The Pennsylvania State University

The Graduate School

College of Engineering

A SIGNAL-BASED APPROACH TO AN INSTRUMENT DRIVER SYSTEM

A Thesis in

Computer Science and Engineering

by

Katherine Barbara Kilroy

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

December 2008

The thesis of Katherine Barbara Kilroy was reviewed and approved* by the following:

*Signatures are on file in the Graduate School

ABSTRACT

Although automated test equipment standards have improved substantially since as recently as the early 1990s, the standards have not reached the point where instrument models are interchangeable. This may never be achievable, which is why a software solution is necessary in order to allow for instrument interchangeability.

This paper introduces an instrument driver system that can solve the problem of instrument interchangeability for those companies and organizations that have test programs that last for many years or test stations with dozens of test programs. Although there is a specification for driver systems, called the Interchangeable Virtual Instrument Measurement and Stimulus Subsystems (IVI-MSS) specification, it does not describe implementation details or issues. This paper introduces guidelines and recommendations for implementation details for a driver system, but it does not replace IVI-MSS or any other high-level specification. The driver system that is introduced focuses on the signals to be sourced or measured rather than specific instrument settings. It also builds on current driver technology, including IVI and VXI*plug&play* driver standards.

To prove the validity of such a system, an oscilloscope driver class was created and tested for three different oscilloscope models by two manufacturers. The code was written in Microsoft Visual C# .NET® for use on a test station that uses Microsoft Windows® XP on the controlling computer. The driver system successfully took the same three measurements with all three oscilloscope models using the same driver calls, thereby proving that such a system can work even with discrepant underlying specific drivers.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

In the modern world of automated test and measurement equipment (henceforth called instruments), engineers must write test programs that can control those instruments as accurately, efficiently, and thoroughly as possible. The way they do this is through the use of instrument drivers. Commercial drivers have come a long way from the days of “free-for-all”, non-standardized drivers to the current state of the art, which are standardized Interchangeable Virtual Instrument (IVI) drivers. Even with IVI drivers, though, there is still room for improvement. This is because one of the biggest problems with instrument drivers is the difficulty in replacing instruments without having to rewrite the test programs that use those instruments.

While IVI drivers address some of the problems associated with instrument replacement, there are still lingering issues that need to be addressed. Despite the fact that the word “interchangeable” is used in the name of the specification, it is still not possible to guarantee interchangeability using IVI drivers alone. One of the IVI specifications, Measurement and Stimulus Subsystems, proposes additional layers between an application program (test program) and the instruments it uses. This specification is more complicated than what is required for many automated test systems; for one, the specification dictates that interchangeability extend beyond the class of the instrument. For example, if the test program takes a voltage measurement, the user application should not care whether it’s taken with a digital multimeter (DMM) or an oscilloscope. Therefore, the intermediate layer should allow for substitution of a DMM where an oscilloscope had previously been used.

This thesis will present an efficient instrument driver system that will allow for the substitution of instruments in the same family without the need to change, recompile, or rebuild any of the code in the test programs that use the driver system. This will be accomplished by focusing on the signals and connections that the test program requires rather than the specific settings for the instrument that will be providing the signal, connection, or measurement. With a system such as this in place, companies could potentially save thousands of hours and hundreds of thousands of dollars whenever an instrument they were using goes obsolete.

Chapter 2: Prior Work (Current State)

2.1 History of Instrument Drivers

Although IEEE standard 488.1 standardized the physical and electrical specifications for General Purpose Interface Bus (GPIB[i]) instruments, it wasn't until the introduction of IEEE standard 488.2 in 1987[1] that there was any standardization for the command structure used by the instruments. Driver commands and formats varied between manufacturers and could even vary between different models built by the same manufacturer. This created a larger learning curve for the engineers who needed to write test programs for the instruments and meant that most test programs would have to be re-written whenever an instrument was replaced with a different model. IEEE 488.2 introduced standard command words for important instrument functions and instructions. For instance, the standard dictated that "*RST" would cause an instrument to perform a reset, "*IDN?" would cause the instrument to return its identity in a predefined format, and "*TST?" would cause the instrument to perform a self-test and then return the status of the self-test.

Standard Commands for Programmable Instruments (SCPI)

Although IEEE 488.2 created standards for many commands, it mandated only a few of the commands and left many design and implementation decisions up to the device manufacturers. Additionally, it was originally essentially an extension of IEEE

[i] GPIB is the more common name for IEEE-488. It is widely used in the test and measurement industry. More information about the mechanical and electrical characteristics is available at http://standards.ieee.org/reading/ieee/std_public/description/im/488.1-1987_desc.html

488.1, the GPIB hardware standard. This meant that manufacturers of VXI or RS-232 based instruments, for example, still did not have a standard to follow[2].

To combat these problems, the Standard Commands for Programmable Instruments (SCPI) Consortium was founded in 1990[3]. The SCPI standards were built on IEEE 488.2, which means that SCPI-compliant devices are automatically IEEE 488.2-compliant, as well.[4] To be a SCPI compliant device, an instrument would have to implement the core set of SCPI commands and not violate any of the SCPI rules.[5] In addition, the instrument would have to use the SCPI-defined commands for any functions the manufacturer chose to implement. In other words, if the instrument had the capability to measure DC voltage, it would have to understand the command structure "MEASure:VOLTage:DC?" and not "MEAS:DC:Voltage?" or some other variation of the standard command structure. Although SCPI defined the standard architecture and functionality of instrument I/O messages, it was not incorporated by most of the instrument models available today[6].

VXI*plug&play*

In 1993, not long after the SCPI Consortium was established, the VXI*plug&play* Systems Alliance[ii] was founded. The major goal of the alliance was to establish system software architecture standards for VXI[iii] instruments[7]. Because VXI instruments are controlled entirely through software, the alliance felt that an open, standard architecture for VXI systems and VXI instrument drivers was critical to the continued success and further adoption of VXI instruments.

[ii] In 2003, the VXI*plug&play* Systems Alliance merged with the IVI Foundation. As a result, the VXI*plug&play* Systems Alliance no longer exists.

Although the target for the standard was VXI instruments, VXI*plug&play* driver standards were quickly adopted by most of the major instrument manufacturers and software suppliers for GPIB instruments, as well. This may have been partly because, at that point, the VXI*plug&play* standard was the only high-level instrument driver standard. It could also be because the alliance created a standard library for instrument communication, called the Virtual Instrument Software Architecture (VISA). All VXI*plug&play* drivers are required to use this, and only this, library for instrument I/O, which is freely available for anyone to download and install. This requirement makes it much easier for end users to incorporate VXI*plug&play* drivers into their test programs partly because the library defines all data types in a standard way.

VXI*plug&play* drivers take the format "abxyz_*command*", where "ab" represents a two-character code for the instrument manufacturer, "xyz" represents an abbreviation of unspecified length for the instrument model, and "*command*" represents something like "init", "reset", "measure", or another driver function. The instrument manufacturer code must be registered with the IVI Foundation, thereby guaranteeing that no two drivers will have the same name. As an example, the VXI*plug&play* driver call to initialize communications with a Tektronix TDS5104B oscilloscope is

```
tktds5k_init(ViRsrc resourceName, ViBoolean IDQuery,
ViBoolean resetDevice, ViSession *instrumentHandle)
```

Similarly, for the Agilent (formerly Hewlett-Packard) 54831B oscilloscope, the function call is

```
hp548xx_init(ViRsrc resourceName, ViBoolean IDQuery,
ViBoolean resetDevice, ViSession *instrumentHandle)
```

[iii] VXI, which stands for **V**ME e**X**tensions for **I**nstrumentation, is a standard bus architecture for test instruments. More information is available at http://www.vxibus.org/

Beyond the standard initialize, reset, self-test, and close functions, however, the *command* can vary greatly. This means that it is still necessary to refer to the driver documentation frequently in order to find the correct function and the function signature. For example, in order to set the vertical scaling and offset for the TDS5104B oscilloscope, a programmer would call

```
tktds5k_SetVerticalScale (ViSession instrumentHandle,
ViInt32 channel, ViReal64 scale)
```

and

```
tktds5k_SetVerticalOffset (ViSession
instrumentHandle, ViInt32 channel, ViReal64 offset)
```

For the HP54831B oscilloscope, though, the programmer would call

```
hp548xx_channelScale(ViSession instrumentHandle,
ViInt16 channel_number, ViReal64 rangevalue,
ViReal64 offsetvalue)
```

Differences such as these are common between instrument drivers and indicate the level of variance between them. This left programmers with the same problems they faced before IEEE 488.2 and SCPI – a lack of standardization for instrument driver calls. Now, instead of differences at the hardware and firmware level, the differences were primarily in the software. Regardless of where the differences lie, however, the end result is the same – a higher learning curve and major rewrites when switching to a new instrument. VXI*plug&play* drivers are also strongly instrument configuration-based rather than signal-based, which is partly why there is so much variation in possible function names. Lastly, because SCPI was not a widely adopted standard and because VXI instrument drivers can be register-based instead of message-based, the low-level I/O calls inside the instrument drivers are usually specific to the instrument for which they were written. This means that interchangeability is not possible with most VXI*plug&play* drivers.

Interchangeable Virtual Instruments (IVI)

In order to address these latest problems, the Interchangeable Virtual Instruments (IVI) Consortium was formed in 1998 by a group of instrument vendors and end users. The goal of this group was to create specifications that would promote high quality, high performance instrument drivers and instrument interchangeability. At this point, the consortium has nineteen active specifications under the headings of “Architecture” and “Instrument Classes”.

Currently, there are eleven IVI architecture specifications. These specifications are independent of the type of instrument that is being programmed. For example, specification IVI-3.1, the Driver Architecture Specification, gives an overview of IVI driver requirements and describes all of the required and optional behaviors of an IVI-compliant driver. It also explains, among other things, the different types of IVI drivers and their intended usage. These types include IVI class drivers, IVI class-compliant specific drivers, and IVI custom specific drivers. These various types will be discussed in more detail later.

In addition to the architecture group, the consortium created specifications for instrument classes. These eight specifications describe the base and extended functionality for each of eight types of instruments – oscilloscopes, digital multimeters, function generators / arbitrary waveform generators, DC power supplies, switching matrices, power meters, spectrum analyzers, and RF signal generators. IVI-4.1, for example, is the IviScope Class Specification, which defines the class capabilities, base capability group, and optional extension groups for oscilloscope drivers. To aid in their main goal of supporting interchangeability, the consortium created a goal of covering

95% of the models in a given instrument class in their base capability group specifications[8]. This means that they didn't include functionality that is not present in at least 95% of the models currently on the market for any given instrument class. For the IviScope class, the base capabilities include the attributes and functions that contribute to the configuration of the channel, acquisition, and trigger subsystems. For options that don't make it into the base capability group, there are extension groups. For example, because there are so many types of triggers available, the base capability group for oscilloscopes includes only an edge trigger type; in order to use a runt trigger, the IVI driver must incorporate the IviScopeRuntTrigger extension group.

Types of IVI Drivers

There are three types of IVI drivers, each with different levels of interchangeability. From the most interchangeable to the least, they are IVI class drivers, IVI class-compliant specific drivers, and IVI custom specific drivers. For each of these driver types, the specifications describe both an Ansi C application programming interface (API), called an IVI-C driver, and a COM API, called an IVI-COM driver. IVI class drivers are provided by the IVI Foundation and contain all of the base and extended capabilities for the class. The prefix of the function calls for an IVI-C class driver is the IVI class name; for example, the function call to configure common characteristics of a scope channel is

```
IviScope_ConfigureChannel (…)
```

By using a completely generic function call, the test program is not tied to a specific driver library.

In order to control the hardware, however, there must be another level. Although the IVI class specifications describe the functions and attributes for the instrument class, they do not describe the low-level driver calls. Therefore, where one instrument might expect to see “TRIGger:A:” as the prefix for any commands regarding the main trigger, another instrument might expect to see “TRIGger:MAIn:”. Specifying the correct low level string or register manipulation (in the case of some VXI instruments) is the job of an IVI class-compliant specific driver. As the name implies, the driver must be written to comply with an IVI class specification. The class-compliant specific driver complies with the class specification by exporting the API described by the class specification. The difference is that the class name prefix is replaced with the specific driver prefix. Using the TDS5104B IVI class-compliant specific driver as an example, the call to configure common characteristics of a scope channel is

```
tkds5000_ConfigureChannel (…)
```

Because it is a *specific* driver, though, it contains the underlying low-level instrument calls required to control a *specific* instrument model, which typically makes it incompatible with other instrument models. For this reason, in order to allow for interchangeability, the test program must not make direct calls to the specific driver.

The last type of IVI driver is an IVI custom specific driver. This type of driver does not conform to any of the IVI class specifications. A custom driver might be written for an instrument for which there is no class specification, such as a counter / timer, but the driver author still wants the other benefits of the IVI architecture. These drivers do not allow for hardware interchangeability.

IVI drivers have additional features that make them superior to standard VXI*plug&play* drivers. In order to use IVI drivers, the end user must install the IVI

Shared Components, which are available on the IVI Foundation web site. This package includes all the files necessary for using the class drivers and interacting with the IVI configuration store, which is an XML document that contains all of the information about the instrument models and drivers in the system. In this document, the user can associate a “logical name” with a particular IVI driver. For instance, the logical name “SCOPE” could be set up to refer to the tkds5000 IVI-C class-compliant specific driver. The specific driver would then be tied to the specific piece of hardware (TDS5104B). The following is a snippet of an entry for the tkds5000 driver, edited for length.

```
- <IviSoftwareModule id="p308">
    <Name>tkds5000</Name>
    <Description>Tektronix TDS 5000 Series Instrument Driver</Description>
    <ModulePath>tkds5000_32.dll</ModulePath>
    <Prefix>tkds5000</Prefix>
    <ProgID />
  <SupportedInstrumentModels>TDS5052,TDS5054,TDS5104</SupportedInstru
mentModels>
- <PhysicalNames>
- <IviPhysicalName id="p311">
  <Name>CH1</Name>
  <RCName>Channel</RCName>
  <PhysicalNames />
  </IviPhysicalName>
- <IviPhysicalName id="p312">
  <Name>CH2</Name>
  <RCName>Channel</RCName>
  <PhysicalNames />
  </IviPhysicalName>
- <IviPhysicalName id="p313">
  <Name>CH3</Name>
  <RCName>Channel</RCName>
  <PhysicalNames />
  </IviPhysicalName>
- <IviPhysicalName id="p314">
  <Name>CH4</Name>
  <RCName>Channel</RCName>
  <PhysicalNames />
  </IviPhysicalName>
  </PhysicalNames>
- <PublishedAPIs>
  <IviPublishedAPI idref="p5" />
```

```
</PublishedAPIs>
<AssemblyQualifiedClassName />
<ModulePath64 />
</IviSoftwareModule>
```

In the preceding example, the tag “`ModulePath`” was followed by “tkds5000_32.dll” and the tag “`Prefix`” was followed by “tkds5000”. This is how the IVI class driver knows which specific driver to use and what functions to call in the specific driver. After loading the correct specific driver, the class driver substitutes the value of the `Prefix` tag for the class name. For example, if the test program calls “`IviScope_ConfigureChannel(…)`”, the IVI scope class driver will, in turn, call “`tkds5000_ConfigureChannel(…)`” in the specific driver.

In the hardware asset section, the TDS5104B’s I/O resource descriptor (address) would be specified. The following is a snippet from the hardware asset section of the IVI configuration store for a TDS5104B:

```
- <IviHardwareAsset id="p342">
   <Name>TDS5104B</Name>
   <Description>Tektronix TDS5104B Oscilloscope</Description>
   <DataComponents />
   <IOResourceDescriptor>GPIB0::6::INSTR</IOResourceDescriptor>
  </IviHardwareAsset>
```

In this case, the TDS5104B uses the instrument address “GPIB0::6::INSTR”. When initializing the instrument session in the test program code, the programmer does not have to specify the address of the instrument. This has the added benefit of allowing the test program to be installed and run on a test station where the instruments were assigned addresses that do not match those of the development station. Also, if the end user wanted to use a different instrument model, the IVI configuration store would be edited to have the logical name “SCOPE” point to the appropriate specific IVI driver, which in turn is associated with the new hardware asset. This is how the IVI system allows for instrument substitutions without having to modify the test program in any way.

2.2 Current Software Architecture

As has been shown, it's generally difficult to replace an instrument without modifying the test program. One possible exception is replacing an instrument with a different model within the same model family that uses the same driver – for instance, replacing a Tektronix TDS754D with a TDS784D. Other than this case, however, either an IVI class driver or an abstraction layer would be required to allow for substitution. Unfortunately, neither IVI class drivers nor simple abstraction layers can guarantee interchangeability.

In some cases, instrument replacement is not an issue. One example is a production environment in which very few test programs are developed for a particular test station. In this case, it would most likely be more cost effective to rewrite the existing test programs than it would to develop a comprehensive instrument driver system to allow for instrument replacement without test program modification. In addition, in industries where the products being tested have a relatively short life cycle, such as consumer electronics, the measurement instruments often last longer than the products. In this case, by the time the instrument goes obsolete and must be replaced, the products might no longer be tested and the test programs will no longer be needed. In these cases, the instrument driver system proposed in this paper would not be cost effective.

On the other hand, for a test station with test programs that number in the hundreds, an instrument driver system will certainly cost less in the long run than re-writing and re-testing all of the test programs that used the replaced instrument models. It is also sometimes very difficult to obtain the products to be tested at the time that the instrument is being replaced, which makes it even more difficult to re-write and re-release the modified test programs. Also, in an industry where the products being tested have a very long life cycle, the test instrument models are likely to go obsolete at some point during the lifetime of the test programs; some instrument types might even have to be replaced multiple times.

One solution to this problem is to apply a simple abstraction layer. A driver abstraction layer is a dynamic link library with exported function calls that acts as an interface between the test program and the various specific VXI*plug&play* instrument drivers. It is essentially a pass-through layer between the test program and the specific driver calls for the instrument. This allows the test program to make “generic” instrument driver calls. For example, instead of calling the VXI*plug&play* driver function

```
tktds5k_init(…)
```

to initialize communications with the TDS5104B scope, the test program calls

```
Scope_init(…)
```

This leaves it up to the abstraction layer to determine which scope model is in use and then make the appropriate specific driver calls. A simple abstraction layer removes the need to change function calls or link to a different library, but it does not guarantee interchangeability.

With a simple abstraction layer, the main reason that direct replacement might not be possible is the hardware and firmware. Test program developers have a tendency to rely on default settings of the development instrument model. They issue an instrument reset and then change only the settings that are required to get a signal to appear correctly on the instrument model that is being used for development. If they don’t have the time or resources available to try the code with all of the instrument models, the developers might fail to find problems with the settings until after the test program is released and in use on the production floor.

For example, the Tektronix TDS5104B was a suggested replacement oscilloscope for the discontinued Tektronix TDS654C. Although the TDS5104B primarily contains a superset of the functionality of the TDS654C, Tektronix made a major change to the horizontal time base – instead of using the 1-2-5 sequence of the TDS654C, the TDS5104B uses a 1-2-4 sequence. This means, for example, that while the time-per-division options for the TDS654C in the nanosecond range include 100 ns, 200 ns, and 500 ns, the options for the TDS5104B are 100 ns, 200 ns, and 400 ns. If the programmer wished to verify the period of a 3 us square wave using the TDS5104B during development, he might issue a call to set the seconds per division to 300 ns (3 us / 10 horizontal divisions = 300 ns). With rounding, the TDS5104B would set the time base to 400 ns / division, but the TDS654C would be set to 200 ns / division. If the test program were to be run on a test station with a TDS654C scope, the signal would not be captured fully and the measurement would fail.

Another, more subtle, potential pitfall is the difference in time that it takes for different instrument models to complete various commands. For example, some scopes and their drivers complete commands faster than others, so if a test were to be developed on a slow scope, then later run with a faster scope, the signal might not be available in time to take a measurement or to issue a particular setup call. In this case, a simple driver abstraction layer is insufficient – a study of the characteristics of the underlying driver calls and the behavior of the hardware is required.

Lastly, with a simple abstraction layer, there could be missed opportunities for instrument interchangeability. For instance, two different instrument models might be capable of taking the same measurement or triggering off of a certain condition, but it might not be straightforward to configure either one of the instruments via a single abstraction layer call. Additional logic and multiple function calls might be necessary in order to configure both instruments to get the same functionality.

Even IVI drivers cannot guarantee interchangeability. Besides the difference in execution time of the various function calls mentioned above, the specification allows for custom specific drivers, which are not interchangeable. Even if the test programs utilize a class-compliant driver, if even one instrument model does not have an IVI class-compliant specific driver, as is the case for the TDS654C, the test program won't work for that instrument. Lastly, if the test program uses an instrument that isn't covered by one of the eight IVI class specifications, such as a counter / timer, then it's not possible to use an IVI class driver because none exists. An IVI custom specific driver could be developed, but as previously explained, these do not allow for interchangeability.

Additionally, even with the extension groups, IVI class drivers do not cover some fairly typical measurements. For instance, one of the more common measurements is a delay, or time interval, measurement. This measurement involves two channels and measures the delay between one event on one channel and another event on the other channel. For example, the test program might need to measure the delay between one signal transitioning to a high logic level and another signal transitioning to a low logic level. Unfortunately, the IVI scope class specification does not address measurements involving more than one channel, so this sort of measurement is impossible with only an

IVI class driver. An IVI specific driver could implement this measurement functionality if the hardware supports the measurement, but again, the instrument would no longer be interchangeable.

IVI Measurement and Stimulus Subsystems (IVI-MSS)

The IVI Foundation addressed these issues with IVI-3.10 Measurement and Stimulus Subsystems Specification. This specification is actually excessive for most end users. As previously mentioned, one of the features of an IVI-MSS solution is the ability to substitute an instrument of a different type in order to take a certain measurement. For example, even if the test were developed with a DMM taking a voltage measurement, the IVI-MSS solution would allow for the substitution of an oscilloscope in place of the DMM as long as the scope is capable of taking the voltage measurement. This feature is completely unnecessary and even virtually unworkable in many automated test stations. First of all, test specification documents always provide tolerance levels for signal measurements, which are used to do accuracy analysis based on the measurement capabilities of the target test instrument. If the analysis assumes that the DMM will be used to take the measurement, there is no way to guarantee that the oscilloscope will also meet the accuracy requirements. Secondly, in automated test stations, the signal routing is somewhat predefined by the configuration of the switching matrices. In many cases, the signal would have to take an entirely different route in order to be connected to a different instrument type, which could cause different loading conditions on the signal and adversely affect the measurement. Allowing for the substitution of different instrument types is therefore not necessarily worth the expense, nor is it always feasible.

The IVI-MSS specification refers to software modules called "Role Control Modules", or RCMs, that contain the code with the specific calls to the instruments. Because of the mandated abilities of the IVI-MSS solution to swap instrument types or group instruments together into virtual instruments, these RCMs are so-named because they are not supposed to be specific to a particular instrument type, but rather, they are supposed to perform a particular "role". The following picture was taken from the IVI-MSS specification and illustrates the relationship of the RCMs to the instruments and the rest of the IVI-MSS solution and user application:

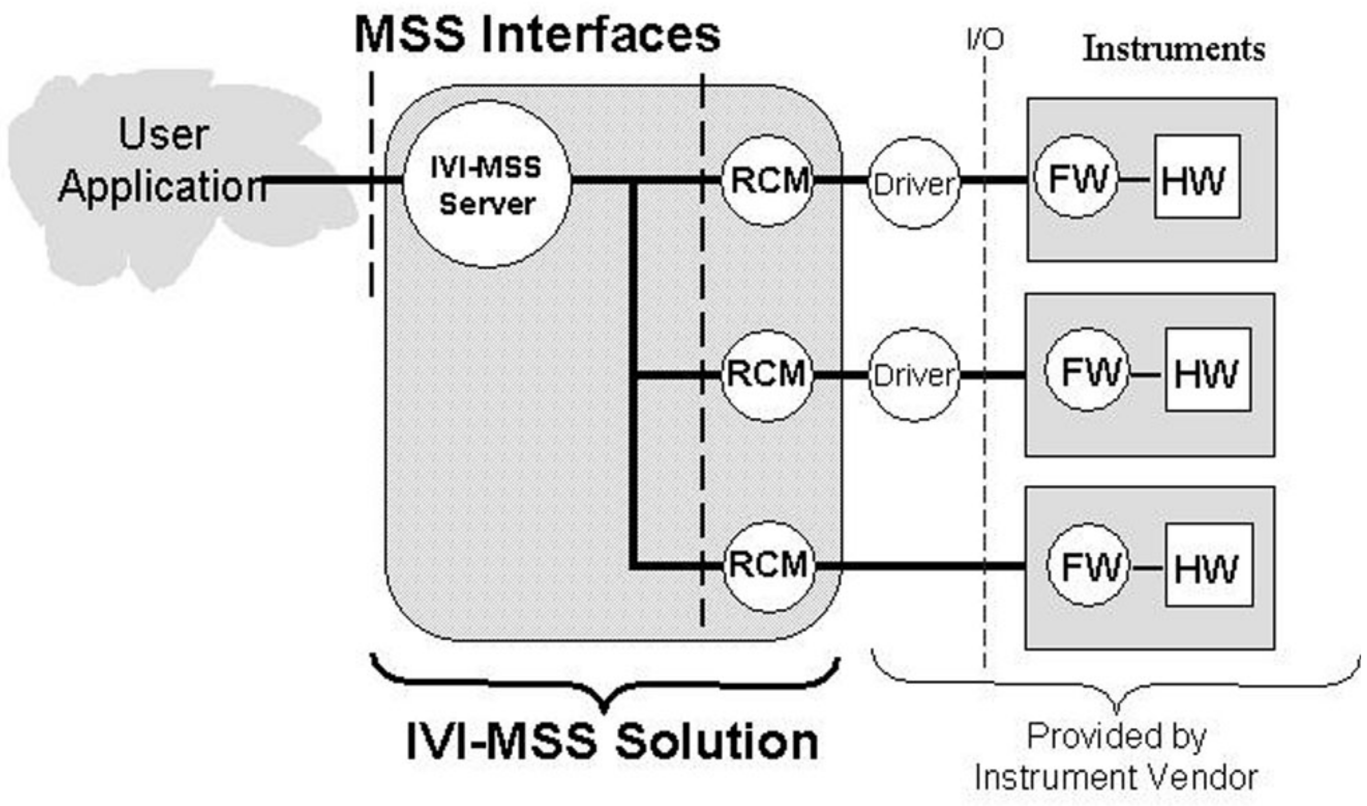


Figure 1: Example IVI-MSS Solution. Source: IVI-3.10 Measurement and Stimulus Subsystems Specification, page 10. Copyright IVI Foundation.

The IVI-MSS server is the part of the solution that controls the interface with the test program and communicates with the RCMs in order to initiate and query the results of measurements. The RCMs are the replaceable units in an IVI-MSS solution, so if an instrument were to be replaced, the RCM would probably be rewritten. One key to interchangeability is that the RCM interface to the IVI-MSS server would have to remain the same.

Another feature that the IVI-MSS specification allows for is the creation of a “virtual instrument” via the grouping of multiple instruments in order to take a single measurement. Though it’s not shown in Figure 1, this would be accomplished by having one RCM control multiple instruments while providing a single interface to the IVI-MSS server. Virtual instrumentation is a special case that is probably not necessary in most automated test systems. For many end users, this is not an issue they need to consider when designing an instrument driver system.

Chapter 3: Design (Proposed State)

One of the reasons that simple abstraction layers and traditional (non-IVI) instrument drivers do a poor job when it comes to interchangeability is that they are too focused on instrument commands and settings. This focus on the instrument is misplaced – the emphasis should be placed on the signal to be measured or sourced. Changing the focus of the abstraction layer from the instrument to the signal is an excellent first step in creating a greater level of separation between the test program and a specific instrument.

The idea of signal-oriented drivers is not original. In fact, SCPI measurement commands were designed to be signal-oriented. Consider this quote from the SCPI consortium web site:

> "The commands in the measure family are "**signal oriented**" in the sense that they specify a measurement in terms of the desired result and in terms of the signal to be measured. A user can specify characteristics of the signal measurement, such as expected signal value or the resolution of the measurement, by adding parameters to the command. Based upon these parameters, the instrument selects suitable settings."[9]

The idea was that the test program could send a command to take a frequency measurement, for example, and the instrument could configure itself with the appropriate settings, take the measurement, and report it to the calling program. For instance, the command ":MEASure:FREQuency? 5000, 1.0" would tell the instrument to configure itself to measure a 5 kHz signal with 1 Hz of resolution, take the frequency measurement, and report the measured frequency to the calling program.

Unfortunately, this signal-oriented philosophy was lost when the VXI*plug&play* standard was developed. Although it was a great help to programmers in that it drastically reduced the learning curve for determining the proper driver calls and syntax, the new driver standard put the focus on the syntax instead of the functionality of the instrument.

IVI drivers brought some of the signal-oriented philosophy back while still maintaining rules regarding driver syntax.

Because IVI drivers have already been developed for most of the instruments on the target test station, it makes sense to use them whenever possible in the proposed instrument driver system. With IVI class drivers, much of the basic programming work has been done already, freeing up resources to create the instrument driver system framework and the other methods and properties needed to track the states of the instruments.

The target test station for the new instrument driver system is a Teradyne Spectrum 9100 with multiple commercial-off-the-shelf (COTS) automated test and measurement instruments. The current standard station configuration includes a Tektronix TDS5104B oscilloscope, but other oscilloscopes have been used in the past and are present in test stations that have been delivered around the globe. These other oscilloscope models are a Tektronix 654C and an Agilent 54831B.

The driver system should establish defaults for all instrument settings that are applied when the instrument is reset. That way, all of the models will be starting from the same point, which takes the burden off of the test program developer. In order to ensure that all of the instrument models will work equally well, the defaults should be set to the least capable setting out of all of the currently used models, if there is more than one model. In the case of the target test station, the only instrument for which there are multiple models currently in use is the oscilloscope. For all of the other instruments in the station, the reset state should be based on the capabilities of the currently used models. This should be a safe starting point because any time an instrument goes obsolete and must be replaced with a new model, the new model should be at least as capable as the model it is replacing. This could involve some lengthy investigation, but it is critical to viable instrument replacement.

It must be assumed that while the test program is running, there will be no other source altering the instrument settings. In other words, there should not be any other running processes that might change a setting on an instrument, nor should an operator be changing settings. In the case that another source will have to change settings, there should be a mechanism for querying the new settings of the instrument and setting them back to their state prior to the interference, if necessary. By not allowing another source to alter the instrument state, time can be saved during program execution because the driver system will not have to check the instrument state constantly. Instrument input / output (I/O) is generally a much bigger source of delay than pure software routines in drivers, so minimizing the amount of I/O should be a priority.

While the amount of I/O should be minimized, the driver system may have to introduce delays in other areas. It should always err on the side of caution when there is a possibility that an instrument setup call might take longer than it takes for the software to issue the next instrument setup or measurement call. One way to handle this would be to study the amount of time it takes for a particular model to perform certain setup calls. One example of this is the case where the test program is asking an oscilloscope for a measurement. There is a correlation between the horizontal time base, the record length, and the amount of time it will take to acquire an entire waveform. Therefore, if the driver call to initiate an acquisition is non-blocking, meaning that it will return control to the calling program before the acquisition is complete, then attempting to query the instrument for the measurement right away could cause an error condition. The driver system should be aware of conditions such as the time base, the record length, and other related settings in order to determine when the acquisition will be completed. Even better, if the instrument supports it, the driver system should query the acquisition status to determine if it has completed or if the program needs to wait longer.

Chapter 4: Implementation

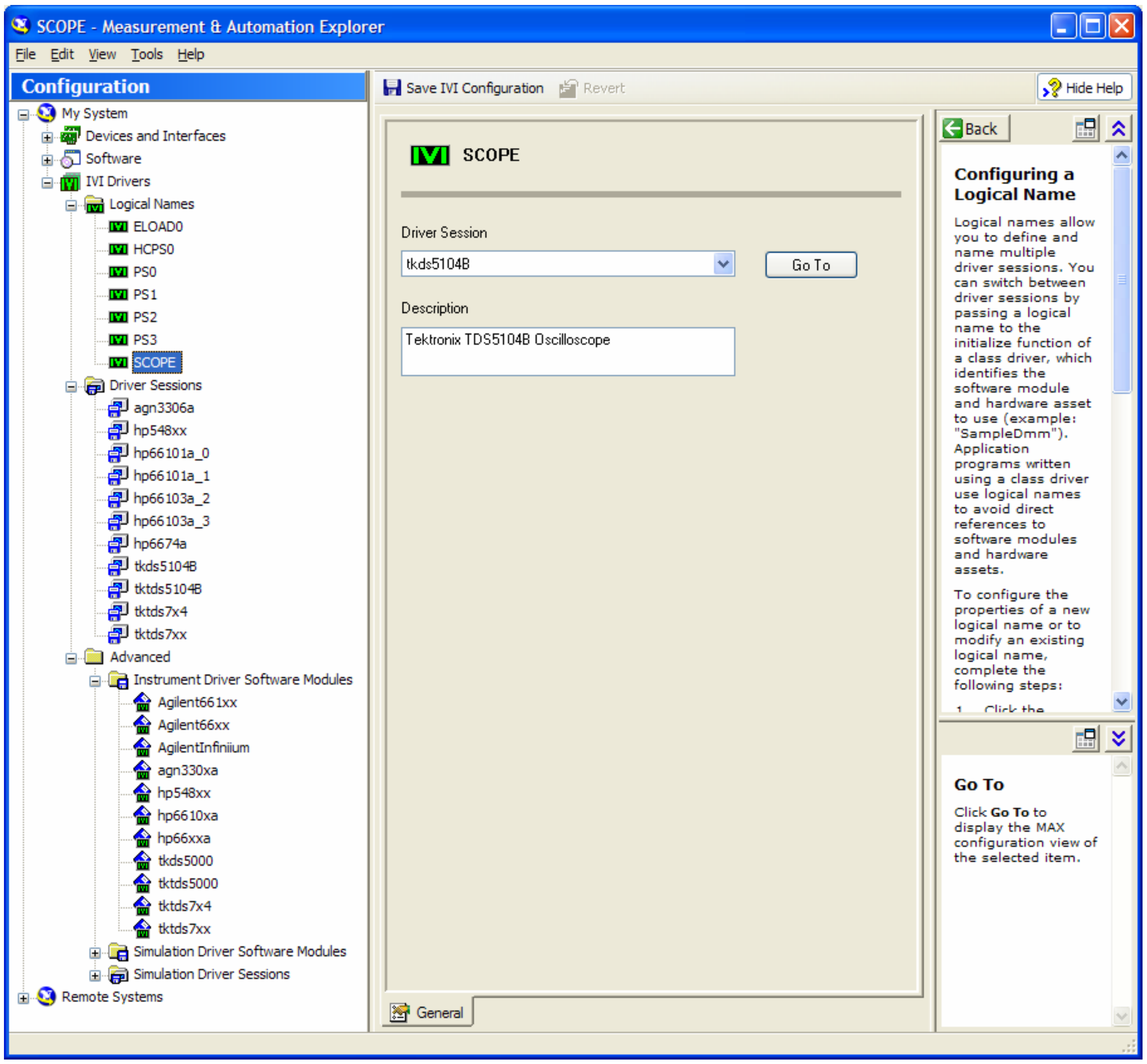
Despite the fact that not all of the instruments in the target test station have IVI specific drivers, the abstraction layer still uses the IVI configuration store to track all of the instruments. This is possible because any driver can be added to the configuration store, whether or not it's an IVI driver. Therefore, for instruments for which there is only a VXI*plug&play* driver available, the IVI configuration store can still hold information about the instrument model. That information can be queried at run time in order to determine which model is currently in use and therefore, which class should be instantiated – the IVI class driver version or the specific VXI*plug&play* driver version.

To configure the IVI drivers and configuration store, IVI Shared Components 1.5.1[iv] was installed on the target computer along with National Instruments' (NI) IVI Compliance Package 3.2[v]. The NI software package includes NI's Measurement and Automation Explorer (MAX), which allows end users to configure their IVI drivers, specify the instrument models that are being used, and associate drivers with logical names. Although MAX is not required to set up the IVI configuration store, it provides an easy-to-use graphical user interface (GUI) and it ensures that the XML file is well-formatted and complete. See Figure 2 and Figure 3 for examples of MAX IVI configuration pages.

Although there is an IVI-COM specific driver available for the HP54831B scope, the abstraction layer currently uses only IVI-C specific drivers. This was a design decision made because there is no IVI-COM specific driver available for the TDS5104B,

[iv] IVI Shared Components 1.5.1 is available on the IVI Foundation web site at http://ivifoundation.org/shared_components/Default.aspx
[v] The current National Instruments' IVI Compliance Package, along with previous versions, is available for download from http://joule.ni.com/nidu/cds/fn/p/sn/n23:3.1637.4711/lang/en

whereas both scopes have an IVI-C specific driver. This means that both scopes can utilize exactly the same IVI setup calls within the abstraction layer, which eliminates the redundancies of writing essentially the same calls in the IVI-COM format. Although the code has not yet been written for the other instrument classes in the abstraction layer, they will probably be structured the same way. This is because most of the IVI instrument drivers currently available for the standard S9100 instruments are IVI-C drivers; many of the instruments have no IVI-COM drivers available right now, and some have no IVI driver at all.



- My System
  - Devices and Interfaces
  - Software
  - IVI Drivers
    - Logical Names
      - ELOAD0
      - HCPS0
      - PS0
      - PS1
      - PS2
      - PS3
      - SCOPE
    - Driver Sessions
      - agn3306a
      - hp548xx
      - hp66101a_0
      - hp66101a_1
      - hp66103a_2
      - hp66103a_3
      - hp6674a
      - tkds5104B
      - tktds5104B
      - tktds7x4
      - tktds7xx
    - Advanced
      - Instrument Driver Software Modules
        - Agilent661xx
        - Agilent66xx
        - AgilentInfiniium
        - agn330xa
        - hp548xx
        - hp6610xa
        - hp66xxa
        - tkds5000
        - tktds5000
        - tktds7x4
        - tktds7xx
      - Simulation Driver Software Modules
      - Simulation Driver Sessions
- Remote Systems

IVI SCOPE

Driver Session: tkds5104B

Description: Tektronix TDS5104B Oscilloscope

Configuring a Logical Name

Logical names allow you to define and name multiple driver sessions. You can switch between driver sessions by passing a logical name to the initialize function of a class driver, which identifies the software module and hardware asset to use (example: "SampleDmm"). Application programs written using a class driver use logical names to avoid direct references to software modules and hardware assets.

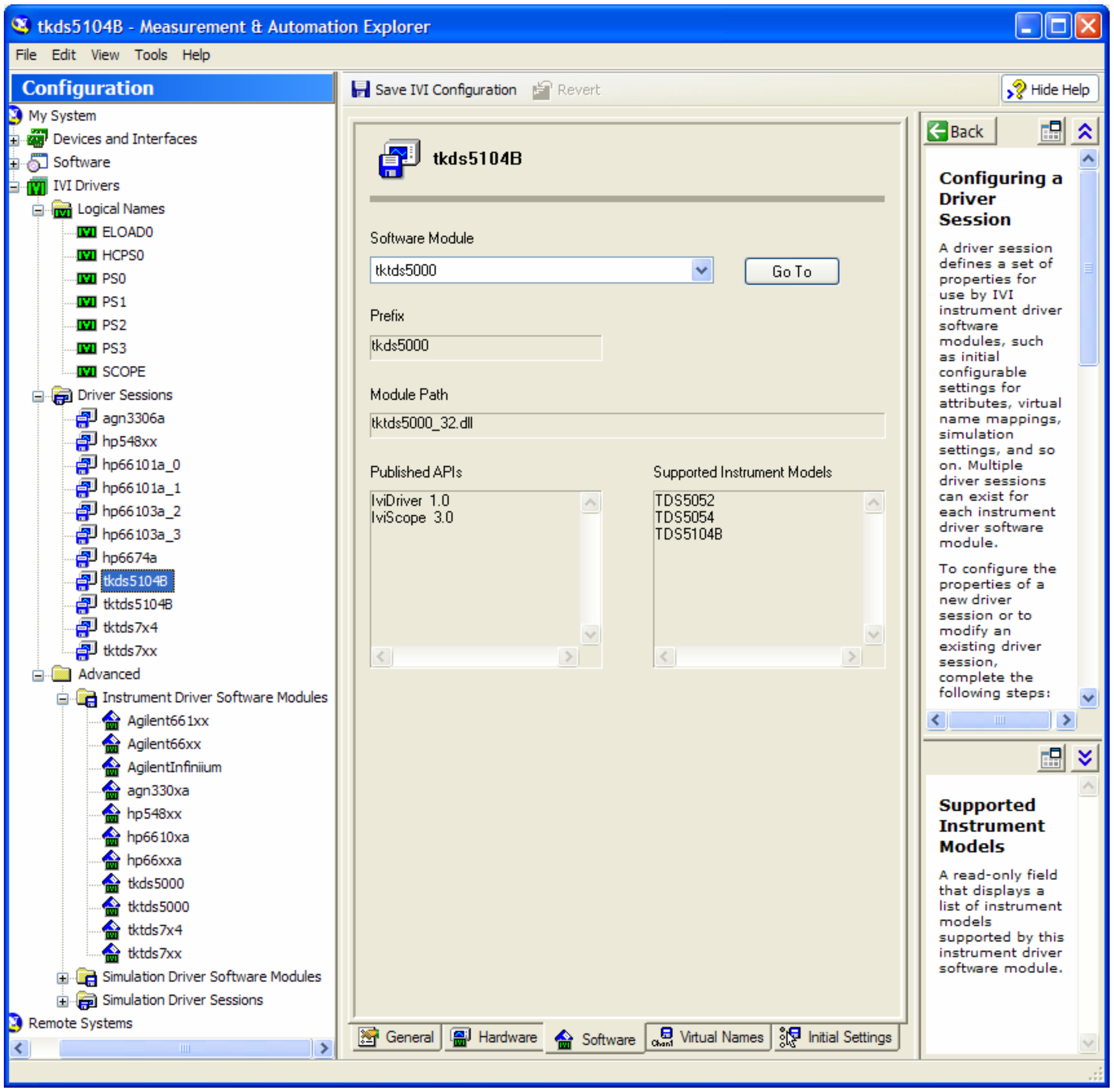
To configure the properties of a new logical name or to modify an existing logical name, complete the following steps:

1. Click the

Go To

Click **Go To** to display the MAX configuration view of the selected item.

Figure 2: National Instruments Measurement & Automation Explorer IVI Configuration Setup – Logical Names



tkds5104B

Software Module: tktds5000

Prefix: tkds5000

Module Path: tktds5000_32.dll

Published APIs:
IviDriver 1.0
IviScope 3.0

Supported Instrument Models:
TDS5052
TDS5054
TDS5104B

Configuring a Driver Session

A driver session defines a set of properties for use by IVI instrument driver software modules, such as initial configurable settings for attributes, virtual name mappings, simulation settings, and so on. Multiple driver sessions can exist for each instrument driver software module.

To configure the properties of a new driver session or to modify an existing driver session, complete the following steps:

Supported Instrument Models

A read-only field that displays a list of instrument models supported by this instrument driver software module.

Figure 3: National Instruments Measurement & Automation Explorer IVI Configuration Setup – Driver Session

In order to use the IVI-C and VXI*plug&play* drivers as easily as possible within the driver system, National Instruments' Measurement Studio was used to create a C# interface for the native C drivers. Measurement Studio achieved this by reading the exported dll functions from the target driver's function panel[vi] to generate C# methods and then adding the C# DllImport command for all of the exported functions. The converter also created a private class called "PInvoke" that contains all of the DllImport commands. As an example, for the tktds7xx (TDS654C driver) "reset" command, tktds7xx_reset, the driver converter created a C# function called "reset", which calls the tktds7xx_reset function:

```
public int reset()
{
    int pInvokeResult = PInvoke.reset(this._handle);
    PInvoke.TestForError(this._handle, pInvokeResult);
    return pInvokeResult;
}
```

In the PInvoke class, the converter created the following lines of code:

```
[DllImport("tktds7xx.dll",EntryPoint="tktds7xx_reset",CallingConvention=
CallingConvention.StdCall)]
public static extern int reset (System.IntPtr Instrument_Handle);
```

In order to use the DllImport calls, the file includes the directive

```
using System.Runtime.InteropServices;
```

One helpful feature when using NI's Measurement Studio to create a C# interface is that it can preserve all of the information about the function calls and their parameters and turn them into C# "summary" tags that provide context sensitive help for when the programmer uses the functions. If the interface were created by hand, it would take hours to copy and paste all of the function and parameter help into the C# code, so this is a useful and extremely helpful feature.

[vi] A function panel (.fp file) is a National Instruments file type that can be read in NI's LabWindows/CVI ADE and allows for GUI-based entry of a function's parameters and return values, along with function- and parameter-sensitive help.

Instrument Driver System Structure

The instrument driver system is written in an object-oriented, hierarchical fashion. All of the instrument types have an abstract base class that implements the IInstrumentDriver interface, which includes the most basic methods for an instrument – Init, Reset, and Close – and C# getters for the instrument status value and error message. For the oscilloscope, the “Scope” class contains many classes that are used to track the state of the instrument and information about the waveforms to be acquired and measured. The four most important configuration classes are shown in a UML class diagram in Figure 4. Every field in each of the configuration classes has a default value that is set by calling the appropriate “setDef” method, such as “`setDefHorizontal`(…)”.

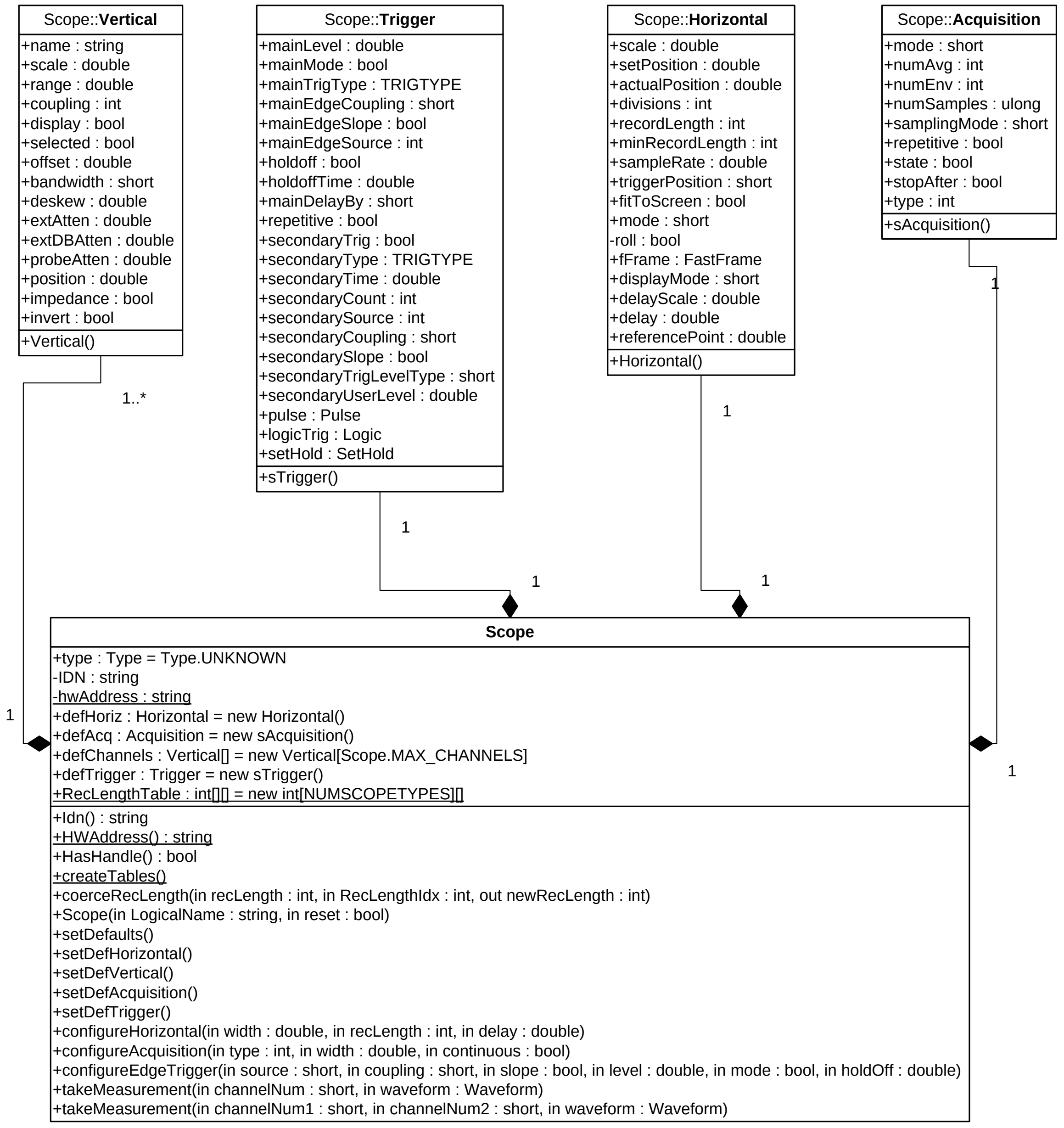


Figure 4: UML class diagram of the Scope class and four of the most important state classes

For each instrument type, the abstract base class has one or more inherited classes. If all of the models for a given type have IVI-C class-compliant drivers, there is only one inherited class. Otherwise, there is one inherited class for each specific driver. In the case of the scope, both the TDS5104B and the HP54831B have IVI-C class compliant drivers, so they both use the IviScope class, which inherits from the Scope class. For the

TDS654C model, though, there is a separate class called "TDS654C", which also inherits from the Scope class. These examples are shown in Figure 5 and Figure 6. In both figures, certain fields and methods, along with initial field values and method parameters, have been deleted to conserve space.

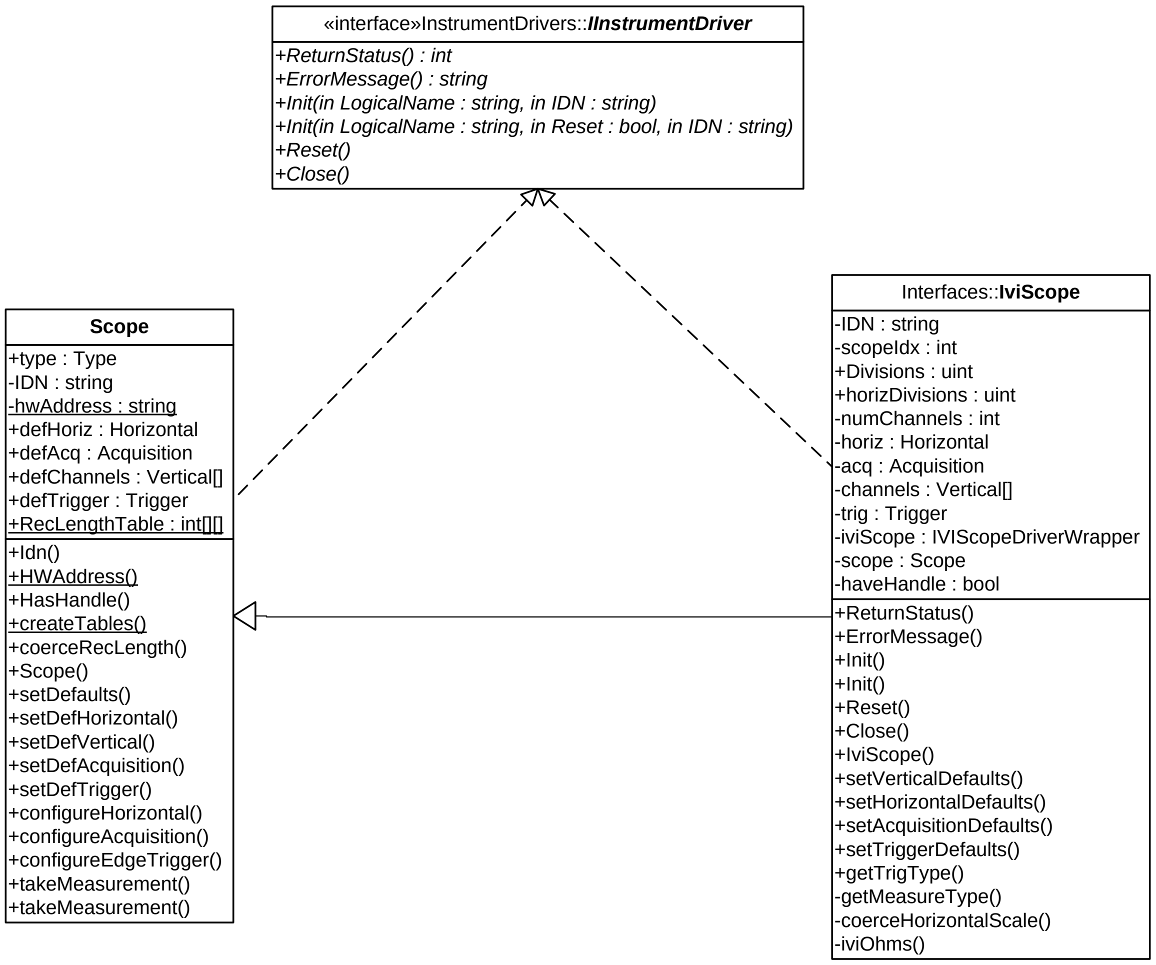


Figure 5: UML Static Diagram of the relationship between the abstract Scope class, the interface IInstrumentDriver, and the class IviScope

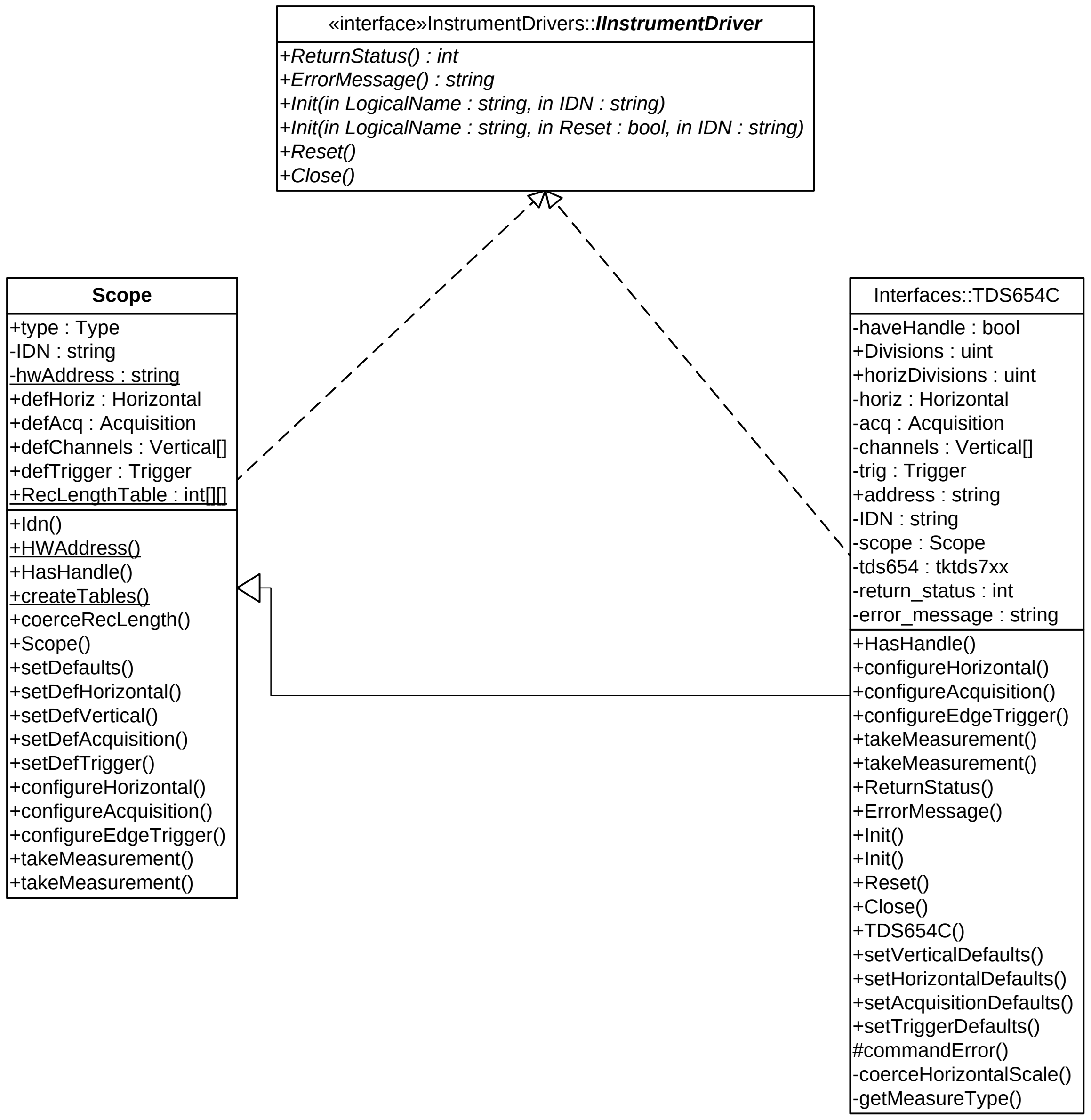


Figure 6: UML Static Diagram of the relationship between the abstract Scope class, the interface IInstrumentDriver, and the class TDS654C

When a test program or other application first calls the “init” command for an instrument, the program passes the logical name for the instrument. From that, the driver system must determine which of the instrument type’s subclasses it must instantiate. To do this, the IVI configuration store is queried for the current driver session, and from there, the exact hardware model and, optionally, the hardware I/O resource string (address) can be determined. The following calls are an example of how to do that (some initialization calls have been omitted):

```
static IviDriverSession ds = new IviDriverSession();
static IviHardwareAsset hwAsset = new IviHardwareAsset();
ds = configStore.GetDriverSession(LogicalName);
hwAsset = ds.HardwareAsset;
hwAddress = hwAsset.IOResourceDescriptor;
```

The driver system compares the hardware asset's name against a list of known instrument models to determine which model is in use in the system. From there, it can instantiate the correct class. All classes that inherit from a base instrument type must implement all of the same public methods so that the same methods can be called regardless of the class that was instantiated.

Once the correct class is instantiated, the "init" method is called. This method is responsible for calling all of the "setDef" methods mentioned earlier. It also resets the instrument, applies the default settings, and creates local versions of the vertical, horizontal, trigger, etc, classes. This guarantees that all models will start in essentially the same state. After that, the driver system will change the instrument's settings, as appropriate, in response to calls from the test program. Each time an instrument setting is updated successfully, the driver system updates the appropriate settings in the appropriate classes. For instance, if the test program requests a particular trigger setting, the driver system makes the appropriate calls to the instrument and then updates the settings in its locally instantiated trigger class.

Most of the classes for an instrument type are instantiated once per instrument. For some classes, though, it is necessary to instantiate multiple copies. One example of this is the "Vertical" class in the oscilloscope class; the "Vertical" class is instantiated once for every scope channel. This is necessary because each channel can have completely separate settings, whereas there is only one trigger and one time base for the entire oscilloscope.

Although using IVI class drivers makes much of the work easier, there are cases where it actually causes problems. Because the IVI class driver specifications don't detail exactly what each of the driver calls is supposed to do at the hardware level, some of the IVI class-compliant specific drivers interpret certain settings in different ways. For example, the TDS5104B treats vertical position and offset as separate entities, whereas the HP54831B does not have a setting called "position". The IVI scope class driver specification appears to use the Agilent scope's method of combining the two, so for setting the signal position for the TDS5104B, the IVI driver is insufficient.

To combat this, the driver system's IviScope class also uses the VXI*plug&play* driver to get lower-level control. The driver system queries the scope type that is in use in order to determine which VXI*plug&play* driver it needs and then instantiates it. If there are any cases in which the IVI class driver does not suffice, the VXI*plug&play* driver can be used to get finer control over the instrument.

Another problem with the IVI driver was encountered during the development of the test case. While using the HP54831B scope, the timing measurements initially failed. Despite the fact that the test signal's period was passed to the appropriate IVI scope class driver call, the scope did not set the time base to allow enough of the signal to be displayed in order to take period or frequency measurements. In order to prevent this problem, the instrument driver system multiplies the expected period by 1.25 and then passes the resulting value to the IVI scope class driver call. This method ensures that enough of the waveform will be displayed in order for the scope to take timing measurements such as period and frequency.

Chapter 5: Results

For the test case, probes were connected to each of the three oscilloscope models in order to access test signals. The signals chosen for this test were the probe compensation outputs provided by the TDS654C and TDS5104B oscilloscopes. The true purpose of the outputs is to calibrate any probes that are attached to the oscilloscope channels, but they provided fairly accurate[vii, 10] signals that were simple to use as control inputs. Both signals had a frequency of approximately 1 kHz, and the signal from the TDS654C scope was approximately 0.5 volts peak-to-peak, versus approximately 1.0 volt for the TDS5104B signal. The TDS654C's signal was connected to channel 1 of each scope, while the TDS5104B's signal was connected to channel 2. A simple test application was written to configure basic settings on the scope, and then the application requested period, frequency, and voltage amplitude readings for both signals.

The test application was run once for each of the three scopes. Before each new scope was used, the IVI configuration store was updated to reflect the new target model. The only system change that was required was to navigate to the Logical Names section in Measurement and Automation Explorer (see Figure 2), click the drop-down box for the driver session, select the appropriate specific driver, and then click "Save IVI Configuration". Each of the oscilloscopes was set to use GPIB bus address 6, so the models that were not currently being used were set to the "off bus" setting. Other than these two minor steps, nothing needed to be changed in order to run the same test application on each of the three scope models.

[vii] Frequency specifications: TDS654C and TDS5104B = 1 kHz ± 5%; Voltage peak-to-peak specifications: TDS654C = 500 mV ± 1.0%; TDS5104B = 1.0 V ± 1.0%

To verify that each of the scope models was taking the correct measurements, the test application printed the results of each of the measurements to text files. The printouts did not include the units, but for each of the measurements, the readings are given in terms of the base units for the measurement type. In other words, voltage measurements are given in volts, period measurements in seconds, and frequency measurements in hertz. The contents of the files are shown below, followed by the results of the mean and standard deviation calculations for each of the measurements.

```
Current scope model: TDS654C
Period Measurement on CHANNEL1: 0.00099999
Frequency Measurement on CHANNEL1: 996.0283
VAmpl Measurement on CHANNEL1: 0.499
Period Measurement on CHANNEL2: 0.00100399
Frequency Measurement on CHANNEL2: 1000.036
VAmpl Measurement on CHANNEL2: 0.992

Current scope model: HP54831B
Period Measurement on Channel1: 0.0009990394
Frequency Measurement on Channel1: 999.68356
VAmpl Measurement on Channel1: 0.4971
Period Measurement on Channel2: 0.00100247827
Frequency Measurement on Channel2: 997.565554
VAmpl Measurement on Channel2: 0.993
```

Current scope model: TDS5104B

Period Measurement on CHANNEL1: 0.0010003

Frequency Measurement on CHANNEL1: 999.6786

VAmpl Measurement on CHANNEL1: 0.49608

Period Measurement on CHANNEL2: 0.0010024

Frequency Measurement on CHANNEL2: 997.5578

VAmpl Measurement on CHANNEL2: 0.98592

	Channel1			Channel2		
	Period	Frequency	Vampl	Period	Frequency	Vampl
TDS654C	9.9999E-04	996.028	0.499	1.0040E-03	1000.036	0.992
HP54831B	9.9904E-04	999.684	0.4971	1.0025E-03	997.566	0.993
TDS5104B	1.0003E-03	999.679	0.49608	1.0024E-03	997.558	0.98592
Mean	9.998E-04	998.463	0.497	1.003E-03	998.386	0.990
Std dev.	6.569E-07	2.109	0.001	8.962E-07	1.429	0.004

Table 1: Mean and standard deviation for measurements taken by all three scope models

The standard deviations for the period measurements for both signals (channel 1 and channel 2) are extremely small – less than 1 us each for measurements of approximately 1 ms. The amplitude reading for the signal on channel 2 has the poorest standard deviation – 0.004 V for a 0.990 V signal. This represents a 4.43% difference from the mean of the amplitude readings. The precision of the amplitude reading by the TDS5104B was good, though, as multiple test runs consistently produced the same voltage reading of 0.98592 V. The specification for the voltage amplitude on the TDS5104B probe compensation signal is 1.0V +/- 1%[10], so the expected readings were between 1.01 V and 0.99 V. The particular TDS5104B oscilloscope used for this experiment was out of calibration, though, and this could certainly account for the inaccuracies.

After the voltage amplitude reading was taken for each signal on each scope, the test application was paused so that screen shots of the signals on the oscilloscopes could be taken. These are shown in the figures below.

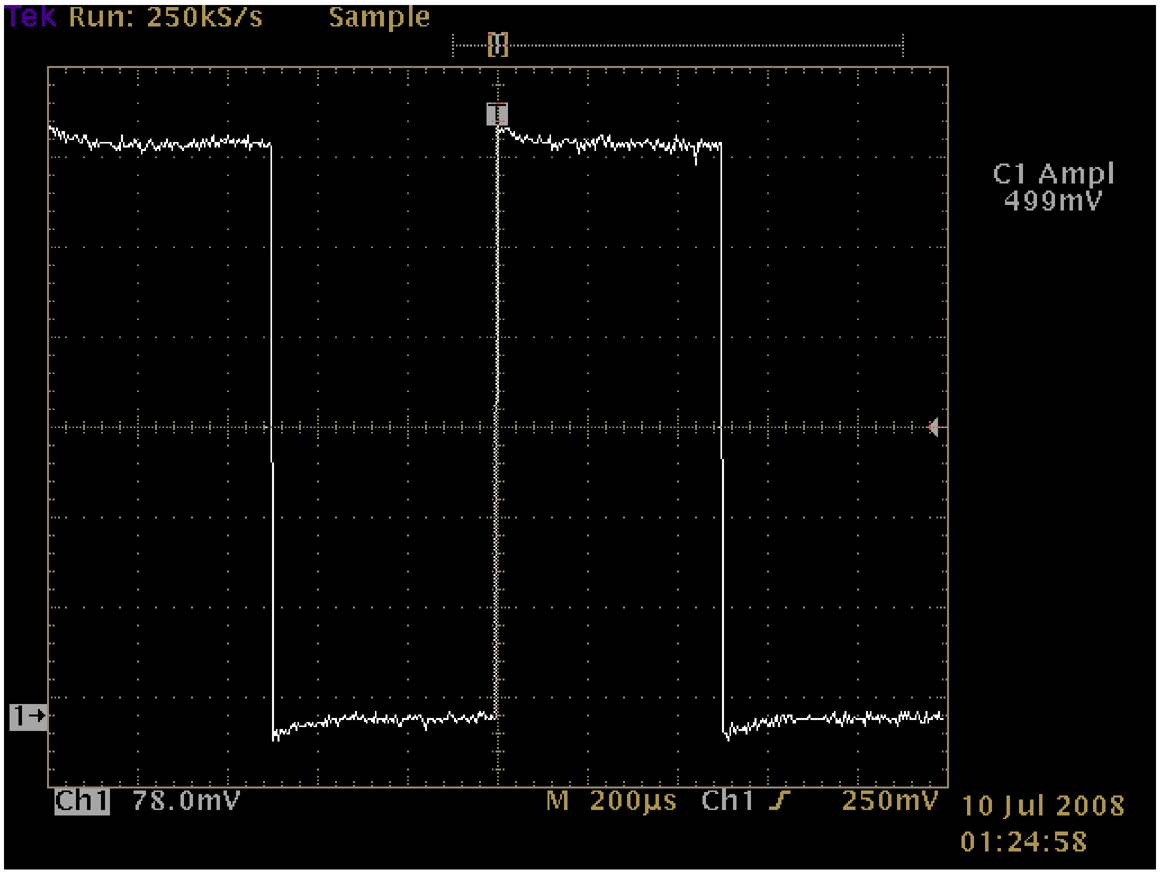


Figure 7: TDS654C scope displaying its own probe compensation signal on channel 1

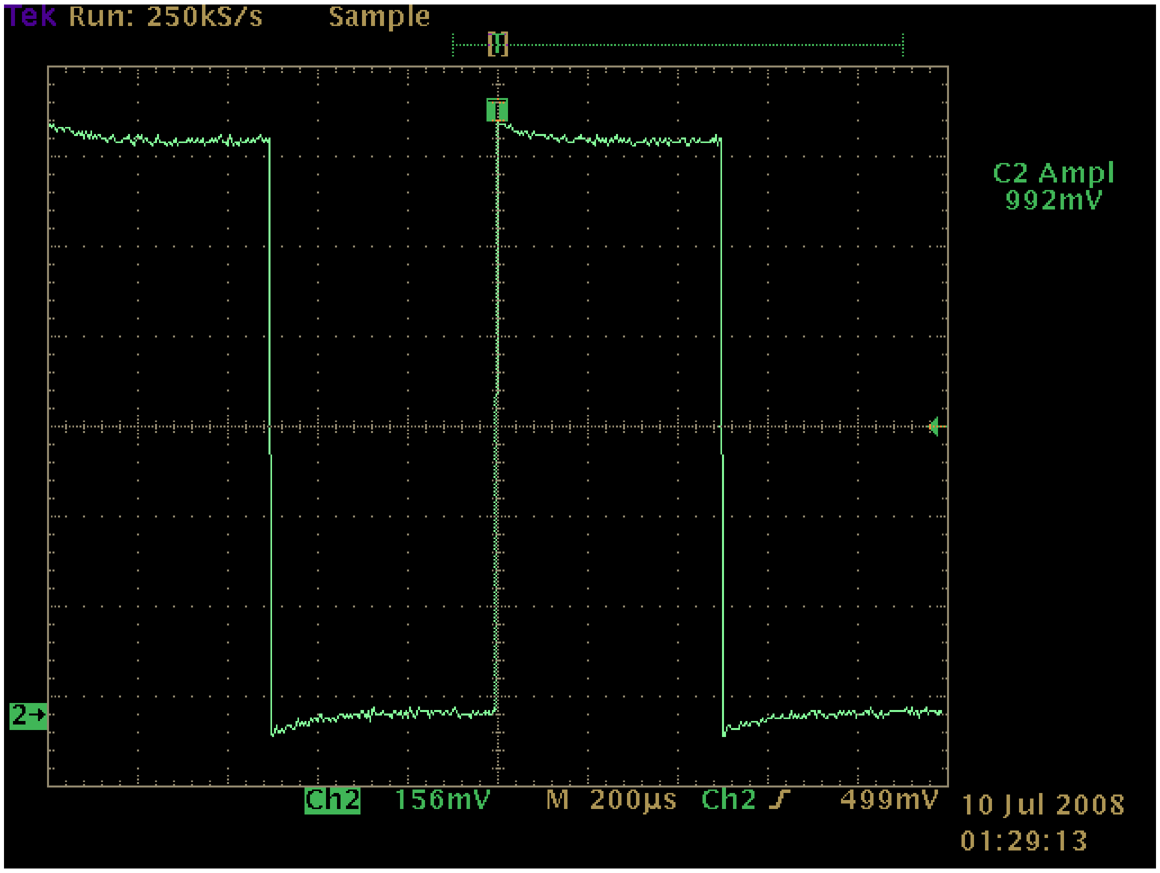


Figure 8: TDS654C scope displaying TDS5104B probe compensation signal on channel 2

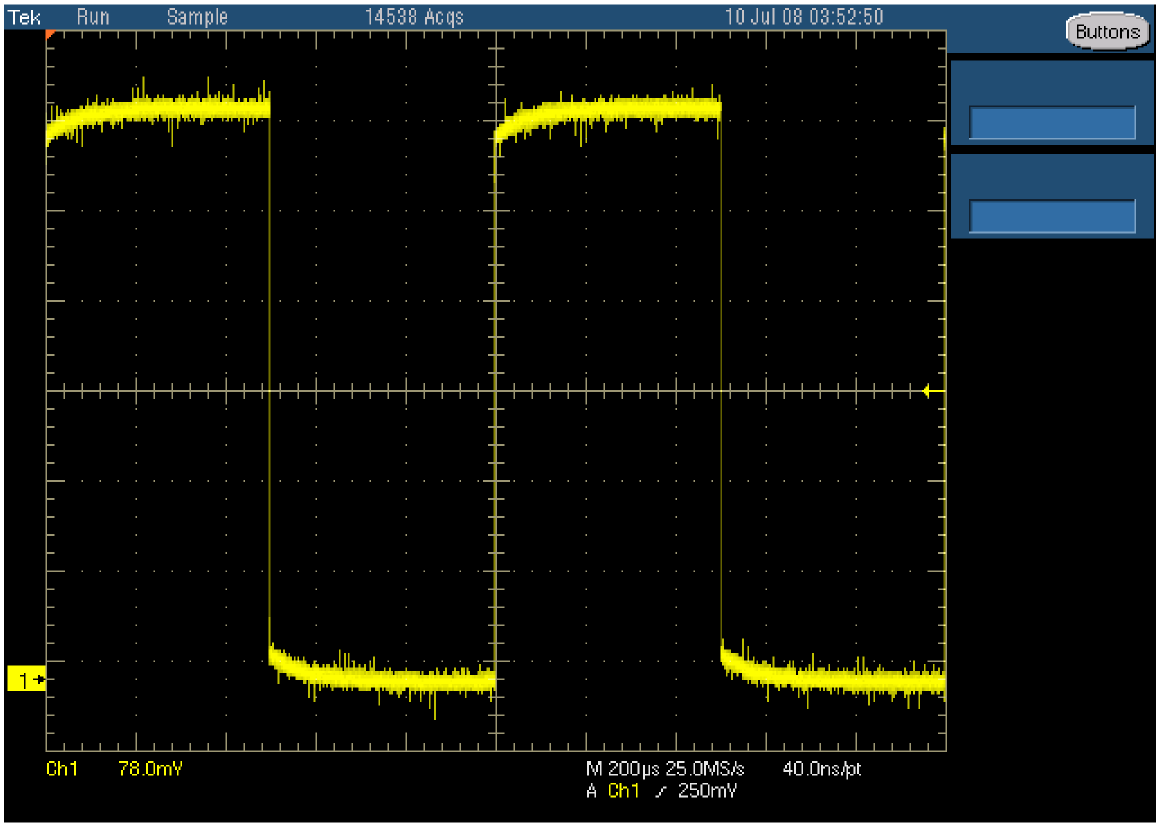


Figure 9: TDS5104B scope displaying TDS654C probe compensation signal on channel 1

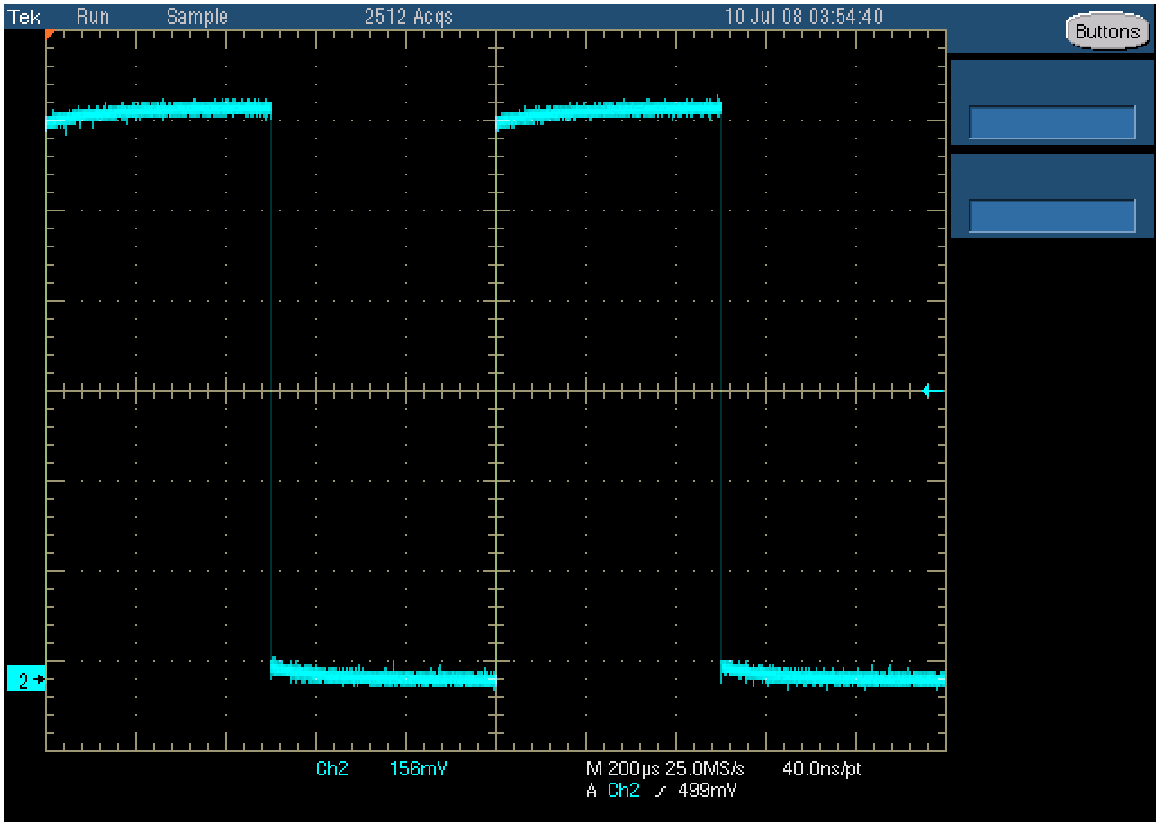


Figure 10: TDS5104B scope displaying its own probe compensation signal on channel 2

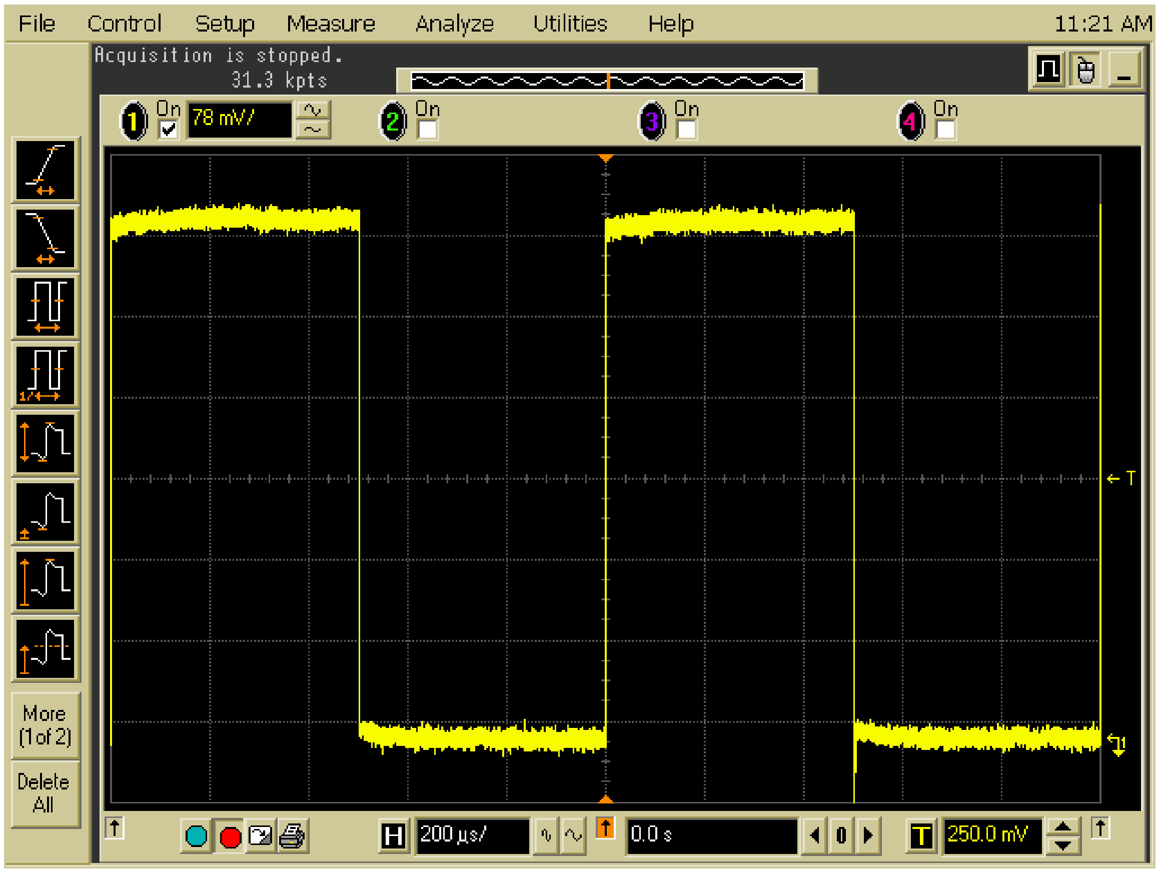


Figure 11: HP54831B scope displaying TDS654C probe compensation signal on channel 1

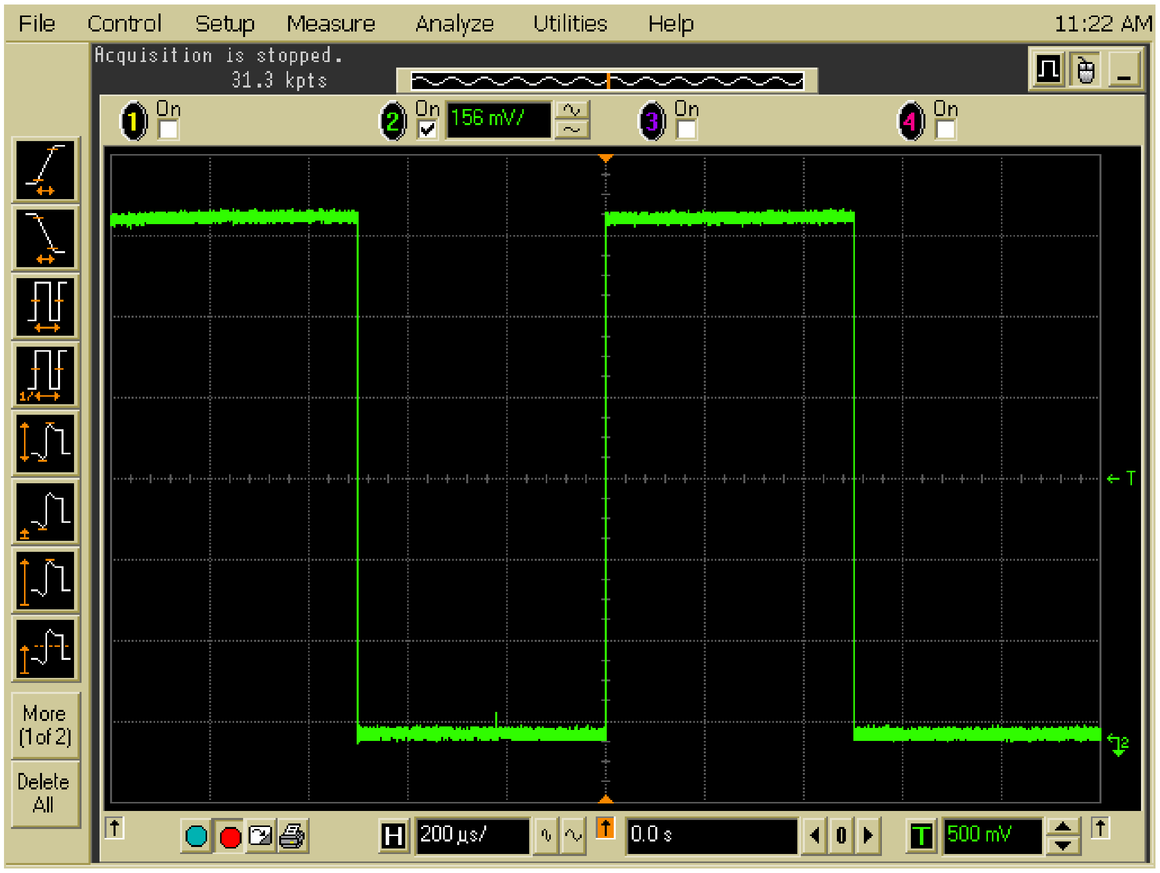


Figure 12: HP54831B scope displaying TDS5104B probe compensation signal on channel 2

The figures show that each of the oscilloscopes was configured with the same vertical scale (volts per division) of 78 mV / division for the first signal and 156 mV / division for the second signal. The vertical scale is stretched as much as possible, while still allowing room for waveforms that could be slightly outside the target voltage. This means that the vertical (voltage) accuracy is improved over a more basic setting of 100 or 200 mV per division. Additionally, all of the scopes captured enough of the waveforms to perform period and frequency measurements without expanding the waveforms so much that horizontal (time) accuracy would be lost. In this case, each scope’s time base was set to 200 us, but if the signals had required a slightly higher time base, there would have been a difference between the TDS654C and the other two oscilloscopes. This is because, as mentioned earlier, the TDS654C uses a 1-2-5 sequence, whereas the TDS5104B uses a 1-2-4 sequence and the HP54831B is not constrained by a set sequence at all. Because the HP scope’s time base is variable, it was forced by the abstraction layer to conform to the TDS5104B’s more limited time base settings.

Chapter 6: Conclusion

Although the test signals required a fairly simple oscilloscope configuration, they still show that a well designed instrument driver system can be a valuable tool in test program development. The test application demonstrated that the driver system can provide the same results even when using instrument models and specific drivers from different manufacturers. With IVI class drivers alone, the TDS654C would not have been available as a replacement instrument because it does not have an IVI driver. Similarly, with a simple abstraction layer, the huge differences in driver function calls and structure between the Tektronix VXI*plug&play* scope drivers and the Agilent VXI*plug&play* scope driver would make it extremely difficult to come up with a logical driver interface for the test program. Additionally, the IVI-C driver for the Agilent scope did not adequately configure the scope in order to take periodic measurements during the experiment.

Using Microsoft Visual C# .NET 2003 made it easy to create a flexible system where new code for additional instrument models could be added easily. Because classes are instantiated only after querying the IVI configuration store, only the driver for the currently active model need be installed on the controlling computer. With a simple abstraction layer written in a language such as C, drivers for all the possible instrument models must be present on the controlling computer or else the abstraction layer’s dll will not load. This means that a test station without a TDS654C scope, for example, will still need to have the TDS654C driver installed on the computer, which is a superfluous requirement.

Because of the number of lines of code and the hours involved in creating and maintaining an instrument driver system, it is not a cost-effective solution for all situations. For test stations that have only a small number of test programs, or for production environments where test programs have a short life span, an instrument driver system is an unnecessary expense. On the other hand, for test stations with hundreds of test programs that can last for ten years or longer, a driver system can save hundreds of thousands of dollars in the long run.

Future Work

Although it is a major software project in itself, the driver system should not be seen as a standalone product. Rather, it should be part of a suite of applications that all aid in the development of instrument setup and measurement for a test program. A natural extension to the driver system is a GUI that the test program developer can use to fill in the function calls. This method of test program creation will ensure that the driver system is used in the manner intended and it will speed test program development time. The GUI should pose a series of increasingly specific questions in order to lead the user to the appropriate function calls.

One possible series of questions is shown below, in Figure 13, Figure 14, and Figure 15. In the case that the user wants to apply a 10 kHz, 5 volts peak-to-peak square wave from an arbitrary waveform generator, he would select the options shown. First the user would click “Source a signal” on the first page, which would then cause the “Source a Signal” page to appear. On that page, he would click the “Standard” button, which would then lead to a panel where he could select a square wave (not shown), and so on.

Finally, after clicking the appropriate sequence of buttons, the user would be presented with a screen on which to enter the specific signal characteristics (Figure 15).

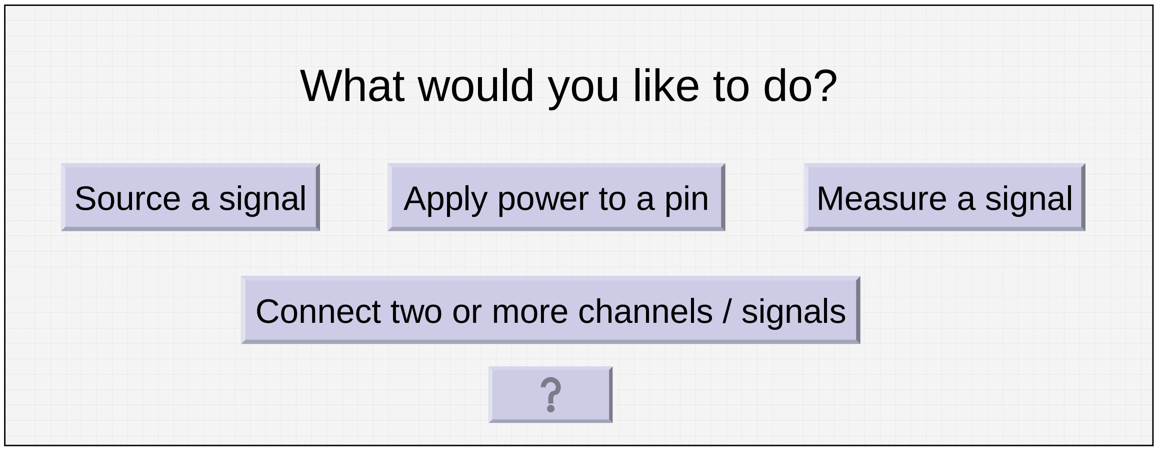

What would you like to do?
Source a signal
Apply power to a pin
Measure a signal
Connect two or more channels / signals
?

Figure 13: Abstraction Layer User GUI -- Initial window displayed to test program developer

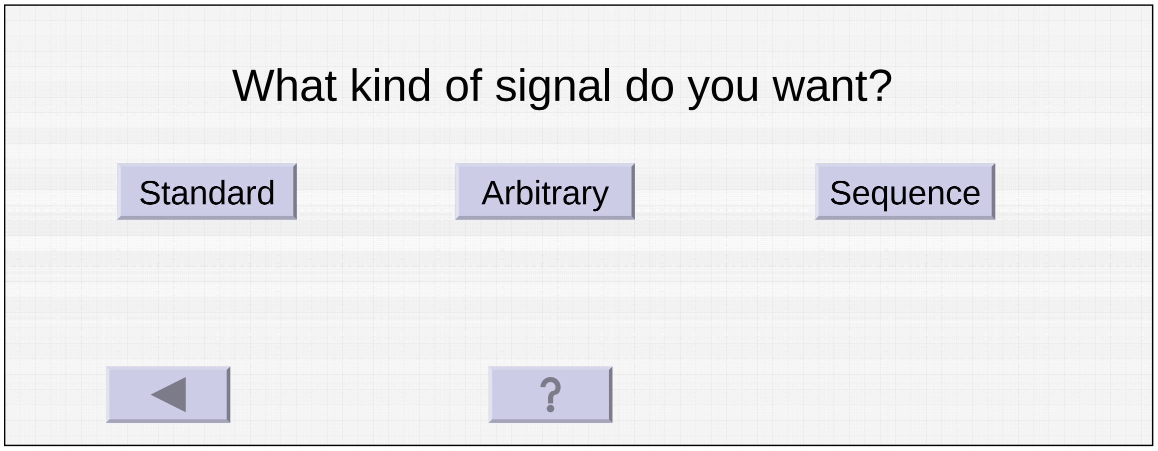
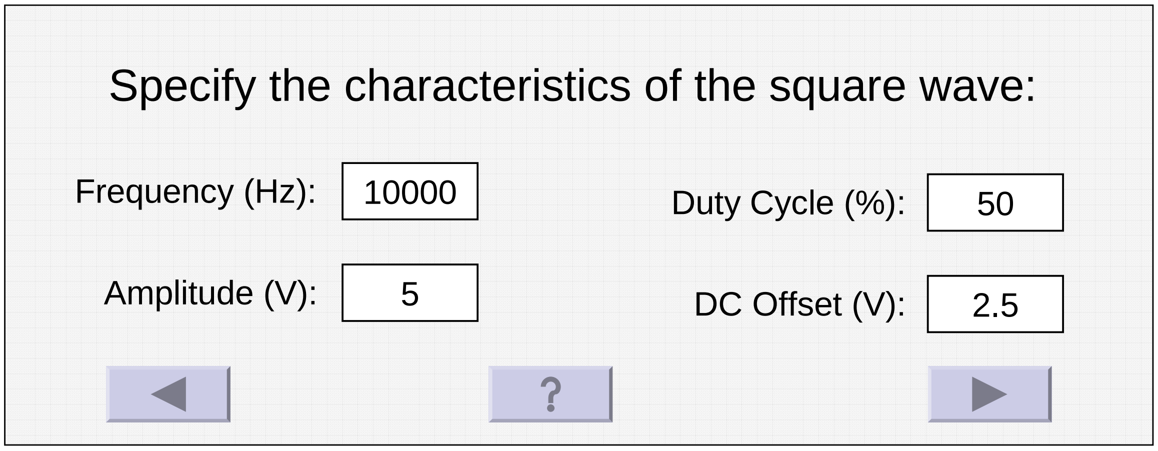


Figure 14: Abstraction Layer GUI -- Page presented after user selects "Source a Signal"


Specify the characteristics of the square wave:

Frequency (Hz): 10000

Duty Cycle (%): 50

Amplitude (V): 5

DC Offset (V): 2.5

Figure 15: Abstraction Layer GUI -- Page presented once user has walked through "Source a Signal" path

Finally, the abstraction layer GUI would produce the code snippet required by the driver system in order to produce the desired signal (this code is just an example as the driver system interface for the arbitrary waveform generator has not yet been developed, but it conveys the general idea):

```
ARB_StdWaveformType (SQUAREWAVE);
ARB_Waveform_Freq (10e3);
ARB_Waveform_Vpp (5.0);
ARB_Waveform_Offset (2.5);
ARB_Waveform_Duty (0.5);
ARB_Output (OUTPUT_ON);
```

In addition to creating the test program code, the code generator could inform the developer about any instrument limitations. For example, if the developer wanted to use a particular oscilloscope trigger mode that isn’t supported by at least one of the models currently in use by the driver system, the code generator would pop up a message that explains which models will not work and why. The developer would then have the option to change the test strategy or decide that they just will not be able to run the test program with the less capable oscilloscope model.

Another utility that would be useful and could be added to the code generator GUI is an accuracy analysis tool. As mentioned earlier, all test programs require accuracy analysis for the signals that will be sourced or measured based on the equipment that will be used to source or measure the signal. The developer could enter the required measurement and tolerance from the test specification, and then the accuracy analysis tool could determine if the accuracy is good enough with each of the possible target instrument models. It could also produce an accuracy analysis report for the developer to include as part of the development process.

Lastly, the instrument driver system requires a COM interface. Because the driver system was created in Microsoft's C# .NET language and compiled into a class library assembly, it can be utilized in its native form only by other .NET languages. On the target test platform, the S9100 test station, C is the current standard language used for test program development, which means that the assembly cannot be used natively. To handle this, there must be a COM interface to the driver system, which can be registered in the Windows® environment, allowing the C-based test programs to use it. This COM interface will essentially be a pass-through layer, compiled into a dynamic link library (dll) with exported functions that hide the COM interface from the test program. This way, the test engineers will not have to understand or directly use COM technology; they can simply call the exported functions as if there were no COM interface involved. The exported functions in the driver system interface will then turn the calls from the test program into COM function calls in order to access the .NET drivers. See Figure 16, below. For test programs or other end applications that are written in a .NET language, they can directly access the driver system functions without using COM; they will merely reference the assembly.

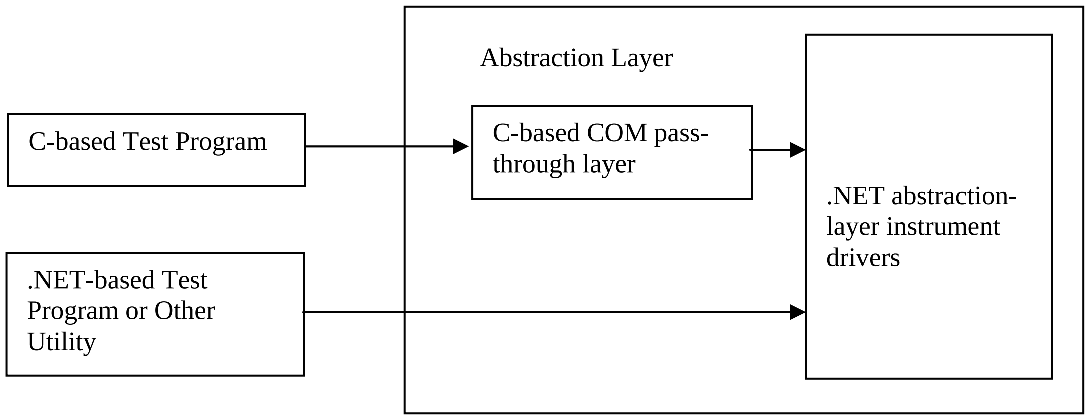

Abstraction Layer
C-based Test Program
C-based COM pass-through layer
.NET abstraction-layer instrument drivers
.NET-based Test Program or Other Utility

Figure 16: Abstraction Layer Interface Options

References

[1] http://ieeexplore.ieee.org/xpls/abs_all.jsp?tp=&isnumber=775&arnumber=19528&punumber=2610
[2] http://www.scpiconsortium.org/faqs.htm#3
[3] Bode, F. "IVI comes of age: An overview of IVI specifications with current status" Aerospace and Electronic Systems Magazine, IEEE Volume 18, Issue 8, Aug 2003: 31-34 < http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=1224970&isnumber=27499&punumber=62&k2dockey=1224970@ieeejrns>
[4] http://www.scpiconsortium.org/scpistandard.htm
[5] http://www.scpiconsortium.org/faqs.htm#8
[6] *VXIplug&play Systems Alliance Specification VXIplug&play-3.1: Instrument Drivers Architecture and Design Specification: http://ivifoundation.org/docs/vpp31.pdf*
[7] *VXIplug&play Systems Alliance Specification VXIplug&play-1: Charter Document* http://ivifoundation.org/docs/vpp1.pdf
[8] IVI Foundation Specification *IVI-3.1: Driver Architecture Specification http://ivifoundation.org/docs/Ivi31%202007-10-22.pdf*
[9] http://www.scpiconsortium.org/characteristics.htm
[10] "TDS 500D, TDS 600C, TDS 700D & TDS 714L Digitizing Oscilloscopes Performance Verification and Specifications" Chapter 1-74 to 1-76 <http://www2.tek.com/cmsreplive/marep/10151/071063003_2008.06.12.14.07.27_10151_EN.pdf > and "TDS5000B Series Digital Phosphor Oscilloscopes Specifications and Performance Verification" Chapter 1-19 <http://www2.tek.com/cmsreplive/marep/10475/071142003_2008.06.16.11.15.35_10475_EN.pdf >