The MachStdMill project: Customizing CNC Control Software. Part 3

By David Bagby, Calypso Ventures, Inc.

In part 1 of this series, we looked at an updated User Interface which resulted from screen pages designed to model the work flow steps common to personal CNC machinists. In part 2 we covered example enhanced features that can significantly simplify personal CNC operations by making the CNC control automate many of the CNC operator's tasks.

Early in the MSM development effort, it became apparent that key project goals were impractical using only the scripting tools Mach then provided. After some of the initial capability had been prototyped, it was evident that software modularity and maintenance would be significant issues. Fortunately, I was able to work with ArtSoft to extend Mach's script capabilities to address the project's needs. My thanks to Brian Barker of ArtSoft USA for his supporting efforts.

The resulting Mach script extensions are now available to all Mach users. Many Mach users are probably unaware of the capabilities provided by the scripting extensions. We'll cover the key Mach scripting extensions and how you can use them in your own Mach script development.

In this article, I'll introduce the key enhanced programming interfaces, and touch on when & why one should consider their use. I'll also explain how some prior "macro" coding practices are probably no longer desirable.

The enhanced script interfaces were introduced in the Mach 3.43.xx series. Mach 3.43.22 is the first lockdown release which contains the extended script interfaces.

The two most important additions to the Mach scripting facilities are #Expand and RunScript.

Script Preprocessing: #Expand

#Expand is a simple text substitution mechanism which was modeled along the lines of the C language #include.

#Expand allows a script file to reference source code that lives in a separate disk file. The contents of the disk file are used to replace the #Expand line before the script is executed. The addition of the simple #Expand to Mach made the MachStdMill project possible.

A bit of historical context will illustrate the value of #expand:
I'd written a pre-processor application for Mach scripts when it became clear that MSM's goals were impractical without the ability to share source code between multiple scripts.

Back then, the code that does page changes in MSM was stored inside the screen set .set file (as were all scripts prior to the introduction of #Expand).

In MSM, there are around 150 places a page can get changed... debugging the initial page change code this was a nightmare as **any** change meant a change to all 150+ separate copies of the code inside the set file. Making matters worse, MSM supports multiple separate set files that have to be kept consistent (multiple supported screen resolutions etc)… that made 600+ places to edit for each revision to the PageChange code.

ArtSoft understood the problem and offered to add simple script pre-processing to Mach. This would provide a better tool for all Mach script writers and it eliminated the need for CVI to use a custom script pre-processor to develop MSM.

With the use of #Expand, this is the typical page change button code in MSM:

```
Option Explicit
#expand <Masters\Headers\CopyRightAndLicenseNotice>
#expand <Masters\Headers\MSMpageNames>
#expand <Masters\Headers\MSMPageConstants>

Call MSMPageChange(MSMReferencePage )
exit sub

#expand <Masters\Scripts\Common\Expands\MSMPageChange>
```

The first #Expand line pulls in the vendor copyright notice from a separate file which contains the common copyright legal language.

The second #Expand line pulls in the names of all the pages in MSM from the common page name definition file (thus avoiding the undesirable use of any hard coded screen set page numbers).

The third #Expand file pulls in the name definition of the MSM user space DROs used by MSM to handle page changes (avoiding the use of hard coded magic numbers which is not considered good programming practice).

There are only two lines of code unique to the button: the call to the MSMPageChange routine with the page name parameter, and a line to exit the button subroutine. The only thing that changes from one page button to another is the PageName parameter.

Finally, the last #Expand line pulls in the actual PageChange source code from the file where it lives.

This is a huge improvement; instead of needing to edit 600 copies of the PageChange code, it is now only necessary to edit the single copy in the MSMPageChange.m1s file!

#Expand is only a simple subset of the preprocessor facilities commonly found in modern programming languages. Even so, it enables a quantum leap forward in the ability to write modular, maintainable Mach scripts. If you have code that you use in more than one place, it will pay off to learn how to use #Expand.

The syntax for #Expand allows the programmer to specify where to look for expansion scripts. The #Expand <QFN[1]> format tells Mach to search for the expand file in a location dependant on the currently loaded screen set. The search path used is:

    <Mach install dir>\ScreenSetMacros\<ActiveScreenSetName>.set\

Using MachStdMill as the active screen set example, and assuming the standard Mach install directory, this would be:

    C:\Mach3\ScreenSetMacros\MachStdMill.set\

The syntax allows all screens sets to have a structured file location where their scripts are found by Mach. This is a much cleaner approach than dumping files into the Mach install directory (as some older screen sets do). Screen set writers are encouraged to use and follow these conventions so that multiple installed screen sets may easily coexist without script file conflicts.

For those interested in more details, the pre-processing facilities for Mach scripts are documented in the Mach Programmers reference manual.

Hammers and Nails…
Ever hear the old saying that "when the only tool you own is a hammer, all problems look like nails"?

Mach allows users to create "user defined M-codes", and to invoke them from G-code. Mach reserves M-Code numbers from 0-99 and allows User M-codes to be numbered from 100 – 9999. When a user M-code is encountered, Mach looks for a file of the name Mxxx to match the Mxxx code in the running G-Code program (i.e. M123 causes Mach to look for M123.mcc first and then M123.m1s).

Prior to Mach 3.43.xx there were only two places that could serve as storage for a script:
1) The screen set "set" file and 2) a M-Code disk file.

Since an Mcode file was the only type of file that Mach knew how to "call", Mach users tend to put all scripts into M-code files…. Mach was the hammer and M-code files are the nails.

The M-code file approach has some serious limitations.
a) M-code file names have to be of the Mxxx format. This does not promote the good programming practice of using human readable file names to identify file contents.
b) There are serious problems with one M-code "calling" another M-code. Mach runs M-codes as independent asynchronous threads and thread synchronization

[1] QFN = Qualified File Name

support is not provided by Mach. This leaves the casual script programmer with a serious asynchronous parallel programming challenge.

c) Mach searches for M-code files in a location which is Profile name dependant. This is not a very good model as it ties user program M-Codes to particular profiles (a profile is primarily a machine configuration file) and it forces M-codes used as system extensions to have to be co-located with M-Codes used in user G-Code programs.

These factors make the use of M-codes as system functionality extensions hard to implement, hard to manage, hard to understand and hard to maintain.

RunScript()

To improve this situation, a second key scripting extension was added: RunScript().

RunScript is a programming mechanism to call a script file which is resident on disk. The use of RunScript addresses most (but not all) of the major shortcomings of the use M-code files as “script code modules”.

The Mach Programmer’s reference manual says

> ‘Prior to the addition of this call, it was common practice to put script code into a Mxxx.m1s macro, and use the Code call to execute the Mxxx macro. The execution of an Mxxx macro involves the use of the Mach G-Code interpreter (as what you are really doing is executing a G-Code M word block) and can result in the programmer having to invent and handle semaphores to coordinate the asynchronous execution of the Mxxx macro.
>
> It is recommended that the use of the Code “as a subroutine call method” be avoided and that, when possible, the RunScript call should be used instead.” ’

The semantics of RunScript are similar to a standard Call statement in Basic, i.e. code flow is not continued past the RunScript call until the called script has finished execution. RunScript avoids the asynchronous complexity required when a M-code calls an M-code and there is no (subtle and sometimes unpredictable) interaction required with the mach G-Code interpreter.

My advise is that unless you are actually creating a M-Code for use within a G-Code program, new scripts should be designed to be stored in M-code files.

The “hammer syndrome” has impacted the vocabulary of Mach script writers. Many Mach users refer to a Mach Script as a “macro”. This is a historical side effect of there being only one type of script container: the M-Code file. In an effort to avoid confusion from the use of the historical term, these articles have consistently referred to Mach’s Basic language source code as “Scripts” and the use of “macros” was reserved for scripts which are actually M-Code implementations.

Here is an example script which uses RunScript to call a second script from a disk file:

```
Option Explicit
Const CR = 13  ' carriage return char

' we don't do much here in the outer layer of the macro -
' Just call main function...

        Call ScriptMain()                ' call macro main outer block of code
        Exit Sub

' Script has completed.


Sub ScriptMain()
        Dim i As Integer

        MsgBox ("1st script called - about to use RunScript to start 2nd script...")

        Call RunScriptWrapper("TestDiskScript1")

        MsgBox("1st script ending")

        Exit Sub

End Sub         ' ScriptMain

Sub RunScriptWrapper(ByVal ScriptName As String)
        Dim QFN As String
        Dim result As Integer

        QFN = "DemoScripts" & "\" & ScriptName

        MsgBox "Script to run: " & Chr(CR) & "    " & QFN

        result = RunScript(QFN)

        If result < 0 Then
                MsgBox "Script not run: " & Chr(CR) & "    " & _
                QFN & Chr (CR) & _
                "Error result = " & result, 48, "RunScript Error."
        End If
End Sub
```

The “upper” level script uses a subroutine to create a wrapper around the actual call to RunScript. This provides a convenient location to place logic re where to look for the script to be called. User scripts are not normally a function of the loaded screen set. It is necessary to pass a QFN to RunScript which is relative to the Mach install directory (RunScript doesn’t do the <> syntax location searching that #expand does).

The sample wrapper assumes the demo scripts are in the “DemoScripts” subdirectory within the Mach3 install directory. The wrapper technique can also be used to mitigate a RunScript limitation (discussed below).

Here is the script that RunScript “calls”:

```
Option Explicit

' we don't do much here in the outer layer of the macro -
' Just call main script function...

        Call ScriptMain()                 ' call script main outer block of code
        Exit Sub

Sub ScriptMain()
        Dim j As Integer, k As Integer

        MsgBox ("2nd script alive... will create a msgbox every 10 sec")
        For j = 1 To 30 Step 1
                ' error notifications can't be seen and handled while sleeping –
                ' so we loop 10 sec of short sleeps to be able to respond if need be
                For k = j To 100 Step 1
                        sleep 100
                Next
                MsgBox ("2nd script still alive after " & j*10 & " secs...")
        Next
        MsgBox("2nd script ending normally")

        Exit Sub
End Sub
```

You can run the upper level script from the Mach script editor and watch the sequence of message boxes to see that RunScript actually does implement “sequential Call” semantics.

RunScript does have a limitation compared to M-code files. Since M-code files are processed as MCodes by the Mach G-code interpreter, it is possible to pass parameters with numeric values to M-codes using the P,Q,R words.

The RunScript implementation did not implement any parameter passing support for disk scripts. This is the other reason the script examples show a wrapper routine around RunScript: the wrapper can be used to implement parameter passing.

It’s not difficult to use a few Mach user space DROs (numeric values), LEDs (Booleans) or even UserLabels (strings) as global parameter passing locations. You can pass parameters to the wrapper, have the wrapper place the parameter values in the globals and the scripts can get the values from the user space globals.

Now that we have introduced how #Expand and RunScript, when would you want to use them and why?

Creating modularity:
The practice of storing scripts inside set files is only manageable for small, simple screen sets. A better practice is to let the set file contain only the screen set graphics rendering information. To accomplish that we need to “remove” the scripts from the set file; #expand can be used to accomplish this goal.

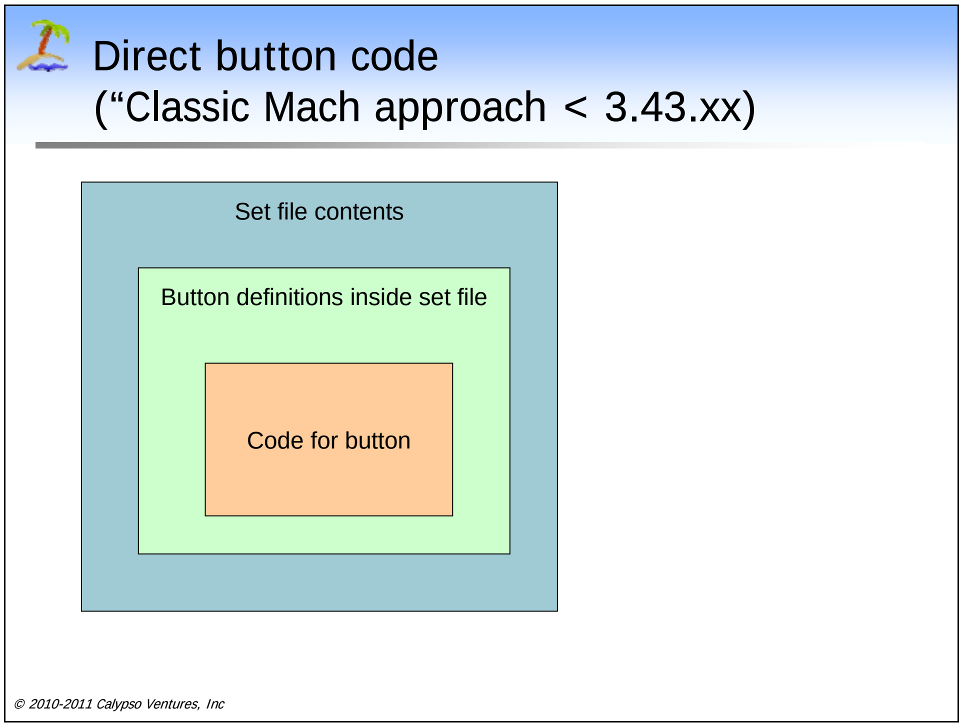
Let’s consider the classic situation of a button in Mach. When activated, Mach needs to run the script code associated with the button, and Mach uses the set file to find the code to execute.

Prior to Mach 3.43.xx button scripts were required to be stored inside the set file itself. I call these “direct coded buttons”.

With #expand we can cause Mach to load a script from a disk file. I call these types of buttons “indirect coded buttons”. Indirect buttons are the technique MSM used to solve the “600+ buttons to edit problem”.

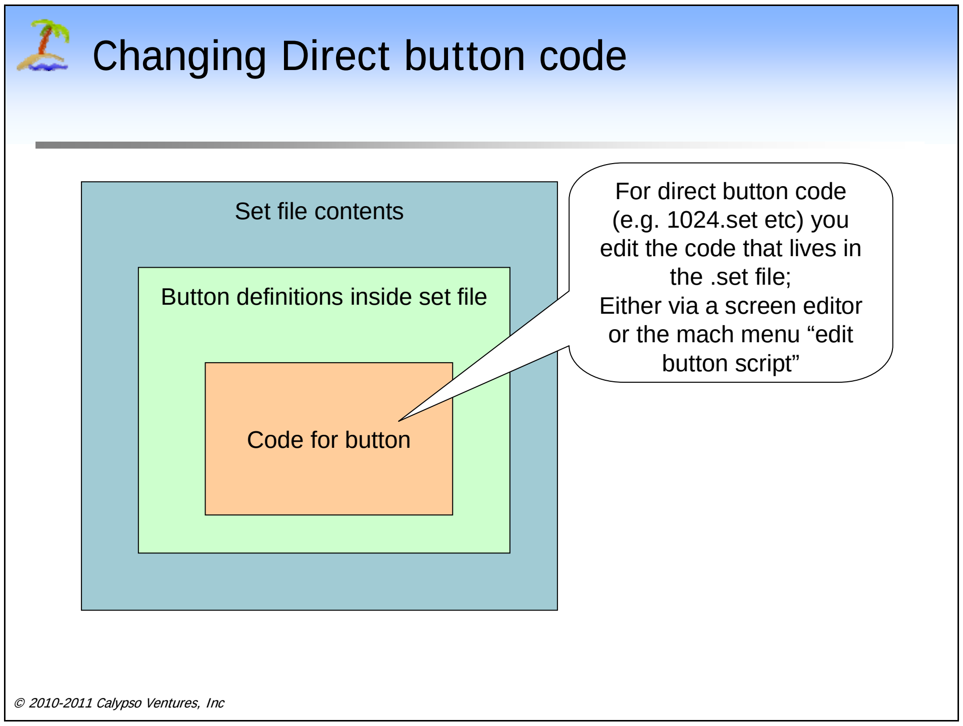
The following series of diagrams illustrate the differences between “direct” and “indirect” buttons.

In “Classic Mach” direct buttons are structured as shown in Figures 1 thru 4:



Direct button code
("Classic Mach approach < 3.43.xx)

Figure 1



Changing Direct button code

Set file contents
Button definitions inside set file
Code for button
For direct button code (e.g. 1024.set etc) you edit the code that lives in the .set file; Either via a screen editor or the mach menu "edit button script"

Figure 2

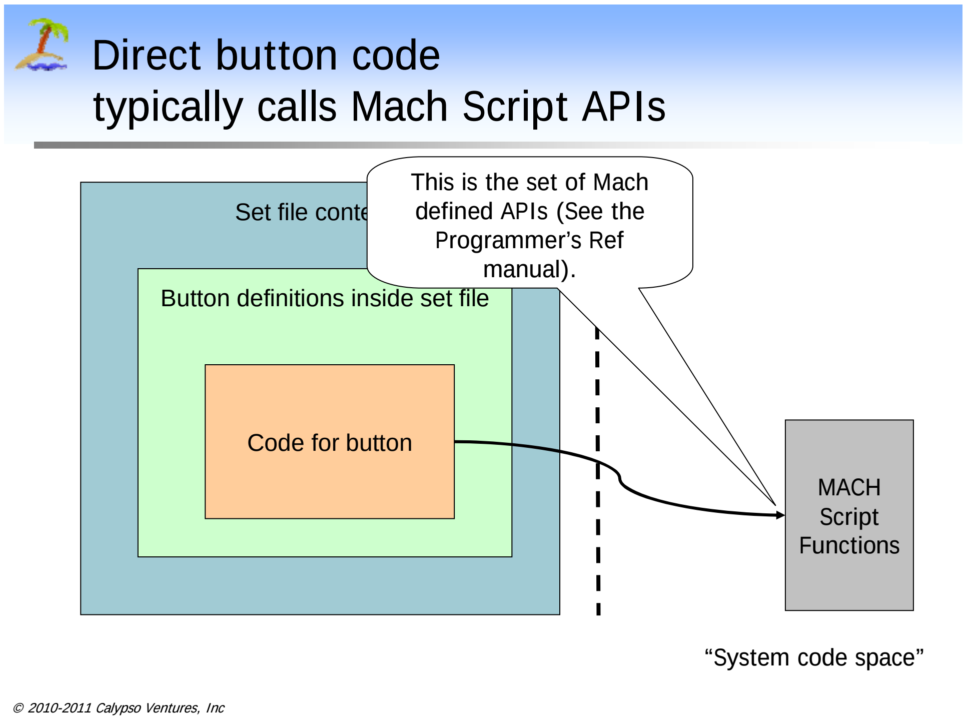

Direct button code
typically calls Mach Script APIs
Set file conte
This is the set of Mach defined APIs (See the Programmer's Ref manual).
Button definitions inside set file
Code for button
MACH Script Functions
"System code space"
© 2010-2011 Calypso Ventures, Inc

Figure 3

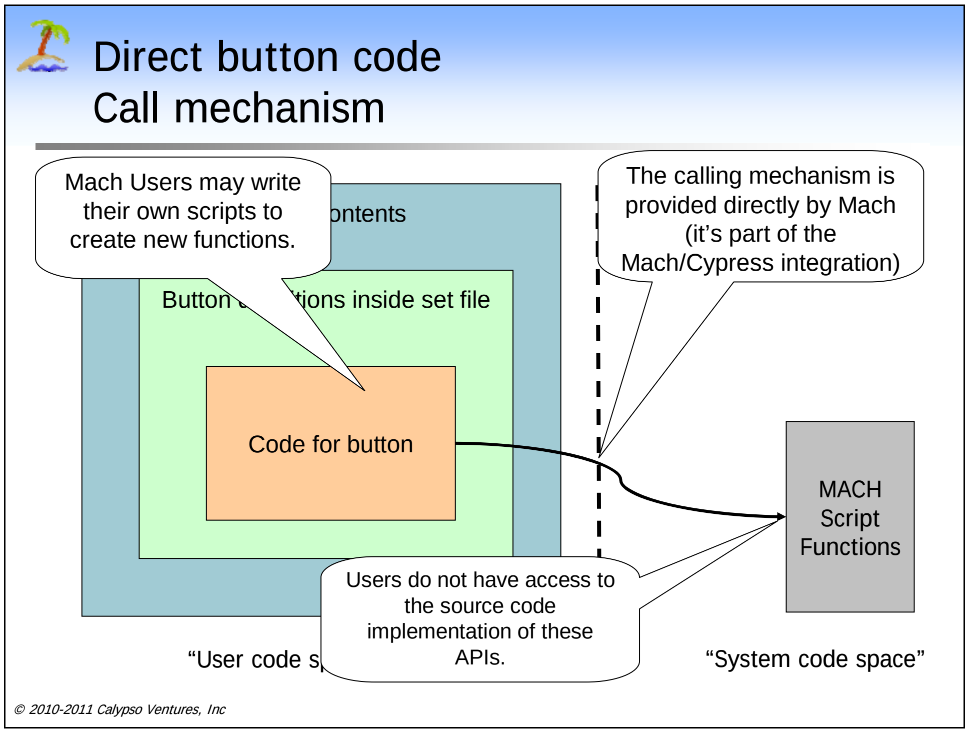

Direct button code
Call mechanism
Mach Users may write their own scripts to create new functions.
ontents
The calling mechanism is provided directly by Mach (it's part of the Mach/Cypress integration)
Button d tions inside set file
Code for button
MACH Script Functions
Users do not have access to the source code implementation of these APIs.
"User code s
"System code space"
© 2010-2011 Calypso Ventures, Inc

Figure 4

With "New Mach" (3.43.xx and later), #expand can be used to create Indirect button structures as shown in Figure 5.

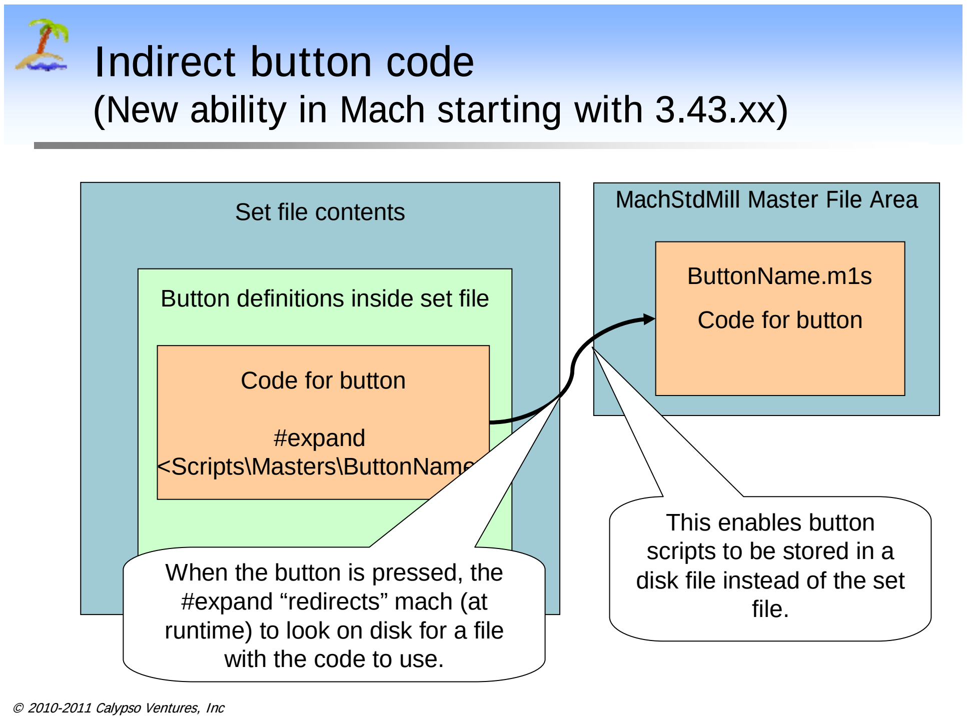

Indirect button code
(New ability in Mach starting with 3.43.xx)
Set file contents
Button definitions inside set file
Code for button
#expand
<Scripts\Masters\ButtonName
MachStdMill Master File Area
ButtonName.m1s
Code for button
When the button is pressed, the #expand “redirects” mach (at runtime) to look on disk for a file with the code to use.
This enables button scripts to be stored in a disk file instead of the set file.
© 2010-2011 Calypso Ventures, Inc

Figure 5

Protecting User Customizations

With the adoption of some file location conventions, the basic indirect button approach can be expanded to solve another problem. The use of M-Codes for system level extensions forces a co-mingling of system and user scripts. This creates a difficult problem wrt to software updates. How can button functionality be updated without wiping out user modifications? When all scripts are stored in the same container (the set file) this is pragmatically impossible.

Solving this problem was a prerequisite to being able to distribute MSM software updates. MSM uses the modularity gained from the use of indirect buttons to solve this problem.

Within the MSM screen set file area, two sub-trees are created. One sub-tree is for the use of the screen set and anything within that sub-tree may be altered, deleted or replaced by a software update. MSM calls this sub-tree “Masters”.

A second parallel sub-tree is created by MSM (as a user convenience) called “Custom”. MSM never modifies files within the “Custom” sub-tree.

The Masters/Custom parallel sub-tree structure makes it practical to change individual buttons. For an indirect coded button, all you need to do is point the button from the "Masters" sub tree to the "Custom" sub tree. This lets users change any button individually without impacting any other buttons.

For development work, it's much easier to alter a disk file than to open a set file and edit scripts embedded in the set file. It also means that a screen set is not forced to overwrite user scripts (as happens when it is all rolled up in a single set file and the install has to update the set file).

The worst an MSM user has to do when the software is updated is a search/replace of "Masters" to "Custom" in a few buttons. The user’s modifications are now up on the updated software release. Since all the other buttons are pointing to updated code, you get both the screen set software updates and your customizations. Of course you don't get an update for a button whose code you replaced - that's part of the cost of customizing a button.

This scheme isn't perfect, but it's a lot better than what is available with screen sets that only have direct coded buttons.

Figures 6 and 7 illustrate how this is used by MSM:

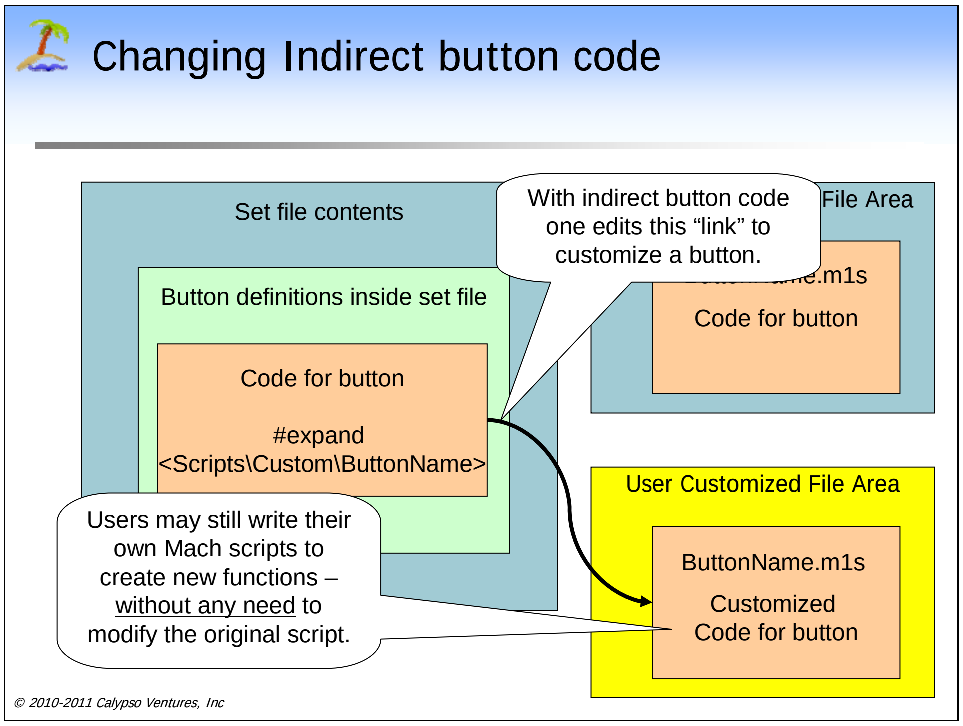
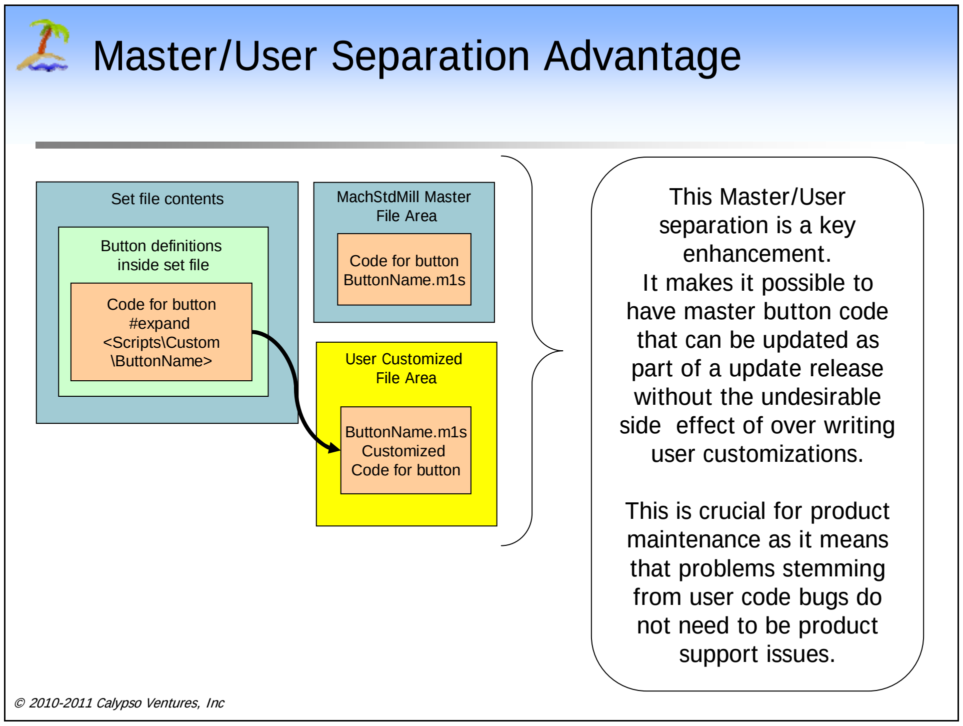

Changing Indirect button code
Set file contents
Button definitions inside set file
Code for button
#expand <Scripts\Custom\ButtonName>
With indirect button code one edits this “link” to customize a button.
File Area
.m1s
Code for button
User Customized File Area
ButtonName.m1s
Customized Code for button
Users may still write their own Mach scripts to create new functions – without any need to modify the original script.
© 2010-2011 Calypso Ventures, Inc

Figure 6



Master/User Separation Advantage

Set file contents
Button definitions inside set file
Code for button #expand <Scripts\Custom \ButtonName>
MachStdMill Master File Area
Code for button ButtonName.m1s
User Customized File Area
ButtonName.m1s Customized Code for button

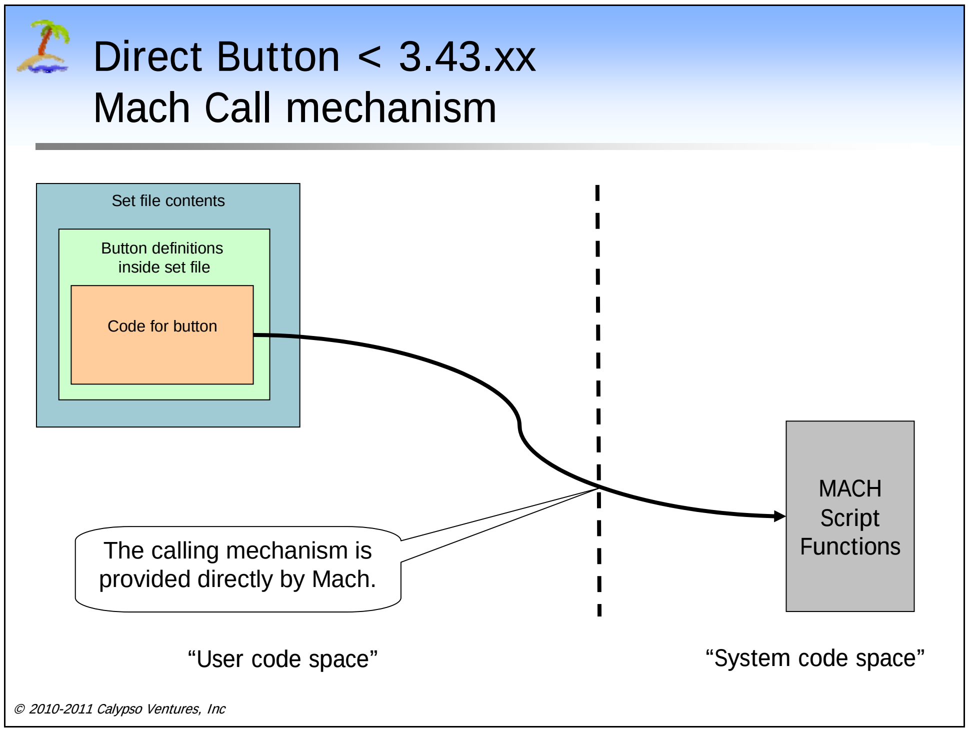
This Master/User separation is a key enhancement.
It makes it possible to have master button code that can be updated as part of a update release without the undesirable side effect of over writing user customizations.

This is crucial for product maintenance as it means that problems stemming from user code bugs do not need to be product support issues.

© 2010-2011 Calypso Ventures, Inc

Figure 7

The use of indirect buttons also has the nice effect of being able to create a separation between system functional extensions and user scripts (See Figures 8 thru 11).



Direct Button < 3.43.xx Mach Call mechanism

© 2010-2011 Calypso Ventures, Inc

Figure 8

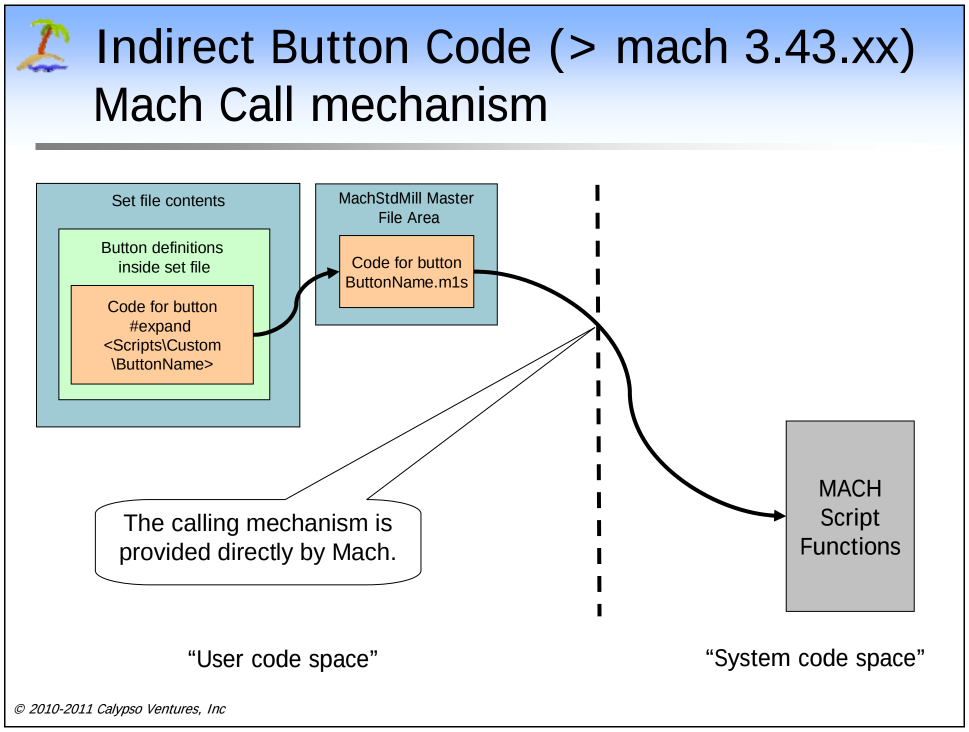

Indirect Button Code (> mach 3.43.xx)
Mach Call mechanism
Set file contents
Button definitions inside set file
Code for button #expand <Scripts\Custom \ButtonName>
MachStdMill Master File Area
Code for button ButtonName.m1s
MACH Script Functions
The calling mechanism is provided directly by Mach.
"User code space"
"System code space"
© 2010-2011 Calypso Ventures, Inc

Figure 9

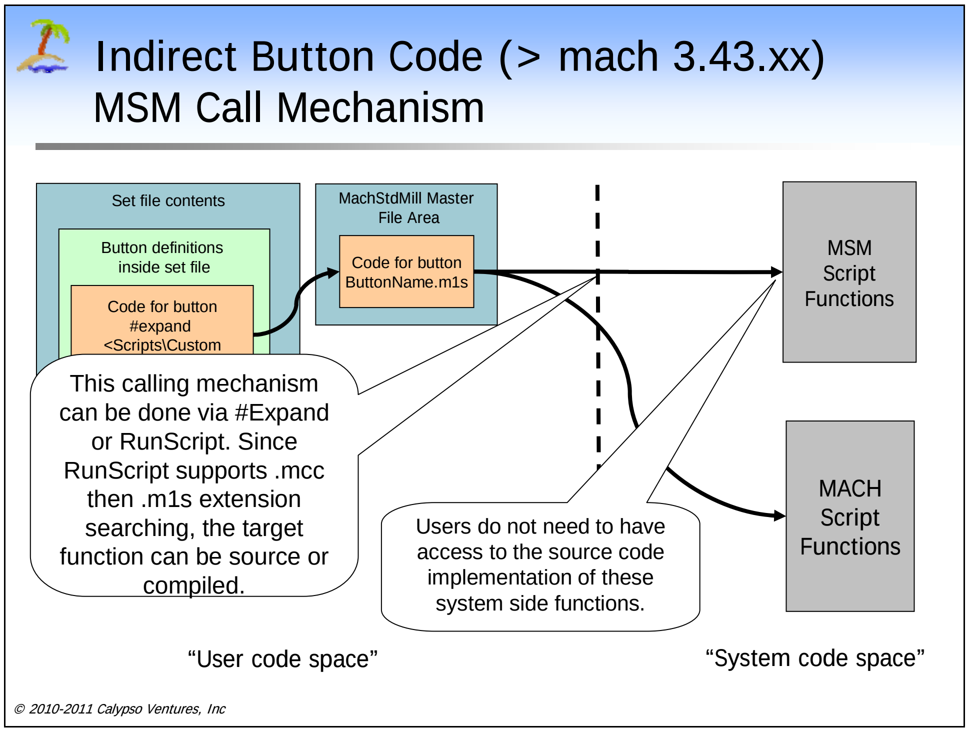

Indirect Button Code (> mach 3.43.xx)
MSM Call Mechanism
Set file contents
Button definitions inside set file
Code for button #expand <Scripts\Custom
MachStdMill Master File Area
Code for button ButtonName.m1s
MSM Script Functions
MACH Script Functions
This calling mechanism can be done via #Expand or RunScript. Since RunScript supports .mcc then .m1s extension searching, the target function can be source or compiled.
Users do not need to have access to the source code implementation of these system side functions.
"User code space"
"System code space"
© 2010-2011 Calypso Ventures, Inc

Figure 10

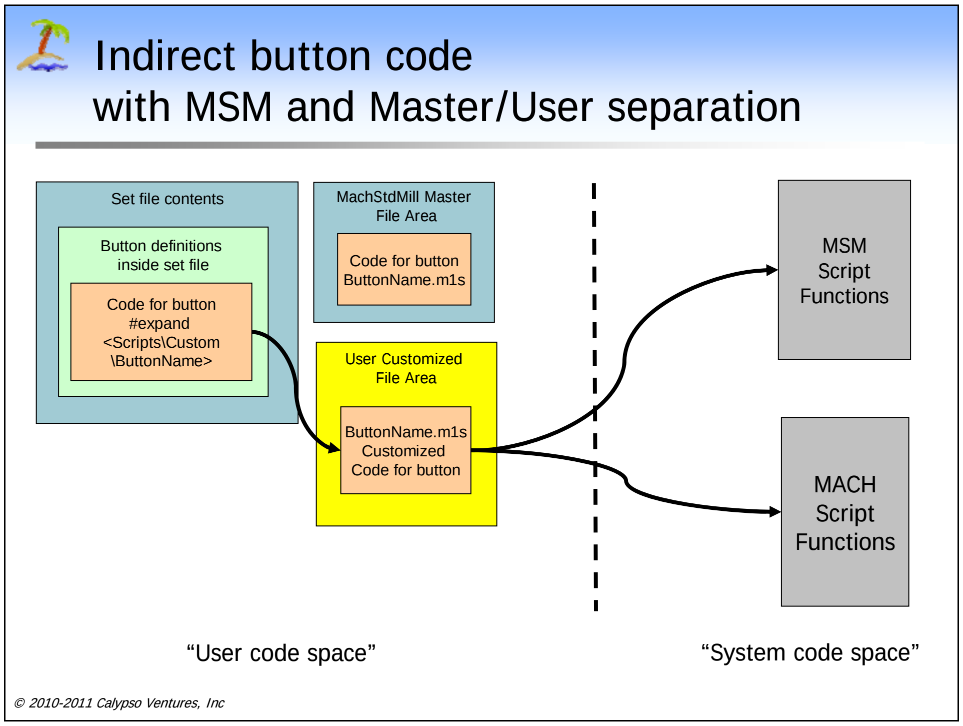

Indirect button code
with MSM and Master/User separation
Set file contents
Button definitions inside set file
Code for button #expand <Scripts\Custom \ButtonName>
MachStdMill Master File Area
Code for button ButtonName.m1s
User Customized File Area
ButtonName.m1s Customized Code for button
MSM Script Functions
MACH Script Functions
"User code space"
"System code space"
© 2010-2011 Calypso Ventures, Inc

Figure 11

We can now see how these two key Mach extensions enabled some key project design goals:

- Backward compatibility:
  Backwards compatibility considerations mandated that we avoid set file format changes.
  This approach avoided the need to alter .set file formats, thus avoiding any need to deal with set file version migration issues and the updates to screen editors that would require.

- Reasonable implementation effort
  The implementation time available in ArtSoft mandated that we find the minimal tools necessary to solve the problems at hand.

- Extends user abilities:
  The approach enables Mach users to utilize the new scripting features to build modular user customizations.

- Modularity and User/system code space separation
  The approach supported the needed separation between user and system extensions (so that system level extensions could be maintainable).

Clean Event Interfaces:

RunScript can also be used to create clean, modular interfaces between system extensions and user supplied scripts (without the need for a user to first understand a complex system extension script).

Here is a code snippet similar to what MSM uses to implement the calling of a user supplied script for ATC (automatic tool changer) logic.

```
Sub M6ATCWrapper()
        Const CR = 13

        Dim ATCScriptPathSpec As String
        Dim result As integer

        ' now to find and call the externally supplied auto tool changer code script
        '
        ' the script we run is not installed by MSM - it is expected to be supplied
        ' by the Auto TC implementer.
        ' we get the script contents at run time so that changing the ATC script
        ' does not require changes to MSM M6Start
        ' the ATC script is hardware profile dependant - so we look for it in the
        ' macros\profile name subdir

        ATCScriptPathSpec = "Macros\" & GetActiveProfileName() & _
                                "\M6ATC"

        result = RunScript(ATCScriptPathSpec)
        if result < 0 then
                ' error - script was not run
                msgbox "Script: " & Chr(CR) & "    " & ATCScriptPathSpec & _
                         Chr(CR) & _
                        "Error = " & RSErrorText(result), 48, _
                        "MSM M6ATCWrapper Error."

                ' abort TC and reset Mach, stop the G-code execution
                DoOEMButton( MachResetOEMBtn )
                MachMsg("Control RESET, Tool change Aborted", _
                "ERROR: Auto TC Extension Script not found!", 0)
        End If
        Exit Sub
End Sub
```

The use of RunScript enables the user to supply code unique to their hardware. MSM invokes the user script at the appropriate time during M6 processing.

The user only needs to understand his ATC mechanism and can concentrate on that as a independent code module.

Screen Set Initialization
Finally, I want to mention that Mach 3.43.xx provides some support for screen set initialization and cleanup.

When a screen set is started, Mach looks for a script in the appropriate ScreenSetMacros directory called “ScreenSetLoad”. This script is run (if it is present) by Mach whenever Mach loads the screen set. This provides a place to put code to initialize your screen set.

Similarly, whenever a screen set is unloaded, Mach looks for a script called “ScreenSetUnload” and runs it. This provides a place to put code to clean up screen set resources.

With the tools described in this article, it is possible create extensive customization of Mach’s feature set. We have seen examples of how these tools were used to create a new, modern Mach User Interface tailored to Personal CNC operators, and also how the tools were used to create system level functionality extensions for Mach.

MachStdMill has been an interesting project and the “hole in the bucket” is satisfactorily repaired. I hope that some of the design issues and techniques developed along the way will be of use to you in your own projects.

Correspondence regarding this article may be emailed to the author via MSM@CalypsoVentures.com. The MachStdMill software referred to in the article is available from www.Calypsoventures.com.