mc.mcProfileWriteString(0 , tostring(m_iniName), "Ycenter", tostring(m_center_y:GetValue()))
    mc.mcProfileWriteString(0 , tostring(m_iniName), "Dia", tostring(m_circle_dia:GetValue()))
    mc.mcProfileWriteString(0 , tostring(m_iniName), "Holes", tostring(m_NumHoles:GetValue()))
    mc.mcProfileWriteString(0 , tostring(m_iniName), "Depth", tostring(m_z_depth:GetValue()))
    mc.mcProfileWriteString(0 , tostring(m_iniName), "RapidHeight",
tostring(m_rapid_height:GetValue()))
    mc.mcProfileWriteString(0 , tostring(m_iniName), "RetractHeight",
tostring(m_retract_height:GetValue()))
    mc.mcProfileWriteString(0 , tostring(m_iniName), "PeckDepth",
tostring(m_peck_depth:GetValue()))
    mc.mcProfileWriteString(0 , tostring(m_iniName), "Feed", tostring(m_feedrate:GetValue()))
    mc.mcProfileWriteString(0 , tostring(m_iniName), "Cycle",
tostring(m_cycletype:GetCurrentSelection()))
end
function GenGcode()
        local x_center = m_center_x:GetValue()
        local y_center = m_center_y:GetValue()
        local numberofholes = m_NumHoles:GetValue()
        local dia = m_circle_dia:GetValue()
        local stAngle = (math.pi/180)*m_StartAngle:GetValue()
        local RapidHeight = m_rapid_height:GetValue()
        local retheight = m_retract_height:GetValue()
        local peck = m_peck_depth:GetValue()
        local feed = m_feedrate:GetValue()
        local depth = m_z_depth:GetValue()
        local gcode = string.format("G00 Z%.4f\n", RapidHeight)

        local drilltype = m_cycletype:GetCurrentSelection()

        local x = x_center+(dia/2)*math.cos(stAngle);
        local y = y_center+(dia/2)*math.sin(stAngle);

        local spin = m_spindle:GetCurrentSelection()
        if(spin == 0)then
            gcode = gcode .. "(No Spindle M Code)\n"
        elseif(spin == 1) then
            gcode = gcode .. "M03 (Spinle CW)\n"
        elseif(spin == 2) then
```

```
            gcode = gcode .. "M04 (Spinle CCW)\n"
        end

        if(drilltype == 0)then
            gcode = gcode .. string.format("G81 X%.4f Y%.4f Z%.4f R%.4f F%.4f\n", x, y, depth,
retheight, feed )
        elseif(drilltype == 1) then
            gcode = gcode .. string.format("G83 X%.4f Y%.4f Z%.4f R%.4f Q%.4f F%.4f\n", x, y,
depth, retheight, peck, feed )
        elseif(drilltype == 2) then
            gcode = gcode .. string.format("G73 X%.4f Y%.4f Z%.4f R%.4f Q%.4f F%.4f\n", x, y,
depth, retheight, peck, feed )
        end

        local StepAng= (2*math.pi)/numberofholes;
        local i
        for i=1, numberofholes-1, 1  do
            x=x_center+(dia/2)*math.cos(stAngle+(StepAng*i));
            y=y_center+(dia/2)*math.sin(stAngle+(StepAng*i));
            gcode = gcode .. string.format("X%.4f Y%.4f\n", x, y )
        end
        gcode = gcode .. "G80\nM05\nM30"

        local file = wx.wxFileDialog(panel, "Select Gcode File", "", "", "Text files (*.txt)|
*.txt|Tap files (*.tap)|*.tap",
                              wx.wxFD_SAVE,wx.wxDefaultPosition,wx.wxDefaultSize, "File Dialog"
);
        if(file:ShowModal() == wx.wxID_OK)then
            local path = file:GetPath()
            --wx.wxMessageBox(tostring(path))
            io.output(io.open(path,"w"))
            io.write(gcode)
            io.close()
            mc.mcCntlLoadGcodeFile( 0, tostring(path))
        end
        SaveSettings()
end
function main()

if(mcLuaPanelParent == nil)then
    -- create the wxFrame window
    mainframe = wx.wxFrame( wx.NULL,          -- no parent
                        wx.wxID_ANY,          -- whatever for wxWindow ID
                        "Mach4 Bolt Hole Wizard", -- frame caption
                        wx.wxDefaultPosition, -- place the frame in default position
                        wx.wxDefaultSize,     -- default frame size
                        wx.wxDEFAULT_FRAME_STYLE ) -- use default frame styles

    -- create a panel in the frame
    panel = wx.wxPanel(mainframe, wx.wxID_ANY)

    -- create a simple file menu with an exit
    local fileMenu = wx.wxMenu()
    fileMenu:Append(wx.wxID_EXIT, "E&xit", "Quit the wizard")

    -- create a simple help menu
    local helpMenu = wx.wxMenu()
    helpMenu:Append(wx.wxID_ABOUT, "&About", "About Bolt Hole Wizard")

    -- create a menu bar and append the file and help menus
    local menuBar = wx.wxMenuBar()
    menuBar:Append(fileMenu, "&File")
    menuBar:Append(helpMenu, "&Help")

    -- attach the menu bar into the frame
    mainframe:SetMenuBar(menuBar)

    -- create a simple status bar
    mainframe:CreateStatusBar(1)
    mainframe:SetStatusText("No Error.")

    -- connect the selection event of the exit menu item to an
    -- event handler that closes the window
    mainframe:Connect(wx.wxID_EXIT,
                      wx.wxEVT_COMMAND_MENU_SELECTED,
                      function (event)
                          mainframe:Close(true)
                      end )

    -- connect the selection event of the about menu item
    mainframe:Connect(wx.wxID_ABOUT, wx.wxEVT_COMMAND_MENU_SELECTED,
        function (event)
             wx.wxMessageBox("Bolt Hole pattern wizard \n\nAuthor: Brian Barker\nDate: 8/3/13\
nThis wizard is to be used as an example of how to make a wizard",
                            "About wxLua",
                            wx.wxOK + wx.wxICON_INFORMATION,
                            mainframe)
        end )

else
    panel = mcLuaPanelParent
end

    --Set up the main sizer so we can start adding controls
    local mainSizer = wx.wxBoxSizer(wx.wxVERTICAL)
    local InputsGridSizer  = wx.wxFlexGridSizer( 2, 4, 0, 0 )
    InputsGridSizer:AddGrowableCol(1, 0)

    function AddInputControl(name_string, width)
        if(width == nil)then
            width = 100
        end
        local textCtrlID = GetNextID()
        local staticText = wx.wxStaticText( panel, wx.wxID_ANY, name_string)
        local textCtrl   = wx.wxTextCtrl( panel, textCtrlID, "0.000", wx.wxDefaultPosition,
wx.wxSize(width, -1), wx.wxTE_PROCESS_ENTER ,wx.wxTextValidator(wx.wxFILTER_NUMERIC))
        InputsGridSizer:Add( staticText, 0, wx.wxALIGN_CENTER_VERTICAL+wx.wxALL+wx.wxALIGN_RIGHT,
2)
        InputsGridSizer:Add( textCtrl,   0, wx.wxGROW+wx.wxALIGN_CENTER+wx.wxALL+wx.wxALIGN_LEFT,
2)
        return textCtrl, textCtrlID
    end

    function AddCheckControl(name_string)
        local ID = GetNextID()
        local staticText = wx.wxStaticText( panel, wx.wxID_ANY, name_string)
        local Ctrl   = wx.wxCheckBox( panel, ID, "", wx.wxDefaultPosition, wx.wxDefaultSize,
wx.wxTE_PROCESS_ENTER ,wx.wxTextValidator(wx.wxFILTER_NUMERIC))
        InputsGridSizer:Add( staticText, 0, wx.wxALIGN_CENTER_VERTICAL+wx.wxALL+wx.wxALIGN_RIGHT,
2)
        InputsGridSizer:Add( Ctrl,   0, wx.wxGROW+wx.wxALIGN_CENTER+wx.wxALL+wx.wxALIGN_LEFT, 2)
        return Ctrl, ID
    end

    function AddRadioControl(name_string)
        local ID = GetNextID()
        local sizer = wx.wxBoxSizer( wx.wxHORIZONTAL )
        local Ctrl   = wx.wxRadioButton( panel, ID, name_string, wx.wxDefaultPosition,
wx.wxDefaultSize, wx.wxTE_PROCESS_ENTER ,wx.wxTextValidator(wx.wxFILTER_NUMERIC))
        InputsGridSizer:Add( sizer, 0, wx.wxALIGN_CENTER_VERTICAL+wx.wxALL+wx.wxALIGN_RIGHT, 2)
        InputsGridSizer:Add( Ctrl,   0, wx.wxGROW+wx.wxALIGN_CENTER+wx.wxALL+wx.wxALIGN_LEFT, 2)
        return Ctrl, ID
    end

    function AddSelectControl(name_string, selections, selCtrlID)
        local selCtrlID = GetNextID()
        local staticText = wx.wxStaticText( panel, wx.wxID_ANY, name_string )
        local selCtrl   = wx.wxComboBox(panel, selCtrlID, "", wx.wxDefaultPosition,
wx.wxSize(100, -1), selections)
        selCtrl:SetSelection(0)
        InputsGridSizer:Add( staticText, 0, wx.wxALIGN_CENTER_VERTICAL+wx.wxALL+wx.wxALIGN_RIGHT,
2)
        InputsGridSizer:Add( selCtrl,   0, wx.wxGROW+wx.wxALIGN_CENTER, 2)
        return selCtrl, selCtrlID
    end
    -- Add image to the top
    local hbmp = wx.wxBitmap(m_image)
    local TopImage = wx.wxStaticBitmap(panel, wx.wxID_ANY, hbmp )
    --Setup the inputs
    Setupinputs()
    -- make the bottom buttons
    local buttonSizer = wx.wxBoxSizer( wx.wxHORIZONTAL )
    local genGcode = wx.wxButton( panel, ID_GENGCODE_BUT, "&PostGcode")
    genGcode:SetBackgroundColour(wx.wxColour(0,255, 128))

    buttonSizer:Add(    genGcode, 0, wx.wxALIGN_CENTER+wx.wxALL, 2 )
    if(mcLuaPanelParent == nil)then
        local closeButton = wx.wxButton( panel, ID_CLOSE_BUTTON, "E&xit")
        buttonSizer:Add( closeButton, 0, wx.wxALIGN_CENTER+wx.wxALL, 2 )
    end

    --Set up the sizers
    mainSizer:Add(        TopImage, 0, wx.wxALIGN_CENTER+wx.wxALL, 2 )
    mainSizer:Add( InputsGridSizer, 0, wx.wxALIGN_CENTER+wx.wxALL, 2 )
    mainSizer:Add(     buttonSizer, 0, wx.wxALIGN_CENTER+wx.wxALL, 2 )

    panel:SetSizer( mainSizer )

    panel:Connect(ID_GENGCODE_BUT, wx.wxEVT_COMMAND_BUTTON_CLICKED,
    function(event)
       GenGcode()
    end)

    panel:Connect(ID_CYCLE_TYPE, wx.wxEVT_COMMAND_COMBOBOX_SELECTED,
    function(event)
        if(m_cycletype:GetCurrentSelection() == 0)then
            m_peck_depth:SetEditable(false)
            m_peck_depth:SetBackgroundColour(wx.wxColour("LIGHT GRAY"))
        else
            m_peck_depth:SetEditable(true)
            m_peck_depth:SetBackgroundColour(wx.wxColour(wx.wxNullColour))
        end

    end)



-- Connect a handler for pressing enter in the textctrls
    panel:Connect(wx.wxID_ANY, wx.wxEVT_COMMAND_TEXT_ENTER,
    function(event)
        -- Send "fake" button press to do calculation.
        -- Button ids have been set to be -1 from textctrl ids.
      --  dialog:ProcessEvent(wx.wxCommandEvent(wx.wxEVT_COMMAND_BUTTON_CLICKED, event:GetId()-
1))
    end)

    -- show the frame window

    if(mcLuaPanelParent == nil)then
        panel:Connect(ID_CLOSE_BUTTON, wx.wxEVT_COMMAND_BUTTON_CLICKED,
                        function(event) mainframe:Destroy() end)
        panel:Fit()
        mainframe:Fit()
        mainframe:Show(true)
    end
end

main()

wx.wxGetApp():MainLoop()
```

First is “m_iniName”. Variable values from the input boxes will be stored in a section in the machine.ini file. Set “m_iniName” to be something descriptive, usually similar to the name of the wizard. This helps with organization in the .ini file.

```
m_iniName = "CircleHolePat"
```

Next is “m_image.” This variable sets the path to the image to be displayed at the top of the wizard window. Change the file path and name to match the location of the desired image.

```
m_image = wx.wxGetCwd() .. \\Wizards\\HolesNew.png
```

The first function is Setupinputs(). This function defines the inputs to be displayed at the bottom of the wizard window. The formatting is already done, all that is required is to add or remove lines and rename them to the desired variable and description. Generally each input will have two lines of code, the first calls the function AddInputControl() to add the label and input box to the wizard window. In the snippet below "m_center_x" will be assigned the value of whatever is input into the input box labeled "Hole Center X". The second line sets the value of "m_center_x" and the input box to the last value that was saved in the machine.ini file, in the "CircleHolePat" section with the handle "Xcenter". If there is no value in the .ini then a default value of 0.00 will be assigned.

```
    m_center_x = AddInputControl("Hole Center X",nil)
    m_center_x:SetValue( mc.mcProfileGetString(0 , tostring(m_iniName), "Xcenter", "0.000") )
```

Drop down selection boxes are also possible using the following format:

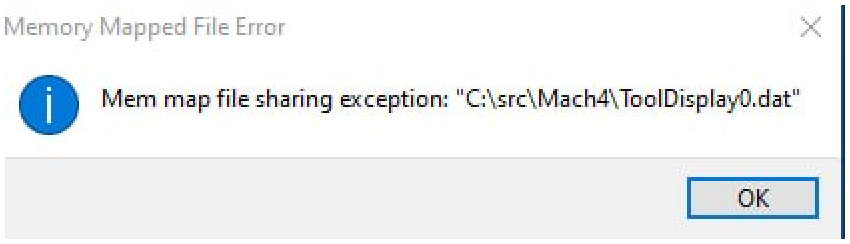
```
    m_spindle = AddSelectControl("Spindle Dir", {"None", "CW", "CCW"}, nil)
```

Radio controls and check boxes also have functions for easy creation.

```
    m_Test = AddCheckControl("Check me")

    m_Test2 = AddRadioControl("Radio killed the video star")
```

Troubleshooting

Memory Mapped File Error:


Memory Mapped File Error

Mem map file sharing exception: "C:\src\Mach4\ToolDisplay0.dat"

If you're getting this error, check the file that seems to be causing this for mc.mcToolPathGenerate(). This used to have to be called every time a file was loaded to regenerate the tool path but this call is now automatically called every time a file is loaded. The error is called by both of them trying to edit the tool path file at the same time. If it's present after a GCODE load call then try commenting it out.