Using IVI Drivers to Build Hardware-Independent Test Systems with LabVIEW™ and LabWindows™/CVI

John Pasquarette

Interchangeable Drivers

Interchangeable Virtual Instrument (IVI) drivers are an exciting new technology for test engineers because they can now reuse their test programs with different instruments. LabVIEW and LabWindows/CVI users can buy a collection of these drivers as the IVI Driver Library. This application note outlines the theory, tools, and mechanisms involved in using these tools to build a hardware-independent test system.

The Benefits of Instrument Interchangeability

The benefits of instrument interchangeability extend into a wide variety of application areas and industries. Test system developers in the military and aerospace industries, who must maintain test systems and code for many years, can easily reuse their test code on new equipment as instruments are improved or become obsolete. Manufacturers in competitive, high-volume industries, such as telecommunications and consumer electronics, can keep their production lines running when instruments malfunction or must be recalibrated. And large manufacturing companies of all kinds can more easily reuse and share test code between departments and facilities without being forced to use the same instrumentation hardware.

IVI Driver Architecture Overview

Interchangeability using IVI drivers is achieved through generic instrument class drivers. A class driver is a set of functions and attributes for controlling an instrument within a specified class, such as an oscilloscope, digital multimeter (DMM), or function generator. The IVI Driver Library has five classes – oscilloscope, DMM, arbitrary waveform/function generator, switch, and power supply. Each one of these generic class drivers makes calls to specific instrument drivers to control the actual instruments. The specific instrument drivers contain the information for controlling a particular instrument model, including the command strings, parsing code, and valid ranges of each setting for that particular instrument. From your test program, you make calls to the class drivers, which in turn communicate through the specific drivers for your instruments. You can change the specific instrument drivers (and corresponding instruments) in your system underneath the class driver without affecting your test code.

July 1998

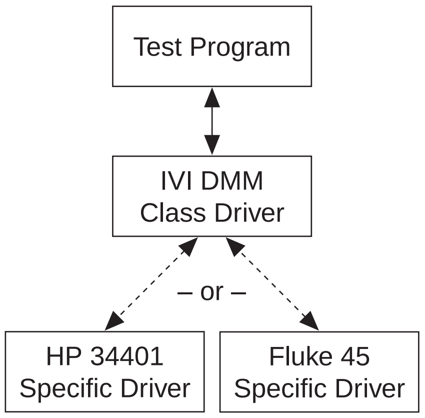


Figure 1. IVI Driver Architecture – the class driver contains generic functions for controlling a DMM. The specific drivers contain information for controlling a specific instrument, such as command strings, parsing code, and valid ranges for instrument settings.

A Word on Interchangeability

Instrument interchangeability has long been a goal of many engineers building test systems, particularly in the military and avionics industries. When discussing these possibilities, it is important to realize that software interchangeability of instruments through instrument drivers is still limited by the interchangeability of the hardware in question. For example, with IVI drivers you can develop test code that works with any DMM. However, the requirements of your test system are still the driving force behind which particular instruments you use. If your test system requirements are DMM measurements with 8½ digits of precision, that means you must always use an 8½ digit DMM. You cannot replace an 8½ digit DMM with a 5½ digit DMM in your test system, unless you only need 5½ digits of precision, regardless of the software architecture. IVI defines a standard architecture for swapping instruments that are capable of delivering the minimum measurement requirements of your test system.

Instrument Classes – Standard Instrument Programming Interfaces

The model for achieving instrument interchangeability is through instrument classes, or types. By defining a generic programming interface for instrument classes, such as oscilloscopes, DMMs, and function generators, test developers can write test code that is independent from the instrument hardware underneath. This enables test developers to change instruments across manufacturers, and even from GPIB to VXI or PXI as the need arises.

Defining the instrument classes was driven by an open end-user organization, the IVI Foundation. The IVI Foundation's charter is to define flexible programming interfaces for instrument classes that meet the needs of test system developers. (For more information on the IVI Foundation, visit the IVI Foundation web site at `www.ivifoundation.org`). The specifications developed by the IVI Foundation have been used as a guideline for building the IDL ToolPak drivers. Instrument classes are defined as a collection of instrument attributes and a standard API for programming these attributes. For example, the oscilloscope class contains a collection of attributes that are common to all oscilloscopes, such as vertical range, offset, timebase, trigger mode, and so on. The class also contains functions for programmers to set these attributes or retrieve data from the instrument, such as `ConfigureVertical`, `ConfigureHorizontal`, `ReadWaveform`, and so on. By defining a standard definition for each of these functions and attributes for an oscilloscope, it is possible to write test programs that work with any oscilloscope.

Because all instruments do not have identical functionality or capability, it is impossible to create a single programming interface that works with all of them. For this reason, the IVI Foundation instrument class specifications are divided into *Fundamental Capabilities* and *Extensions*. The fundamental capabilities are the functions and attributes of an instrument class that should be common across most of the instruments available in that class (the IVI Foundation's goal is to cover 95% of the instruments in a particular class). For example, you can easily outline a fundamental set of functions and attributes for an oscilloscope for setting vertical and horizontal settings and taking measurements. Extensions are functions and attributes that represent more specialized features of an instrument class. For example, although all oscilloscopes have very similar fundamental capabilities for vertical and horizontal settings, there is a wide variety of trigger modes across oscilloscopes. The IVI Foundation specification for oscilloscopes includes extensions for different scope trigger modes, such as video triggers, runt trigger, width trigger, and so on. Through extensions, the IVI Foundation has created *standard* programming interfaces for features and capabilities that are not standard in every scope. Therefore, every scope that accepts video signals will then comply with the video signal extension functions and attributes of the IVI specification.

Configuring Your System

Class drivers communicate with specific instrument drivers through the use of an INI configuration. The `IVI.INI` file identifies the specific instruments that are being used in your system. When you change instruments in your system, you simply change the INI file without editing or recompiling your test programs. This mechanism is triggered through the initialize functions in the class drivers. For example, when you initialize an instrument using a class driver, you do not pass the driver standard VISA resource string, such as "GPIB::2::INSTR". Instead, you pass it a logical name, such as "DMM1." The INI file contains information that associates "DMM1" with a particular digital multimeter, as well as information about the location of the instrument driver and the initial configuration of the driver.

The information that follows is taken from an INI file with information for controlling a Fluke 45 using the class drivers. As you can see, the `IVILogicalNames` section of the file identifies the logical names used in your program ("DMM1" is the string you pass to the `IVIDMM_Initialize` function to begin communication). In this case, "DMM1" is actually referring to the Fluke 45 driver.

The information specifying how to communicate with a Fluke 45 is broken down into three sections of the file: the Virtual Instrument, the Driver, and the Hardware settings. The Virtual Instrument section specifies the initial settings of the IVI attributes on the drivers, such as state-caching[1] (do you want the driver to track all instrument settings in software for better performance), simulation[2] (for developing test code when the instrument is not available), and so on. The Driver section of the file contains information on where to find the specific driver for the Fluke 45 instrument. The Hardware section contains information on the physical GPIB address of the instrument, the expected ID string, and the default setup command for the instrument, as shown below:

```
[IviLogicalNames]
DMM1 = "VInstr->Fl45"

[ClassDriver->IviDmm]
Description = "IVI Digital Multimeter Class Driver"
SimulationVInstr = "VInstr->NISimDMM"

[VInstr->Fl45]
Description = "Fluke 45 Digital Multimeter"
Driver = "Driver->Fl45"
Hardware = "Hardware->Fl45"
RangeCheck = True
Simulate = True
UseSpecificSimulation = True
```

1. Refer to Application Note 122, *Improving Test Performance through Instrument Driver State Management*, for more details.
2. Refer to Application Note 120, *Using IVI Drivers to Simulate Your Instrumentation Hardware in LabVIEW and LabWindows/CVI*, for more details.

```
Trace = True
InterchangeCheck = True
QueryStatus = True
ChannelNames = "ch1"
DefaultSetup = ""

[Driver->Fl45]
Description = "Fluke 45 Digital Multimeter Instrument Driver"
ModulePath = "c:\cvi50\instr\Fl45_32.dll"
Prefix = "FL45"
Interface = "GPIB"

[Hardware->Fl45]
Description = ""
ResourceDesc = "GPIB::2::INSTR"
IdString = "FLUKE, 45, 4940191, 1.6 D1.0."
DefaultDriver = "Driver->Fl45"
```

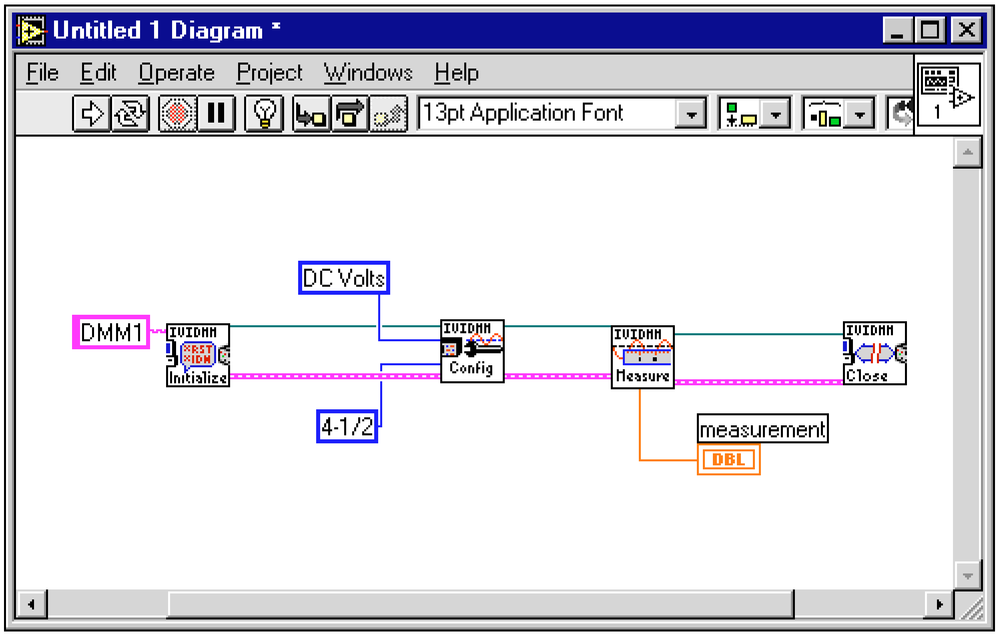
This file is updated when you install drivers on your system for new instruments. For example, if you were to replace the Fluke 45 with an HP34401 DMM, you would simply change the `IVILogicalNames` section of the INI file to specify “DMM1” to be associated with the HP34401 information in the file. T&M Explorer includes a new utility for interactively updating this `IVI.INI` file to make these changes.

Writing Your Program

Once you have the system configured, you simply make calls to the IVI class driver. Your program is completely isolated from the specific drivers for communicating with the instruments. For example, to set up and take a measurement using a DMM, you would make the following function calls from your program:

```
IviDmm_Initialize (“DMM1”, &dmmHandle);
IviDmm_Configure (dmmHandle, IVIDMM_VAL_DC_VOLTS,
       IVIDMM_VAL_AUTO_RANGE_ON, IVIDMM_VAL_4_5_DIGITS );
IviDmm_Read (dmmHandle, 5000, &reading);
```

For LabVIEW users, you have VIs for performing the same functions.



When you initialize DMM1, the class driver looks up the entry for DMM1 in the INI file, automatically finds the specific driver (in this case, the Fluke 45 driver), dynamically loads it into memory, and references the function pointers of the class driver to the corresponding functions in the specific driver DLL. From that point on, the class driver functions/VIs are simply passed directly to the Fluke 45 versions of these same functions to perform the actual instrument programming I/O with the Fluke 45.

Performance Concerns

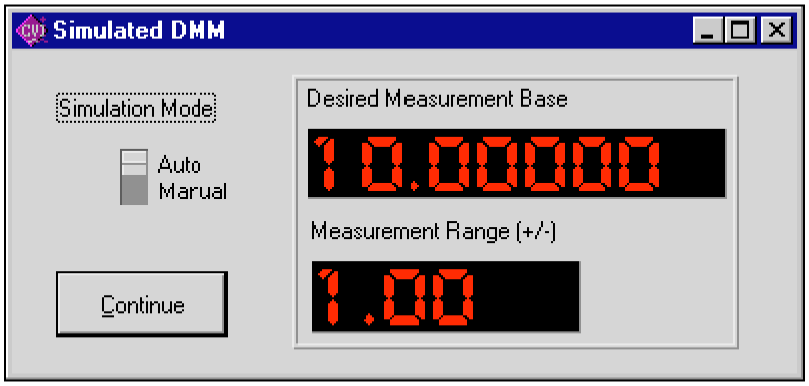
After seeing this approach initially, some users were fearful that the layered driver architecture can introduce significant overhead in the system. In reality, this approach introduces very little overhead. The compiled 32-bit C code of the class driver is packaged in a DLL that simply redirects its calls to the specific instrument driver functions. This is a very fast exchange, adding insignificant overhead to the process.

Interchangeability Checking

The class drivers have some built-in error checking to ensure that you do not develop test code that is not interchangeable. This feature is called interchangeability checking. Interchangeability checking monitors your program as it is running, looking for cases in which you do not completely specify the settings of an instrument before making measurements. Doing this can lead to incorrect operation of your program when you change instruments in the future. For example, if you initialize a DMM and immediately call `Read` in your program, you are making a measurement based on the default settings of the DMM. If you were to change DMMs, your new DMM may actually come up in a different mode upon initialization, which means your `Read` function would be taking a completely different type of measurement on this new DMM. The proper approach to this situation is to call `Configure` first, before `Read`, to configure the instrument exactly as it needs to be set up before any measurements are taken. Interchangeability checking asserts a warning whenever you make measurements that rely on the default settings of the instrument.

Other Tools Included with IVI Class Drivers – Simulation Drivers

In addition to the interchangeable instrument drivers (class drivers and specific drivers), and configuration tools for building the INI files, the IVI Driver Library includes some other utilities that will help you build your test system. Simulation drivers are “virtual instruments” that plug into a class that generate data. For example, there are five simulation drivers that are included with the IVI Driver Library – the oscilloscope, DMM, arbitrary waveform/function generator, switch, and power supply simulation drivers. Each of these simulation drivers plugs into the generic class drivers and performs flexible data generation when the drivers are used in simulation mode. Simulation drivers pop up user interface panels so developers can interactively configure the data generated. For example, when you are using a DMM driver in simulation mode with a simulation driver enabled, a data panel will be displayed whenever the `Measure` function is called. From the panel, you can select a base measurement value and an offset for the value generated. For example, generate a value of 3.0 V within a range of ±0.05 V. You can also configure the driver to pop up the panel each time the function is called, or automatically generate the data within the specified range every time.



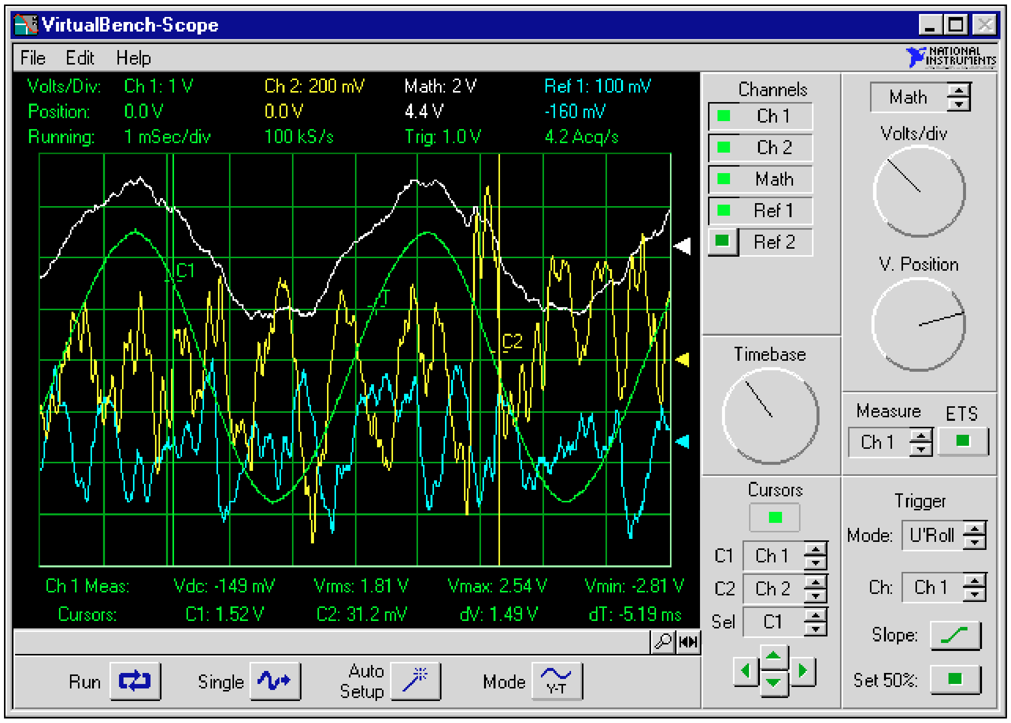
Desired Measurement Base: 10.00000
Measurement Range (+/-): 1.00

Figure 2. *DMM Simulation Driver* – When you call the `DMM_Measure` function with the DMM Simulation Driver enabled, the driver displays the measurement panel above, from which you can enter values to be "acquired" *either* manually *or* automatically, based on a specified range.

Simulation drivers are included with the IVI class drivers with source code. Therefore, you can develop very robust simulated data generation algorithms for your test systems and plug them into the simulation drivers. Because simulation drivers work with the class drivers, the code you develop can be reused without change when you swap specific instruments.

Other Tools Included with IVI Drivers – Soft Front Panels

Soft front panels are applications that enable you to interactively operate your instruments to ensure that they are operating correctly and to make simple, interactive measurements. Each class driver includes a soft front panel for controlling the instruments in your system. For example, the oscilloscope soft front panel below is an interactive interface for controlling any oscilloscope that is configured to work with the IVI class drivers. This is a very helpful troubleshooting tool when you are debugging system issues and need to take interactive measurements with your equipment. Since this soft front panel is generic for any oscilloscope that plugs into the class drivers, you only need to learn how to use it once for all scopes in your system.



Volts/Div:	Ch 1: 1 V	Ch 2: 200 mV	Math: 2 V	Ref 1: 100 mV
Position:	0.0 V	0.0 V	4.4 V	-160 mV
Running:	1 mSec/div	100 kS/s	Trig: 1.0 V	4.2 Acq/s

Ch 1 Meas:	Vdc: -149 mV	Vrms: 1.81 V	Vmax: 2.54 V	Vmin: -2.81 V
Cursors:	C1: 1.52 V	C2: 31.2 mV	dV: 1.49 V	dT: -5.19 ms

Math
C1: Ch 1
C2: Ch 2
Sel: C1
Measure: Ch 1
ETS
Trigger
Mode: U'Roll
Ch: Ch 1
Slope:
Set 50%:

Conclusion

Interchangeability has long been a goal for test engineers. The IVI class drivers delivered in the IDL ToolPak provide a standard architecture for delivering hardware-independent test systems. The architecture is built around standard programming interfaces defined by a consortium of experienced test developers, the IVI Foundation, and includes supporting tools and utilities that provide a powerful workbench of system integration tools.

341521A-01 Jul98