CALIBRATING A UUT ON A REMOTE COMPUTER USING FLUKE MET/CAL®

Michael L. Schwartz
Cal Lab Solutions, Inc.
PO Box 111113
Aurora, CO 80042
mschwartz@callabsolutions.com

Current and next generation test equipment will be modular and tightly coupled with a computer's operating system. Calibration labs have to be highly dynamic supporting all the measurement test systems turned in for calibration. This presents challenges for automation, but technologies can be designed to work together. This paper will show how a Fluke MET/CAL® procedure can be written integrating Metrology.NET® tools to remotely calibrate a UUT connected to a remote computer on a completely different operating system. To do so, we will first cover the basic design patterns of remote computing, show how we create the command interface for a non-message based instrument, then how to remotely communicate with the instrument.

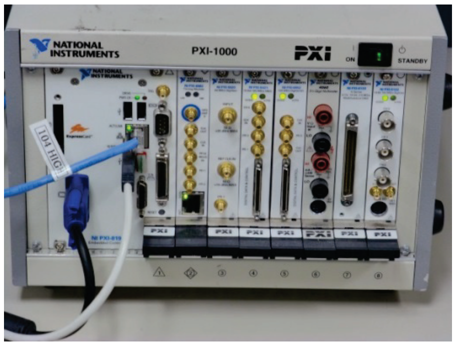
PROBLEM

PXI & PXIE instruments are growing in popularity, so calibration labs are working to find solutions to support them internally. Many manufacturers have created support solutions to help self-maintainers, but not all calibration labs are able to retool to support the software. High accuracy standards are very expensive; so many calibration labs are looking to support more equipment with a smaller footprint of hardware.

In this example, one of our customers was looking for a custom solution to support National Instruments PXI 5122. They have and use the Calibration Executive from National Instruments, but do not support enough oscilloscopes to justify the purchase of a Fluke 9500. Instead, their lab has a Fluke 5520—a standard fully capable of calibrating the PXI 5122.

Looking at the Problem in the Problem Domain

- The calibration lab needs a way to support the PXI-5122 in house
- They do not have a Fluke 9500
- They have a Fluke 5520
- Testing them manually is not an option



OUR APPROACH

Fluke MET/CAL® is a very popular software tool used by many calibration labs around the world. For our problem with testing PXI instruments, utilizing MET/CAL® seemed a good starting point. But controlling instruments like PXI and other modular and software based instrumentation presents itself with a level of complexity.

Having recently developed procedures for two other PXI instruments from another manufacturer, we learned software based instruments do not always run on every operating system. We needed to create a software development model that would contain complexity of the software in an easy to understand programming interface, while at the same time, minimizing operator frustrations related to configuration and operation of the calibration procedure.

We know the life expectancy of most hardware is ten to fifteen years, while software is only five years, so we know we have to expect the software platforms around the hardware will change two to three times over the life of the hardware. We also know that not all systems and customers would upgrade software at the same time; ultimately, we needed to decouple the UUT code from the standards code.

OVERVIEW OF OUR SOLUTION

Working with our customer, we decided to divide the software up into two sections: the UUT code and the standard's code. We have decoupled the UUT code from the standard's code before, but this time we wanted our solution to be able to run cross computer. This would allow the UUT portion of the code to run on a completely different computer and operating system. It would also allow the UUT to be tested in the PXI chassis without the need for the technician to install MET/CAL® on the PXI Controller.

To accomplish this first we had to create a text command interface for the UUT, allowing MET/CAL to control the UUT using commands similar to how we would control equipment on the GPIB bus. Then we needed to create a service to run on the UUT PXI chassis that would translate the commands to instrument function calls. It would be able to read the message, configure the instrument making measurements, and return the results.

Once the UUT Service was completed, we then needed to create a client messaging application because MET/CAL® at present does not have any direct access to TCP/IP based message calls. This was the simplest part of all the code, mostly because we already had tools developed to facilitate integration with MET/CAL®.

Once all the tools were developed, all we had left to do was to write the actual MET/CAL® procedure and then test it, in order to make sure the software was able to run distributed across two computers or all on the same computer.

Cal Lab Solutions Software Layers

We start to see patterns over time as we build software. Different companies and different people will name and label objects and layers in their software differently. But when we step back, the patterns are often the same. As software developers, we have to recognize these patterns and use them to simplify our software designs.

At our highest level of software abstraction is our Metrology Service Bus Layer (see Figure 1). This layer is designed to be language agnostic and platform independent. It creates a common interchangeable layer for driver interchangeability. Below that is the Measurement Process Driver. These can be created in any language and their main focus is measurement quality.

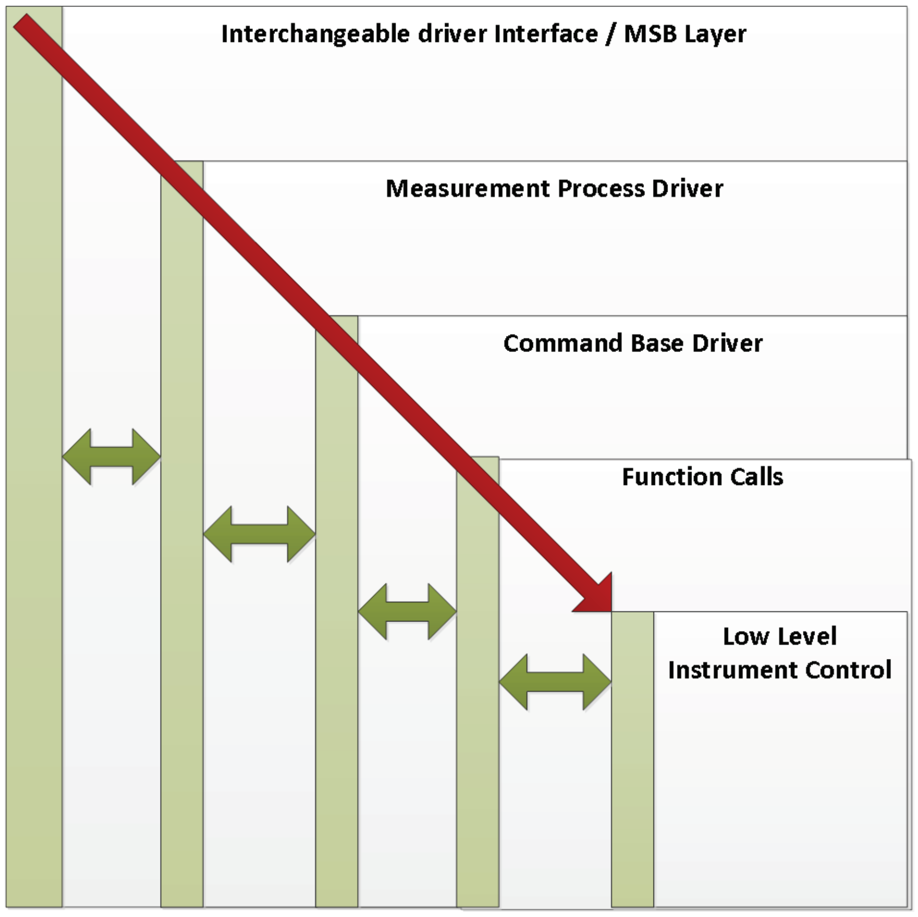

Interchangeable driver Interface / MSB Layer
Measurement Process Driver
Command Base Driver
Function Calls
Low Level
Instrument Control

Figure 1

I should point out that not all instruments are command based; some measurement drivers can communicate directly with the function call and low level instrument control. But the overall goal of this development model is to create more flexibility in software design, so we can include it as part of our standard development methodology.

The command base driver is essentially the IEEE SCPI calls and RS-232 programming we normally use to send commands to an instrument. The firmware in the instrument will recognize our string based commands and call the corresponding function that performs that operation. For example, when sending it a "*RST" command, the firmware would call an internal Reset() function placing the instrument in a known state. The Reset() function would call all the low level peaks and pokes to set up the hardware.

Creating a Command Set

Most instruments are controlled using a command based language, but we are seeing more and more of the newer software based instruments do not support a command based language. Instead, they expose function calls that are accessed directly from our software. This direct exposure to the function calls allows for faster execution and better performance, but then creates issues for the calibration lab because much of our software has been built for command based instruments.

To communicate with a software-based instrument using a message based command language requires a command processer. Writing a command process that will convert the simple text based commands to function calls can be a very simple process. First you have to define the command language then write a string parser to process each command. One simple solution is to create a giant case select that acts on each command. As the command processer receives each command, it will read the strings content and call correct functions for the instrument.

Keeping things simple and streamlining the parsing process, we used a command followed by name value pair format:

```
<Command>: [<Name>= <Value>] [,<Name>= <Value>]
example
ConfigureVertical: Channel= 1, Coupling= DC, Attenuation= 0, Range= 10, Offset= 0
```

For the project, we used the following commands mapping them to function calls:

Command	**Function Call**
IDN:	
Reset:	niScope_init
SelfCal:	
SelfTest:	
ConfigureChanCharacteristics: Channel= ,Impedance= ,Bandwidth=	niScope_ConfigureChanCharacteristics
ConfigureVertical: Channel= ,Coupling= ,Attenuation= ,Range= ,Offset=	niScope_ConfigureVertical
ConfigureHorizontalTiming: SampleRate= ,Position= ,Points=	niScope_ConfigureHorizontalTiming
ConfigureEdgeTrigger: Channel= ,Slope= ,Coupling= ,Level=	niScope_ConfigureTrigger
ConfigureImmediateTrigger:	niScope_Initiate
Commit:	niScope_Commit
Measure: Channel= ,NumberOfAverages= ,Measurement=	niScope_Fetch

Creating the Command Processor

Next, we created a wrapper application in VB.NET®. This allowed us to create an instance of a NI-Scope and control it using text based commands over TCP/IP. Now we can control this scope from any computer on our local network.
Note: While I was testing the code, I could even control the NI-Scope using commands in my web browser.

Sample Code

```
    Public Overrides Function Command(ByVal CMD As String) As String


            If UCase(CMD).Contains("IDN:".ToUpper) Then
               Return myScope.Identity.InstrumentModel
               Exit Function
            End If

            If UCase(CMD).Contains("Reset:".ToUpper) Then
               If Me.Reset() = 0 Then
                  Return "Success"
               Else
                  Return "ERROR!"
               End If
               Exit Function
            End If

            If UCase(CMD).Contains("SelfTest:".ToUpper) Then
               Dim Result As Integer = Me.SelfTest()
               If Result = 0 Then
                  Return "Pass"
               Else
                  Return "Fail! Code -" & Result
               End If

               Exit Function
            End If

            If UCase(CMD).Contains("SelfCal:".ToUpper) Then
               Dim Result As Integer = Me.SelfCal()
               If Result = 0 Then
                  Return "Pass"
               Else
                  Return "Fail! Code -" & Result
               End If

               Exit Function
            End If


            If UCase(CMD).Contains("ConfigureChanCharacteristics:".ToUpper) Then
               If Me.ConfigureChanCharacteristics(CMD) = 0 Then
                  Return "Done"
               Else
                  Return "ERROR!"
               End If
            End If
    If UCase(CMD).Contains("ConfigureVertical:".ToUpper) Then
        If Me.ConfigureVertical(CMD) = 0 Then
            Return "Done"
        Else
            Return "ERROR!"
        End If
    End If

    If UCase(CMD).Contains("ConfigureHorizontalTiming:".ToUpper) Then
        If Me.ConfigureHorizontalTiming(CMD) = 0 Then
            Return "Done"
        Else
            Return "ERROR!"
        End If
    End If

    If UCase(CMD).Contains("ConfigureEdgeTrigger:".ToUpper) Then
        If Me.ConfigureEdgeTrigger(CMD) = 0 Then
            Return "Done"
        Else
            Return "ERROR!"
        End If
    End If

    If UCase(CMD).Contains("ConfigureImmediateTrigger:".ToUpper) Then
        If Me.ConfigureImmediateTrigger() = 0 Then
            Return "Done"
        Else
            Return "ERROR!"
        End If
    End If

    If UCase(CMD).Contains("Commit:".ToUpper) Then
        If Me.Commit() = 0 Then
            Return "Done"
        Else
            Return "ERROR!"
        End If
    End If

    If UCase(CMD).Contains("Measure:".ToUpper) Then
        Dim Result As Double = Me.Measure(CMD)
        Return Result
        Exit Function
    End If


    Return "ERROR! Command Not Processed!"
    Exit Function

End Function
```

Exposing the Command Processor

This was the coolest part of the code. Notice the Overrides in the function call listed below:

```
Public Overrides Function Command(ByVal CMD As String) As String
```

This function overrides the base class function that uses the Operating Contract and WebGet attributes:

```
<OperationContract()>
<WebGet(ResponseFormat:=WebMessageFormat.Xml,
BodyStyle:=WebMessageBodyStyle.Bare)>
Public MustOverride Function Command(ByVal CMD As String) As String
```

This makes creating a web interface for the above command process a as simple as:

```
' Create New host
Dim host = New WebServiceHost(handler, New Uri("http://" & Me.IP & ":" & Me.Port))
Dim EP = host.AddServiceEndpoint(GetType(iTxtCommand), New WebHttpBinding(),
Name)
host.Open()
```

CREATING THE MCNETCOMM.EXE

Now that the code UUT command processing code is written and a service has been enabled, the next thing we need is a link from MET/CAL®. Because we wanted the software to be backward compatible, we chose to write an executable that could exchange data between MET/CAL® using the dosdose.dat file and the DOS FSC.

The McNetComm.exe we designed will support versions of MET/CAL® from 5.0 and up to and including 8.x version. We also created the executable to be COM visible. This will allow new versions of MET/CAL® to access the MSB™ using the LIB FSC in place of the DOS FSC.

The MET/CAL® Procedure

Like all of our procedures, we like to calibrate more with less code, so this procedure supports our standard MET/CAL® programming model with the Test Points Sub calling the Test Routines sub. I want to keep this paper short, so if you have questions on this model, you can read my paper on “Rethinking the Flexible Standards Paradigm” found at http://www.callabsolutions.com/category/papers-articles/.

As we build the communication and control portion of our MET/CAL procedure, we were able to simplify thinks by keeping a set of global variables storing the state of the instrument. Then we could simply call a configure instrument call that would set the UUT up as required for the test.

First, we would call the Default Test Configuration resetting the global variables:

```
3.001  LABEL      Default
# Channel Settings
3.002  MATH       @Channel    = 1
3.003  MATH       @Impedance  = 1e6
3.004  MATH       @Bandwidth  = 100e6
3.005  MATH       @Coupl      = "'DC'"
3.006  MATH       @Atten      = 1
3.007  MATH       @Range      = 4
3.008  MATH       @Offset     = 0
3.016  MATH       @AVG        = 8
# Horizantal Settings
3.009  MATH       @SampleRate = 10e6
3.010  MATH       @Position   = 50
3.011  MATH       @Points     = 100e3
#Trigger Settings
```

```
3.012  MATH       @TChannel   = 1
3.013  MATH       @Slope      = "'POS'"
3.014  MATH       @TCoupl     = "'DC'"
3.015  MATH       @Level      = 0.00125
```

With each test group we would set the Test Channel:

```
3.002  MATH       @Channel    = <Test Channel>
```

And every point we set the required variables and execute the test:

```
#----------------------------
10.005  MATH       @Volts=0.09*1
10.006  MATH       @Range=0.2*1
10.007  VSET       UUT_Res = .001
10.008  IF         Find(S[23],"EnableRepeatability",1)>0
10.009  VSET       U3 = 0
10.010  ENDIF
10.011  CALL       NI 51xx Sub Test Routines-Conf
10.012  MATH       L[9]=Fld(S[31],2,"Unc=")/1
10.013  ACC        0.000%_      L9U
10.014  IF         1==0
10.015  TARGET     -m
10.016  CALL       NI 51xx Sub Test Routines-Meas
10.017  ENDIF
10.018  MATH       MEM=Fld(S[31],2,"Value=")/1
10.019  MEMCX  0.2   %_          0.65U
```

The test routines would configure the UUT using the following Sub Tools Calls:

```
# Set up the Channel
3.023  MATH       S[30]="ConfChanChar"
3.024  CALL       NI 51xx Sub Tools
3.025  MATH       S[30]="ConfVert"
3.026  CALL       NI 51xx Sub Tools
```

The Sub Tools then passes the commands to the UUT as follows:

```
#==================================================================
7.001  LABEL      ConfChanChar
7.002  MATH       MEM2 = "ConfigureChanCharacteristics:"
7.003  MATH       MEM2=MEM2& " Channel= " & @Channel
7.004  MATH       MEM2=MEM2& ",Impedance= " & @Impedance
7.005  MATH       MEM2=MEM2& ",Bandwidth=" & @Bandwidth
7.006  DOS        C:\CLS\McNetComm.exe Query UUT

7.007  IF         Find(MEM2,"Configure",1)
7.008  DISP       Communication Error Command Not Executed
7.009  ENDIF
7.010  END
#==================================================================
8.001  LABEL      ConfVert
8.002  MATH       MEM2 = "ConfigureVertical: "
8.003  MATH       MEM2=MEM2& " Channel=" & @Channel
8.004  MATH       MEM2=MEM2& ",Coupling=" & @Coupl
8.005  MATH       MEM2=MEM2& ",Attenuation=" & @Atten
8.006  MATH       MEM2=MEM2& ",Range= " & @Range
8.007  MATH       MEM2=MEM2& ",Offset= " & @Offset
8.008  DOS        C:\CLS\McNetComm.exe Query UUT
```

```
8.009  IF         Find(MEM2,"Configure",1)
8.010  DISP         Communication Error Command Not Executed
8.011  ENDIF
8.012  END
```

CONCLUSION

So why does all this work? It is not just because it is neat—ironically I write more code so that I can write less code. Most developers and managers don't see the cost for code support and re-writes. We look at our software with respect to support and life expectancy. To be successful as a software development organization in the metrology world, we have to produce quality software solutions that can stand the test of time.

Creating an additional layer to our procedures, the Metrology Service Bus™, will allow greater flexibility. Adding a simple command processer to our software based instruments allows us to control them remotely. In the end as a company we have de-siloed our software, opening up a world of possibilities.