Rethinking the Flexible Standards Paradigm

Speaker/Author: Michael L. Schwartz
Cal Lab Solutions
PO Box 111113, Aurora CO 80042 US
T: 303-317-6670 | F: 303-317-5295
Email: MSchwartz@CalLabSolutions.com

Abstract: In many of today's software projects, developers are challenged with the task of designing interchangeable standards architecture into their metrology based applications. Currently, many developers see an oscilloscope as an oscilloscope and believe that all oscilloscopes are created equal, and are therefore interchangeable; at the same time, any oscilloscope manufacturer will tell you their oscilloscope is different with special features requiring non-standardized sets of commands to implement those special and specific features. Consequently, developers write their code to implement special and unique features in what was designed to be a generic driver. Cal Lab Solutions took a step back to re-evaluate the problem and all the solutions. We came up with a software design methodology that allows the user to incorporate non-standardized features of complex standards while maintaining a highly flexible interchangeable instrumentation model. This paper will demonstrate how a process centric model allows greater flexibility over the generic command centric model.

Overview

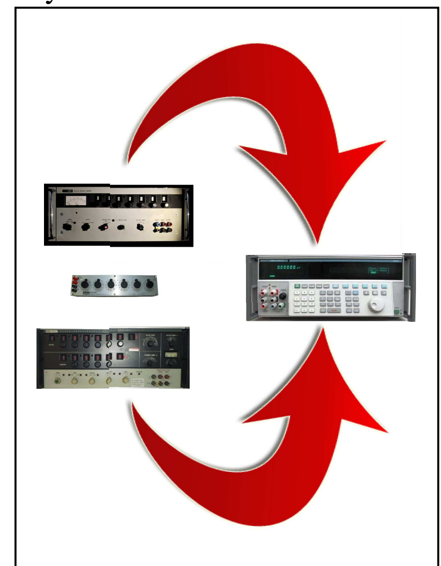
Software design at its core is an abstraction of reality. Software projects succeed or fail based on the architecture and how the developers approach the problem they are designed to solve. A software design based on a solid abstraction is more likely to succeed having an extended lifespan, reusability, and flexibility; whereas a bad abstraction most likely will result in a poor software solution leading to a weak and fragile final product. This paradigm holds true for the metrology related software solutions.

Designing flexible and robust software solutions is no easy task. There are countless hours spent at the drawing board. Ideas are vetted; models are designed, evaluated, and thrown away. Through a lot of trial and error, a solid design appears. And when the design appears, it is so elegant it looks intuitive; and you are left asking yourself why we didn't do this in the first place.

Through this process, we stumbled upon a design model that solves the problem on complex instrumentation, as well as functions cross-platform, by making a slight alteration to the paradigm behind the underling concept of flexible standards.

Background

For years I have attended conference reading papers and watched presentations rehashing the same old interchangeable standards paradigm. Despite dynamic changes in the design of instrumentation, the industry still thinks of equipment and flexibility in terms of equipment classification, as if all instruments fall neatly into some form of generalization. Much of the industry's automation software is based on these assumptions; idiosyncrasies of the equipment classification type are written into the calling code, limiting the flexibility of the software.



The assumption that an instrument will fall into some kind of general classification is flawed; and history demonstrates how problematic this assumption can be. For example, when I came into calibration in the late 1980's we had a DC Voltage Standard, an AC Voltage Standard, and a set of standard resistors. Today, all of those instruments are wrapped up into one instrument called a multi-function calibrator. Looking at the historical evolution of instruments in general, as an instrument peaks in measurement accuracy, manufacturers start adding capabilities and instruments then morph into something else, creating entirely new instruments.

With today's increased computing power, Moore's Law[1] allowed products to morph and hybridize at an ever increasing pace. Today we have oscilloscopes with a built-in arbitrary waveform generator, hand-held digital multimeters that output current and pulses, and even oscilloscopes with fully functional spectrum analyzers built in. And this is just the beginning!

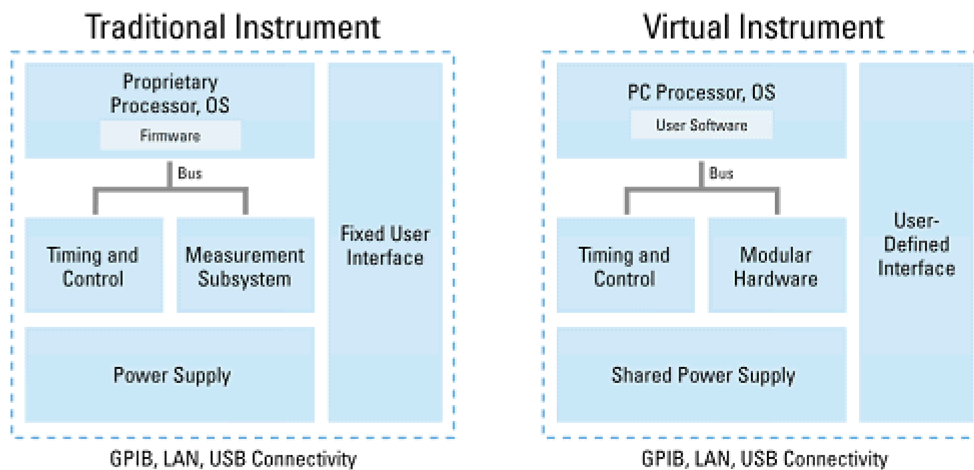
Built-in Complexity

Each time an instrument adds capabilities, it increases its operational complexity. This leads to increased complexity in the instrument's remote operation command structure. This forces developers to write wrappers, patches and Band-Aids adding unwanted complexity to the software.

Another change in complexity we are seeing is in the communication mechanisms. For more than two decades, GPIB has been the dominate medium used for communication. Today, many instruments have several options with it comes to communication, each with its own level of complexity.

Modular Instruments

We are also starting to see a larger number of modular instruments. In the not too distant future, they will become a predominate part of the instrumentation we will be supporting. Tomorrow's instruments will be a conglomeration of the sub-components, assembled in the modular hardware section, and designed to solve a specific set of measurement problems. Manufacturers are already offering built to order configurations of instruments, but a modular architecture gives them much greater flexibility.



Comparison of traditional and virtual instrumentation architectures from a National Instruments' white paper[2]. Both share similar hardware components; the primary difference between the architectures is where the software resides and whether it is user-accessible.

Modular instrumentation provides manufacturers with some very distinct advantages: it allows them to right-size an instrument to the customers' unique requirements; manufacturers are better able to balance cost vs. measurement requirements—where traditional instruments would typically have more features than required for many applications; it allows them to go to market faster, because the underlying hardware is flexible—can be easily configured and reconfigured;

then, user defined interface allows the manufacturer to customize the instrument to the measurement needs.

The impact of this migration to modular instrumentation on metrology will be just as significant, causing some major disruptions in many of the software systems we are currently using. The first major disruption will be that many manufacturers are not investing in the development of a command language to control the instrument. Manufacturers instead rely on software drivers in order to communicate with the instrument; because there is no command language, software solutions designed using a database of commands will no longer work. Developers will then have to create a patch to their software in order to communicate with the manufacturer's drivers.

Where the Model Breaks

The examples cited above demonstrate some weakness in the architectural designs being implemented in several software solutions currently on the market. Despite solving many of today's measurement problems, without extensive rewrites, solutions using the generic command centric model will become more and more difficult to support and maintain as the underlying design principles of instruments changes.

Concerns arise when looking at many of today's flexible standard models. First, as complexity of the command syntax increases, a simple model of inserting a value into a formatted string will be problematic as instruments morph. This model lacks the fundamental flexibility required to adjust for programmatical variations in the instrument's implementation. One prime example of a complex instrument available today is the Agilent E4440A. This instrument has several modes of operation, and thus the complexity of a simple reset now takes several commands and queries.

Another concern is error checking and error handling. When a simple command syntax replacement is being used, there is often no implementation of error handling. Each instrument has drastically different implementations of error handling so it becomes very difficult to compress into simple commands. Furthermore, proper error checking should include, at a minimum, both range checking for measurement validation, in addition to verification the instrument is properly configured with no configuration errors. These errors must also be passed back to the calling environment so they can be handled properly.

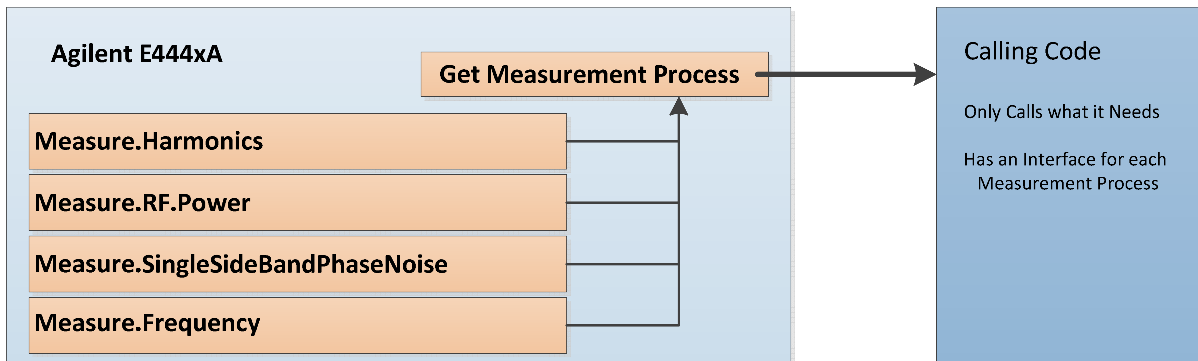
Eventually the command replacement model will become obsolete, as instruments move from a command based control model to a driver based model. As instruments change to modular based instruments, very complex instruments using desktop computer power with Distributed Component Object Models (DCOM) command languages will no longer dominate instrument control. Program control of instruments will become very specific and tightly coupled to the drivers provided by the manufacturer. Solutions on the market today will require a middleware tool or a patch to bridge the incompatibility.

Rethinking the Paradigm

First we need to rethink the concept of instrument interchangeability. With instruments increasingly hybridized to increase their features, the concept of a generic instrument class driver, with commands stored in data, no longer functions. This presented a problem and forced us back to the drawing board, where we threw everything out and rethought the model from scratch. We discovered a driver model that allows us greater flexibility, one that can withstand the changes in technology and hybridization of instrumentation. By understanding and utilizing the principles of Object Oriented Programming (OOP) [3], we broke our software structure down into core reusable pieces of code. When we looked at an instrument from the perspective of a collection of metrology functions, not a device type, we discovered that model was very solid and very flexible[4]. And keeping with OOP, our abstraction matched reality, allowing us to mimic in software what manufacturers were doing in hardware. Because when you think about it, they are just adding measurement capability.

Measurement Process Model™

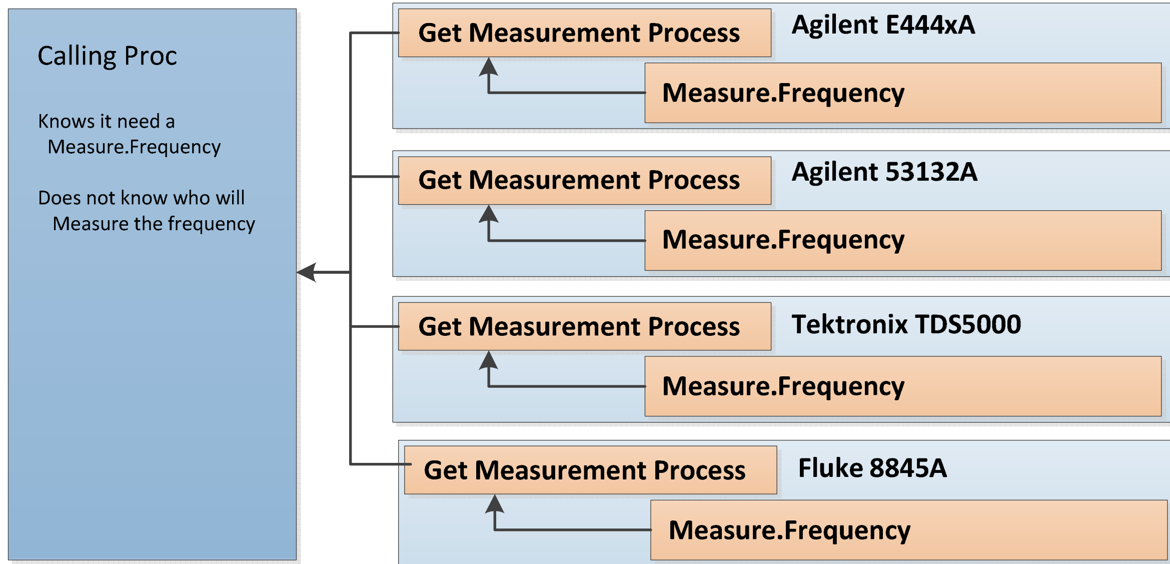
We came up with the Measurement Process Model™, which allows us to create a series of very small drivers for any given instrument and providing a standard methodology of assembling them into a hybridized instrument driver. As shown in the figure below, the Get Measurement Process allows the calling procedure to laterally ask the driver if it supports the measurement functions it needs. If the driver does not support a measurement function, the calling procedure is not able to use that instrument.



The abstraction shows that each of the measurement functions in the driver represents a contract defining the specific operations and interaction between the calling procedure and the instrument driver. The calling procedure explicitly knows how to use the measurement driver though it has no idea of the specific implementation.

When you look at it from the calling code, you can see the power of this new paradigm by changing our focus from an instrument classification basis to a measurement process model. We gain greater flexibility by not limiting standard substitution to a single instruments classification.

Now we can use a wider range of instruments capable of implementing the required measurement process.



The calling procedure has passed complete control of the measurement process to the driver. This provides the greatest flexibility in instrumentation, drivers can now become instrument specific and are able to implement processes that allow them to take full advantage of their specific measurement operations.

Cross Platform Compatibility

In theory, when a concept is sound, it will work in multiple software tools and cross-platform. So far, the implementation has been proven to be very robust in the Microsoft®.Net and Fluke's MET/CAL® platforms.

Microsoft® .Net Implementation

The Microsoft® .Net model proves most flexible, since it is an Object Oriented Programming environment, allowing us to take full advantage of features like interface and inheritance. Microsoft® .Net has a very structured interface which helps the developer fully implement an interface in a driver, taking advantage of the power and flexibility of the Measurement Process Model™.

The sample interface below becomes the contract between the calling code and the implementation in the driver. Notice the interface is very simple and very abstract. This becomes the contract between the calling procedure and the driver. The calling procedure can only call the functions defined in the interface and the driver must implement every one of the functions.

```
Public Interface iDC_Volt_Meter
  Inherits iInstrument
```

```
' Meter Operations
Sub Reset()

' Meter Measurements
Function MeasureDCVolts(ByVal ExpectedValue As Double) As Double

' Instrument Uncertainties
Function GetInstUnc(ByVal Value As Double) As Double

End Interface
```

The driver below must implement the interface per the contract. You can see how the HP 34401A Driver implements several interfaces including the interface listed above.

```
Public Class HP_34401A
  Inherits CLS_DriveBaseClass

  Implements iDC_Volt_Meter
  Implements iAC_Volt_Meter
  Implements iDC_Current_Meter
  Implements iAC_Current_Meter
  Implements i2W_Ohm_Meter
  Implements i4W_Ohm_Meter
…………
  Public Function MeasureDCVolts(ByVal ExpectedValue As Double) _
          As Double Implements iDC_Meter.MeasureDCVolts
    ' Check the Routing Button
    Me.CheckFrontTerm()

    Me.Write("CONF:VOLT:DC AUTO,MIN")
    Me.OPC()

    Return Me.ReadSettled()
  End Function

Public Function GetInstUnc(ByVal Value As Double)
          As Double Implements iDC_Meter.GetInstUnc
    'Uncertainties Based on 1 Year Specifications
    ' Percent of Reading + Percent of Range

    Select Case Value
      Case Is <= 0.1
        '0.0050 %of Reading + 0.0035 %of Range
        Return (Value * 0.00005) + (0.1 * 0.000035)
      Case Is <= 1
        '0.0040 %of Reading + 0.0007 %of Range
        Return (Value * 0.00004) + (1 * 0.000007)
      Case Is <= 10
        '0.0040 %of Reading + 0.0007 %of Range
        Return (Value * 0.00004) + (10 * 0.000007)
      Case Is <= 100
        '0.000045 %of Reading + 0.000006 %of Range
        Return (Value * 0.000045) + (100 * 0.000006)
      Case Is <= 1000
        '0.0045 %of Reading + 0.0010 %of Range
        Return (Value * 0.000045) + (1000 * 0.00001)
      Case Else
        Return 4.99E+39
    End Select
```

Not shown in these programming samples is the code implementing the Get Measurement Process. Microsoft® .Net includes a feature called Reflection, allowing the code to interrogate on object extracting its interfaces. However, in our implementations we have explicitly written the Get Measurement Process function.

Fluke MET/CAL® Implementation

Implementing this Measurement Process Model™ in MET/CAL® is a little more difficult, but not impossible. The Fluke MET/CAL® platform is not an Object Oriented Programming environment, so the programmer will have to pay closer attention to what he or she is doing to insure each driver implement is 100% of the interface. The interface will then have to be defined and maintained in support documentation.

A Sub Procedure in MET/CAL® is a script, so it cannot be instantiated, used and then unloaded. It has limited support for local variables as well as static variables and data storage. But, none of these limitations prevent us from implementing the Measurement Process Model™.

Notice in the MET/CAL® driver below, we are supporting all the key features of the interface above. We have a specific call for the reset command as well as the Measure.Volts.DC. There are only a few specific differences in the implementation. In the VB.Net implementation above, we perform two calls—one for the reading and the other for the uncertainty—whereas in the MET/CAL® implementation, we do it all in a single measurement call and automatically return the uncertainties.

```
Cal Lab Solutions                              MET/CAL Procedure
=============================================================================
INSTRUMENT:          CLSD-Measure.Volts.DC             (34401A Front)
DATE:             2010-12-01 15:24:57
AUTHOR:              Cal Lab Solutions
REVISION:            $Revision: 5 $
ADJUSTMENT THRESHOLD:  70%
NUMBER OF TESTS:      4
NUMBER OF LINES:      127
#============================================================================
 STEP   FSC   RANGE NOMINAL      TOLERANCE     MOD1       MOD2  3  4 CON
  1.001  JMPL        Reset                   (find(S[30], "Reset",1)>0)
  1.002  JMPL        Measure.Volts.DC   (find(S[30], "Measure.Volts.DC",1)>0)

  1.003  DISP        Error Calling the Procedure..
  1.004  END
  1.005  EVAL   CLS

#============================================================================
  2.004  LABEL        Reset
  2.005  IEEE        [@34401]*RST[D299]*OPC?[i!]
  2.006  JMPL         End
  2.007  EVAL   CLS

#============================================================================
  4.001  LABEL        Measure.Volts.DC
# Set the Defaults
# Get the Volts
  4.002  IF        (Find(S[30],"Volts=",1)>0)
  4.003  MATH        L[11]=Sub(S[30],find(S[30],"Volts=",1),1e3)
  4.004  ELSE
  4.005  DISP        Error: Expected Voltage required.
  4.005  DISP        CLSD-Measure.Volts.DC (34401A Front)
  4.006  MATH        S[31]="Value= 99e39  Unc= 0";MEM=99e39
  4.007  END
```

```
  4.008  ENDIF

# Check Range
  4.009  IF         L[11]>1.1e3
  4.010  DISP        Error: Voltage exceeds meter limits.
  4.010  DISP        CLSD-Measure.Volts.DC (34401A Front)
  4.011  MATH          S[31]="Value= 99e39  Unc= 0";MEM=99e39
  4.012  END
  4.013  ENDIF

# Check the Input Terminals
#======================================
  4.014  LABEL       SetInput
  4.015  IEEE        [@34401]ROUT:TERM?[I$]
  4.016  IF         ZCMPI(MEM2, "REAR")
  4.017  DISP        Set the 34401A Front\Rear Input to Front
  4.018  JMPL        SetInput
  4.019  ENDIF

# Configure the Input
#======================================
  4.020  IEEE        [@34401]CONF:VOLT:DC [L11],MIN;*OPC?[i!]

# Settle the Reading
  4.021  IEEE        [@34401]READ?[I]
  4.022  IEEE        [@34401]READ?[I]
  4.023  IEEE        [@34401]READ?[I]

  4.024  IF         abs(MEM)>1e30
# If overranged then go to AutoRange
  4.025  IEEE        [@34401]CONF:VOLT:DC AUTO,MIN;*OPC?[i!]
  4.026  IEEE        [@34401]READ?[I]
  4.027  IEEE        [@34401]READ?[I]
  4.028  IEEE        [@34401]READ?[I]
  4.029  ENDIF

# Set Uncertainties
#================================
  4.030  LABEL       SetTol
  4.031  MATH        L[1]=ACCV("HP 34401A","Volts", MEM)
  4.032  MATH        S[31]="Value= "& MEM &" Unc= "&L[1]
  4.033  MATH        S[31]=S[31]&"Volts= "& MEM &" VoltsUnc= "&L[1]

…………………………………………..

#================================
  4.049  LABEL       End
  4.050  END
```

Note that our MET/CAL® implementation only implements a single measurement process at a time. We did this because scripting languages become very cumbersome to debug as they increase in complexity. Also notice at the top of the procedure we are at revision 5, meaning it only took five edits to write, test, and fully debug this code.

Conclusion

Though it seems obvious and appears to be a very simple course correction, in hindsight the Measurement Process Model™ presents a very innovative approach to solving the flexible standards problem. The fundamental underlying concept is simple: write the code you need as you need it, and then add it to the instrument driver after testing. We've shown fallibilities of the instrument classification driver model and how an alternative method can make our code run more efficiently.

History has shown us the natural evolution of hardware; how manufacturers will continue to add measurement functionality to gain a competitive edge. Newer computers, modular instruments, and communication innovations will repeatedly challenge our implementations and software solutions. We can choose to patch them each time or simply rethink the paradigm. In the end, the instrumentation is changing, and software solutions will have to change to keep pace.

References

1. "Moore's Law and Intel Innovation," <http://www.intel.com/about/companyinfo/museum/exhibits/moore.htm > (accessed March 12, 2012).
2. "Understanding a Modular Instrumentation System for Automated Test," National Instruments, <http://zone.ni.com/devzone/cda/tut/p/id/4426> (accessed March 12, 2012).
3. Shalloway, Alan and Trott, James, *Design Patterns Explained*, 2 ed. (Addison-Wesley, 2005).
4. Michael Schwartz, "Using Fluke MET/CAL® to Implement a Flexible Measurement Driver Model with Expanded Measurement Uncertainties, Error Checking and Standard Flexibility," NCSLI 2012 Workshop & Symposium, National Harbor, MD, August 21-25, 2012.