

How to Mitigate Hardware Obsolescence in Next-Generation Test Systems

Introduction

The problem with many test systems is that the overall system must be in operation longer than the individual system components are supported. Sometimes the device being tested has an active service life measured in decades, while many test instruments are obsolete and no longer supported after five years or less. Other times, the device being tested has an active service life measured in months. Both of these are examples of life-cycle mismatch.

A life-cycle mismatch creates a need to upgrade obsolete instruments without changing test applications, test fixtures, and devices under test (DUTs) or a need to change the test application software without changing any of the hardware or hardware-specific software. Updating test systems requires new test software development, revalidation, and redocumentation that are costly, resource-intensive, and time-consuming. To minimize the time and costs associated with migrating or upgrading test systems, you can use hardware abstraction layers (HALs) in test systems to separate the test application from the instrument hardware. This paper covers HAL architecture, best practices, features, and benefits as well as outlines examples of LabVIEW and C-based implementations.

HAL Categories

Most HALs fall into one of three groups: industry-standard, vendor-defined, and user-defined. An industry-standard HAL is defined and maintained by an industry standards body. Vendor-defined HALs are provided and maintained by a single vendor. User-defined HALs are defined and maintained by the end-users building the test systems. The majority of this paper focuses on user-defined HALs.

Industry-Standard

A well-known industry-standard HAL is Interchangeable Virtual Instruments (IVI), which is maintained by the IVI Foundation. IVI provides a standard application program interface (API) for eight of the most widely deployed instrument types. The IVI specifications have base, extended, and instrument-specific API options; range checking; simulation; and other features that make upgrading instruments easier. One IVI limit is that a user may require a function available only in the instrument-specific driver, thereby reducing interchangeability. This is because it is very difficult for the user to extend the existing IVI Class Drivers.

Vendor-Defined

A vendor-defined HAL is one in which a vendor creates a plug-in system for different instrument types and models. Vendor-defined HALs have the advantage that the vendor invests in the design, production, support, and maintenance. Limits to vendor-defined HALs include breadth and depth of instruments supported, quality, and ability to quickly and easily add new instruments – especially competitors' instruments. In addition, the vendor might focus on maximizing the performance of their instruments in the system, but not the overall system performance. Dependency on the HAL vendor's technical support could increase downtime duration and cost, especially if source code is not provided. The lack of source code limits how much you can help yourself or your ability to quickly add a new instrument. Also, the obsolescence of the vendor-defined HAL would effectively obsolete your entire test system.

User-Defined

The advantage of user-defined HALs is that you can customize them to suit your unique requirements and optimize system performance. If well-architected, a HAL would facilitate better test application development and increase reuse. It is recommended that you choose a widely used, well-supported application development environment (ADE) that is powerful enough and does not require advanced programming skills. Limits to user-defined HALs include the time and cost to design, implement, and maintain them.

Instrument-Centric and Application-Specific HALs

Each of the three HAL categories has an instrument-centric application program interface (API), an application-specific API, or a combination of both.

Instrument-Centric

An instrument-centric API abstracts instrument differences by using a common set of instrument-like function calls that unique instruments can support. For example, IVI takes an instrument-centric view of abstraction – that is, having top level test applications call an instrument-centric API that makes all instruments look similar (for example, `IviScope_ConfigureAcquisitionType`). User-defined HALs with an instrument-centric API can use “myDMM” or “standardSigGen” calls to abstract unique instruments.

Application-Specific

In an application-specific approach, the test applications call an application-specific API that is aligned with the type of tests it needs to perform (for example, LED Test). Application-specific HALs isolate you from how a particular instrument type behaves. User-defined HALs are more likely to use application-specific APIs because they can abstract instrument complexity in addition to hardware differences. Application-specific HALs facilitate separating DUT-specific parameters from reusable test logic. Also, application-specific HALs can use instrument-centric HALs to provide an additional layer of abstraction.

HAL Selection Priorities

Category Selection

Choose an industry-standard HAL if that satisfies your requirements. The investment of many companies and the stability of the standard over many years reduce your need to design, develop, and maintain it, saving you time and money. If industry-standard HALs do not meet your needs, the remaining options are vendor-defined or user-defined. By implementing your own user-defined HAL, you can pick the architecture, tools, and industry software standards that are best suited for your application. Consider vendor-defined HAL solutions very carefully because they may lock you into their technical architecture and business structure, limiting your ability to migrate faster and easier – which is the main point of HALs in the first place.

API Selection

When designing a user-defined HAL, it is important to decide whether an instrument-centric or application-specific API works best to meet your requirements. If an instrument-centric API is a better option, then define an internal common instrument-centric API “standard” that you can use across multiple types of DUTs. Break functions into three categories similar to IVI – base, extended, and specific. The most common functions for each instrument type are included in the base functions. Functions that are shared across many, but not all, instruments are grouped in the extended functions, which have a standard API if that function exists on the instrument. Finally, uncommon functions are grouped in the specific functions. Build on IVI standards where possible because it reduces time and effort to define and implement as well as migrate during system upgrades. If an application-specific API is a better option for your requirements, you need to decide which division of application functions is the most efficient to develop and easiest to reuse.

User-Defined HAL Architecture

Figure 1 shows the architecture of a user-defined HAL. The test application is at the top level. The test application developer only has to know which tests to run, the parameters for those tests, and the test hardware requirements (sampling rate, resolution, and so on). When designing the test, the developer does not need to know the specific instruments. The test application in the LabVIEW example, named “Main Test App.vi,” calls functions from the HAL. It is recommended to have two levels in your HAL. The example HAL has an application separation layer (ASL), which the test application calls into, and a device-specific software plug-in (DSSP) that the ASL calls into.

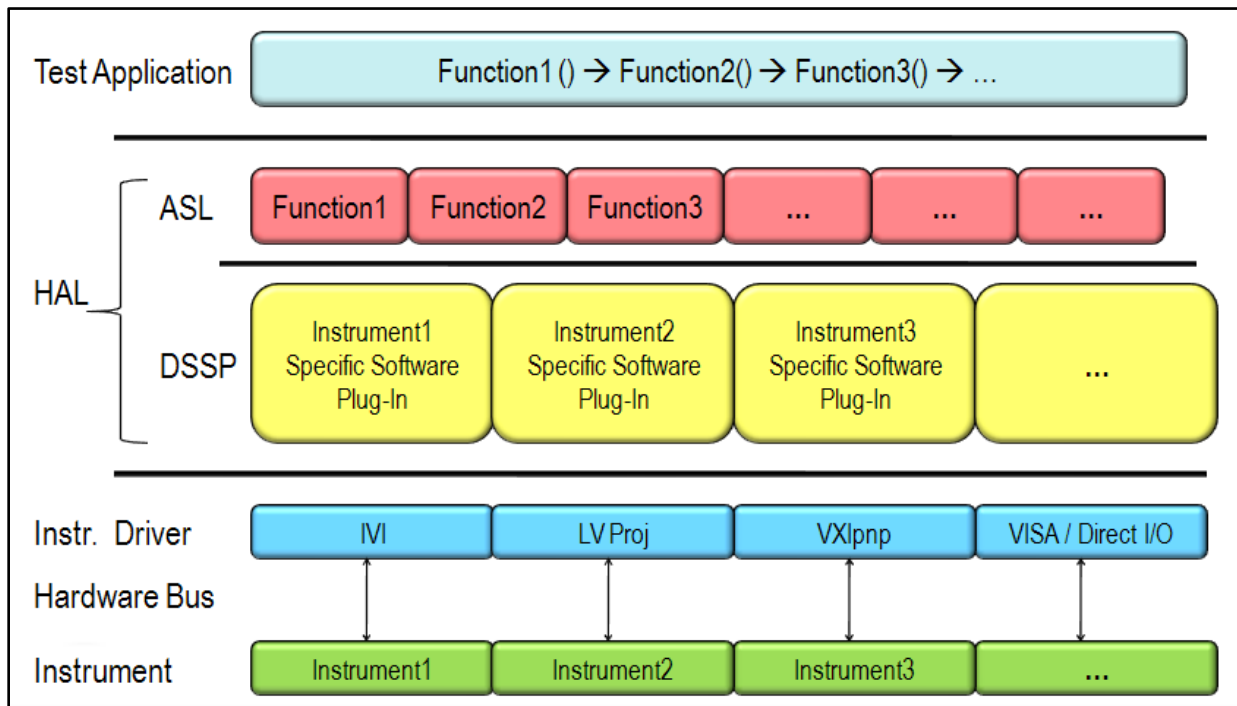


Figure 1. User-Defined HAL Architecture

Application Separation Layer (ASL)

The ASL provides application-specific interfaces, such as `InitAllDevices.vi`, `FreqSweepTest.vi`, and `LED Test.vi`. Because the ASL API is application-specific, it does not expose details about the instrument-centric operations that need to be performed. These interfaces are higher level, more reusable, and easier to maintain. The ASL functions call into the DSSP layer.

Device-Specific Software Plug-In (DSSP)

The DSSP provides device-specific modules that manage the differences between instruments including driver type, driver interfaces, available functions, scaling, timing, and formatting. The DSSP instances either use existing instrument drivers or communicate directly with the instrument. Using Virtual Instrument Software Architecture (VISA) is recommended to abstract the hardware bus and OS-specific hardware communication. The DSSP layer makes the hardware interchangeability possible.

HAL Best Practices

Separate Test Logic from Common Test Functions

An important best practice is to separate DUT-specific test logic from more general, reusable test modules to improve reuse and reduce revalidation and documentation costs. Decoupling and making layers modular improves system architecture so the layers are easier to develop and maintain. For example, separate the DUT-specific test limits (not reusable) from the actual limit test implementation (reusable). Another reason to place the more general test code in the higher-level ASL is to reduce the amount of duplicated code.

Separate Common Test Functions from Instrument Drivers and Hardware

Separating common test functions from instrument drivers and hardware allows the ASL software module to get the same measurement results. It is also valid to add more code to a DSSP instance to make an instrument behave similarly to other instruments. For example, if most of your digital multimeters (DMMs) can implement multipoint measurements and you need to use a DMM that is capable only of single-point measurements, you can add looping code into the DSSP implementation level of the single-point DMM to make it behave like a multipoint DMM from the perspective of the ASL.

Separate Test System Parameters from Test Logic

Another best practice is to separate test system parameters from test logic. This reduces the hard-coded dependence on instruments or specific DSSP instances. It also makes it possible to compile the test application into an executable because you do not have to change the code to modify the test hardware.

Design for Dynamic or Static Interchangeability

When designing an application-specific HAL, you can choose between dynamic or static interchangeability. Dynamic interchangeability means that you select the specific DSSP implementations during run time. In this case, you use an XML configuration file that identifies the hardware resource name and LabVIEW class implementation of the DSSP parent class. Using LabVIEW classes forces you to have the same connector pane (similar to function parameters) in the implemented virtual instruments (VIs), which are similar to functions.

Static interchangeability means having your functions contain the same function parameters but not going to the extra effort of being able to interchange them at run time. A common way to statically interchange your functions is “find and replace.” The main disadvantage of static interchangeability is the effort required to replace the functions. Either type of interchangeability is going to require effort either at development time (dynamic) or after deployment (static).

HAL Implementation Example

This example features a user-defined HAL and instruments from four different manufacturers. It shows a system that was designed and implemented using the best practices described in this paper. LabVIEW is used for its simple graphical system design capabilities that are optimized for test software development, large selection of native instrument drivers, and object-oriented programming (OOP). This example uses OOP because it is an existing framework that ensures consistent interfaces among the child classes. More importantly, it offers dynamic interchangeability, so the only element you need to change is the configuration file. You can implement the same example in other ADEs and using mechanisms other than OOP to implement dynamic interchangeability.

To learn how to design HAL in a C-based ADE, refer to “Appendix A - C Based HAL Implementation Example” in this document.

A simple system best highlights the user-defined HAL design principles and best practices; however, you can apply the same principles and practices to a more complex system. Dynamic interchangeability was chosen because it makes system use easier in the future. Also, choosing the type of interchangeability that requires more upfront effort and coding adds more value to the example.

The example system uses a Fluke DMM, Agilent function generator, Tektronix oscilloscope, and an NI PXI switch to test a DUT with a low-frequency, low-pass filter, and an LED. The system is then migrated to NI PXI modular instruments to demonstrate the HAL. Figure 2 shows the top-level user interface to the test application.

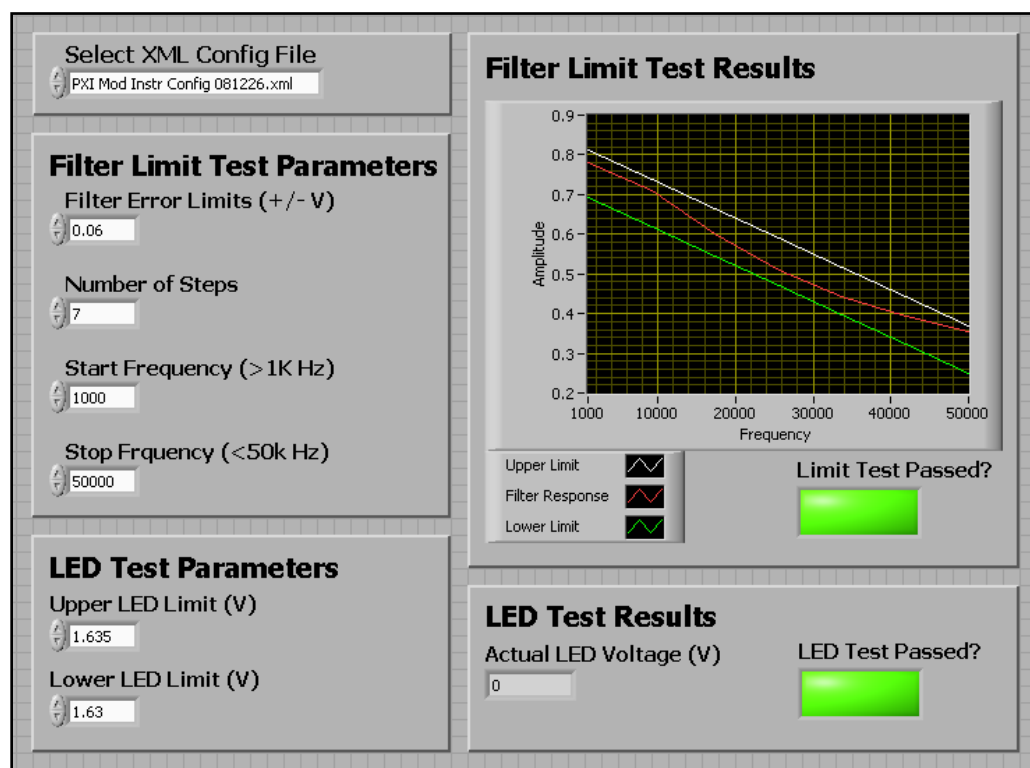


Figure 2. Parameters and Results for Filter and LED Tests Front Panel

The example top-level test application (Figure 3) calls five VIs:

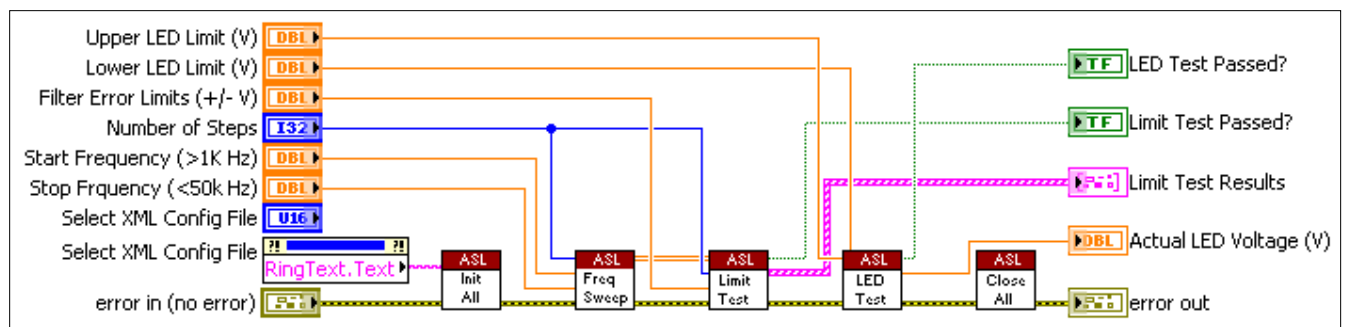


Figure 3. Main Test Application

1. **Initialize All Devices** – InitializeAllDevices first configures the switch to test DUT 1 and then reads the XML configuration file to determine the instrument types, hardware resource names, and DSSP class paths. It then dynamically loads the corresponding DSSP classes and calls the Initialize Device instance of each DSSP class. Each Initialize Device opens the hardware resource session and resets the instrument.

InitializeAllDevices saves the class references in global variables that are used in the other VIs.

2. **Frequency Sweep Test** – The low-frequency, low-pass filter is tested using the FrequencySweepTest ASL VI. The ASL VI calls two main DSSP functions in a loop: Generate Signal and ReadMaxAmp. The Generate Signal VI generates a selectable waveform with variable frequency and amplitude. When implementing the frequency sweep, a sine wave of constant amplitude is used and each time the Frequency Sweep Test loop is executed, the frequency of the sine wave is increased until the stop frequency is met.

During each iteration of the loop, the ReadMaxAmp DSSP VI measures the maximum amplitude that is being output by the filter of the DUT. The output frequencies and measured amplitudes are stored in an array that is passed to the Limit Test function.

3. **Limit Test** – The Limit Test VI takes the raw data from the FreqSweepTest VI and uses the filter error limits to calculate if the filter response data ever went out of bounds. If it did not, then the limit test passed and the VI outputs a “True” from the “Test Passed” indicator. The Limit Test VI also outputs a waveform graph showing the results of the frequency sweep test and the limits.
4. **LED Test** – The LED on the DUT is tested by applying a current and measuring the voltage across the LED. This is accomplished by using the Diode Test function of the digital multimeter (DMM). If the resulting voltage is within the limits, the test is passed and a “True” is passed to the “LED Test Passed?” indicator. The LED Test VI also outputs the actual voltage that was measured.

After the LED Test is completed, all of the references are closed by CloseAll.vi.

5. **Close All Devices** – This VI does not have any inputs or outputs other than the error clusters. It uses the DSSP class global variables to call the Close VIs for each DSSP class that is being used. The individual Close VI of each DSSP then takes care of closing the resource session for the specific instrument.

Initial System Configuration

As shown in Figure 4, the test application is running on an NI PXI embedded controller with Windows XP. The PXI controller is connected to the Agilent 33220A signal generator through LAN, to the Tektronix TDS2024 oscilloscope over USB, and to the Fluke 8845A DMM over GPIB. The NI-SWITCH is controlled over the PXI bus in the PXI chassis. The test fixture is made from commercial-off-the-shelf (COTS) products. A low-pass, low-frequency filter and LED are on the DUT.

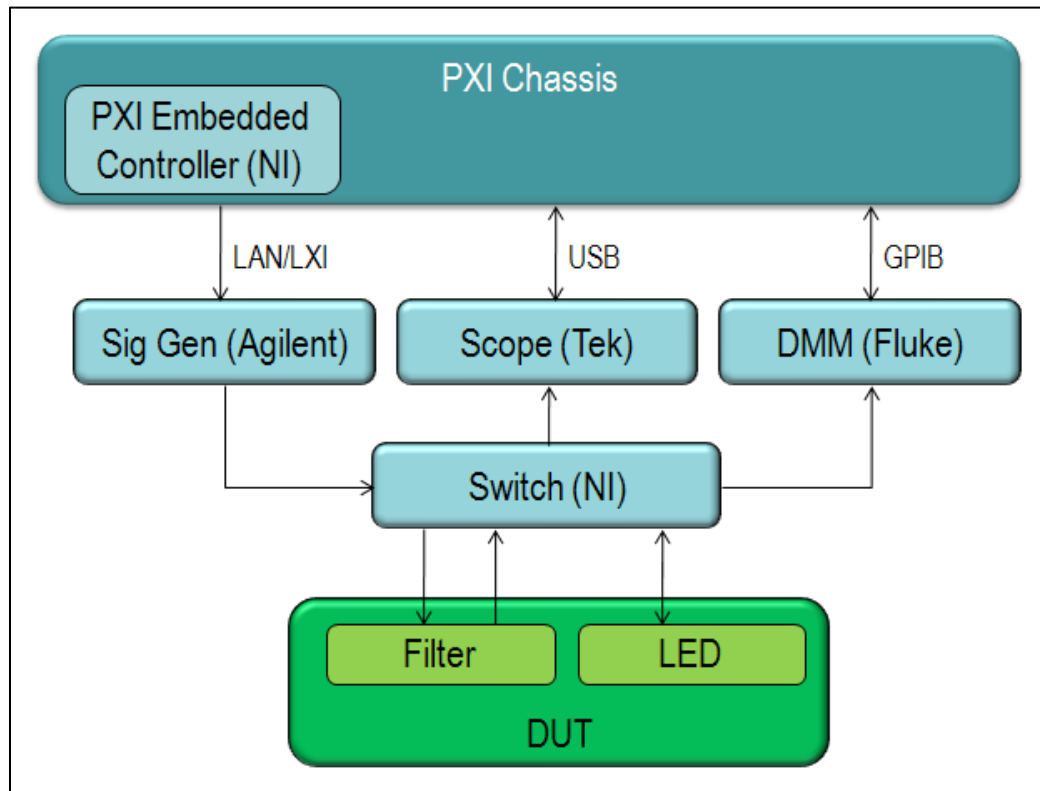


Figure 4. Hardware in Implementation Example

On top of Windows XP, the application uses NI-VISA to call into the bus specific code (NI-488.2 for GPIB, Windows USB stack, Windows TCP/IP stack). Because NI-VISA will support new instrument control buses in the future, you do not have to worry about upgrading to models that incorporate these new bus technologies. Also, just like LabVIEW, NI-VISA is OS independent, so you can move it onto any other OS that NI supports.

To call into NI-VISA, the application uses either LabVIEW Plug and Play instrument drivers or IVI drivers. The IVI drivers require the IVI Compliance Package. NI-VISA includes NI Measurement & Automation Explorer (MAX), which is used to establish communication with the instruments and configure the IVI Class driver sessions necessary to use the interchangeability feature of IVI. It would also have been feasible to call directly into VISA from the DSSP implementation (direct I/O), but that is recommended only if an instrument driver is not available.

HAL Implementation Software Architecture

Figure 5 shows the HAL architecture with specific functions listed and the instrument models displayed. The switch is initialized and closed directly in ASL by using the NI-SWITCH instrument driver. Though less than ideal, this illustrates that sometimes interchangeability is a higher priority for certain instrument/measurement types. Deviating from ASL-DSSP convention leaves room for future improvements. The ability to add switch functions to the DSSP and create DSSP modules for switches shows how you can extend the HAL.

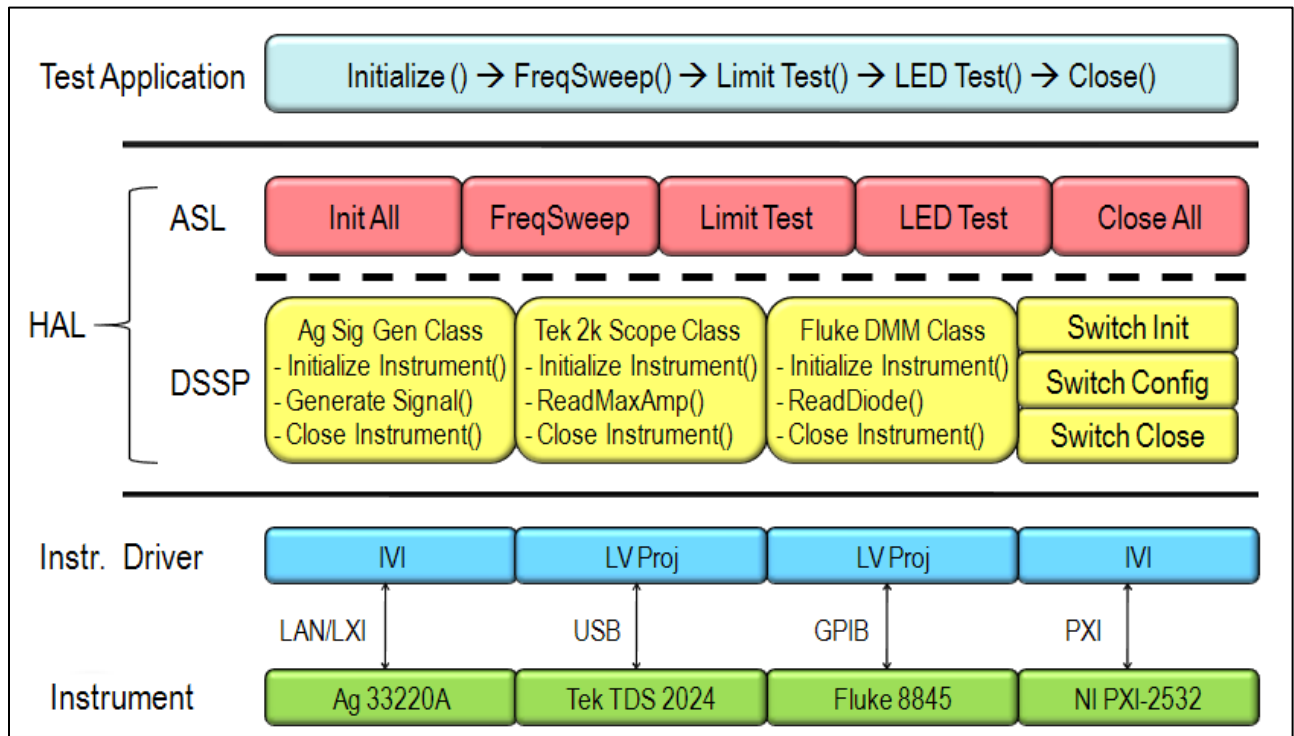


Figure 5. Software in Implementation Example

Test code and ASL code in this implementation are standard VIs, not LabVIEW classes. Only the DSSP layer uses LabVIEW classes (OOP) to allow for dynamic interchangeability and ease development efforts by creating a standard set of DSSP functions. The dashed line in Figure 5 shows where the interchangeability occurs.

Figure 6 shows the project explorer view of the HAL implementation in LabVIEW. Private data that can be accessed only by the class is represented as a control (.ctl file) and requires Data Member Access VIs such as “Read Instr Resource Str.vi.”

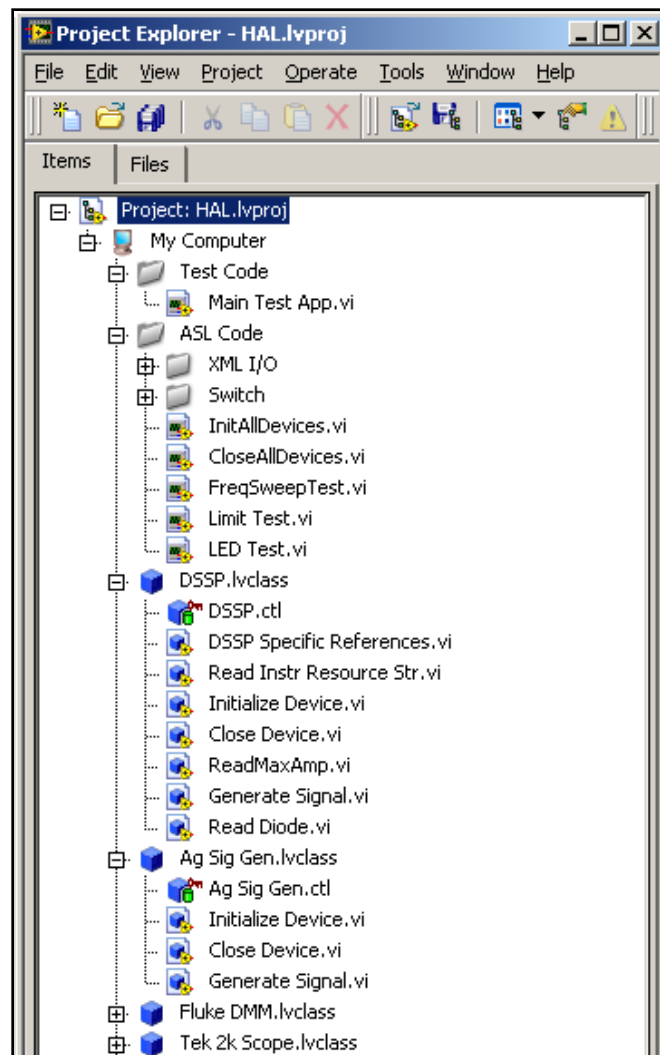


Figure 6. Project View of HAL Implementation in LabVIEW

DSSP Class Definition

The DSSP parent class definition has three different types of functions: parent-only, common, and measurement-based. The DSSP parent class, DSSP.lvclass, contains a superset of all the child class functions plus some functions unique to the parent. A single child instance of the parent DSSP class (for example, Ag Sig Gen.lvclass) has to contain the common DSSP functions and at least one of the measurement-based functions.

Table 1 shows the different DSSP functions and their categories.

Category	Function Names
Parent-Only	DSSP Specific References Read Instr Resource Str
Common	Initialize Device Close Device
Measurement-Based	ReadMaxAmp Generate Signal Read Diode

Table 1. DSSP Function Categories

Table 2 shows which child classes implement parent DSSP functions.

Function	Agilent Sig Gen	NI Sig Gen	Tektronix Scope	NI Digitizer	Fluke DMM	NI DMM
Initialize and Close	✓	✓	✓	✓	✓	✓
ReadMaxAmp			✓	✓		
Generate Signal	✓	✓				
Read Diode					✓	✓

Table 2. DSSP Function Locations

In the example implementation, only the parent DSSP class contains all of the functions that are needed in the test instrumentation. This is possible because the example is not very complex. For a more complex system, you would most likely want to have multiple DSSP definitions to keep the number of functions in each one down to a manageable level.

Figure 7 shows the “simple” implementation and Figure 8 shows the “complex” implementation suggestion.

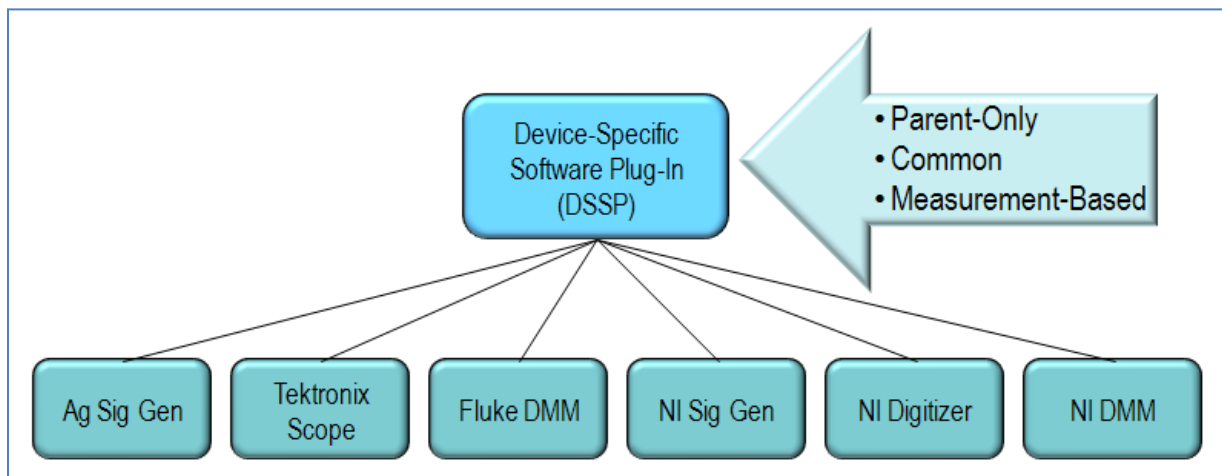


Figure 7. Simple Single-Tier Class Hierarchy in Example Implementation

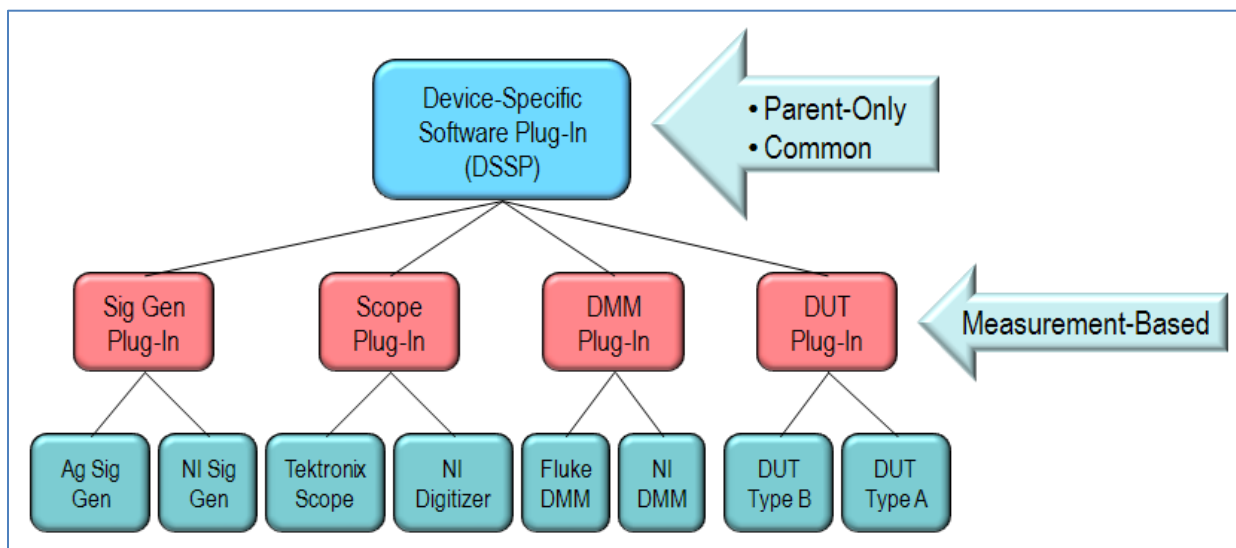


Figure 8. Recommend Multiple-Tiered Class Hierarchy in Complex Systems

The main difference between the two DSSP hierarchies is that the measurement-based functions in the complex system are separated into another level of subclasses or virtual folders. This makes the development and maintenance even more manageable for larger systems.

Refer to Figure 9 to see that both the Ag Sig Gen and NI Sig Gen child classes implement the same functions because they are filling the same role.

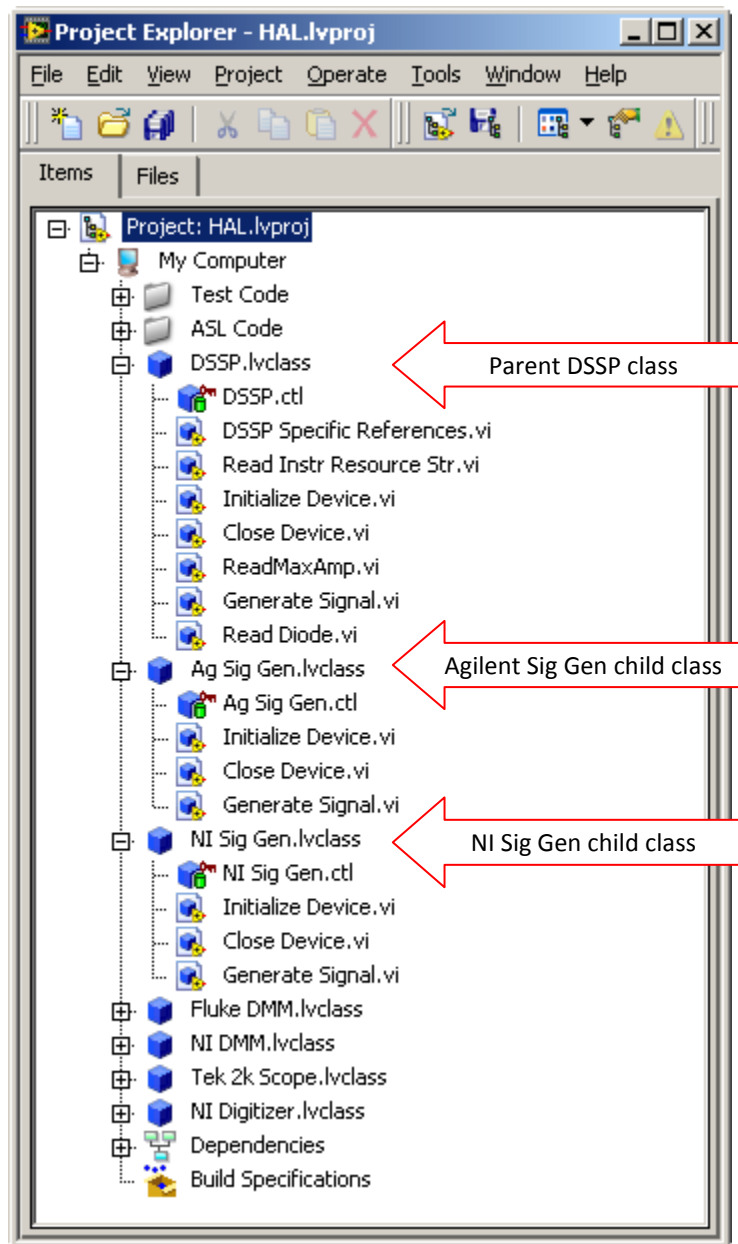


Figure 9. Agilent and NI Signal Generators (inherit same subset of DSSP methods)

Migrated System Configuration with New Hardware

Now update the system's hardware. Figure 10 shows the updated hardware.

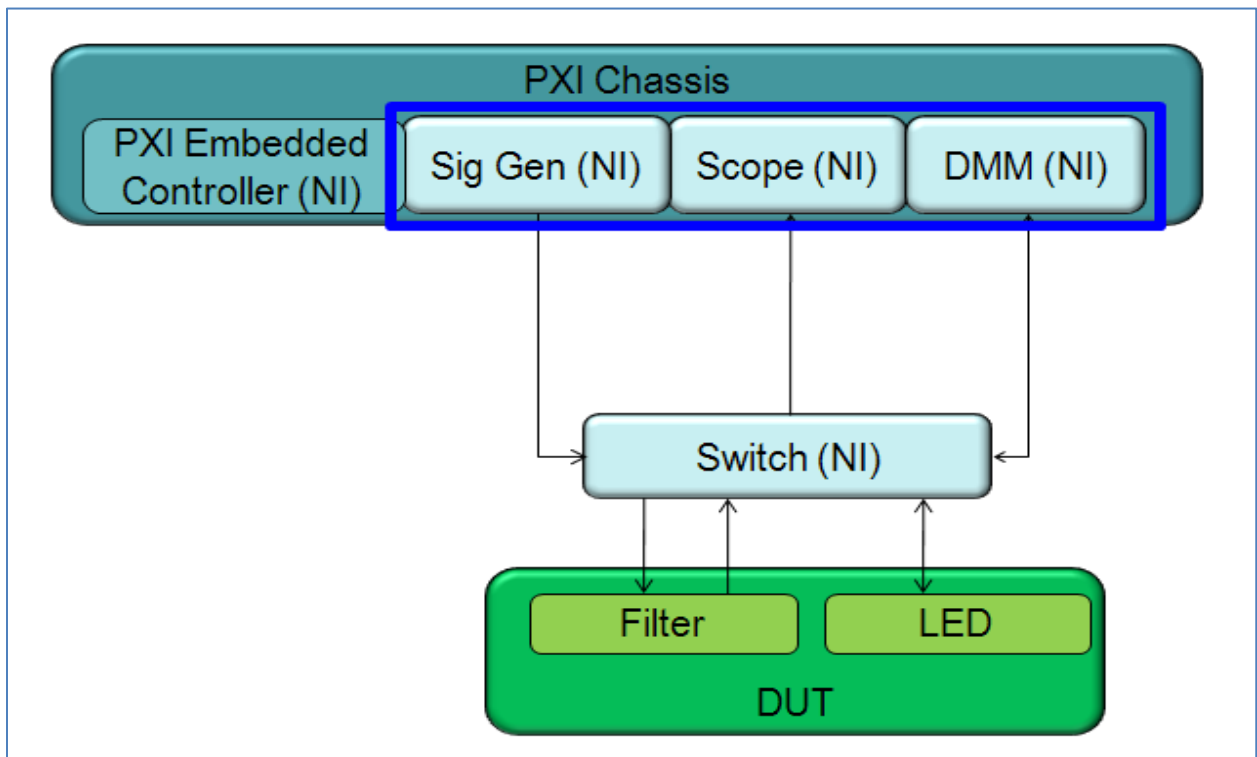


Figure 10. Updated System Hardware (no changes to DUT or fixture)

The system transitions from using four different instrument control buses (USB, LAN, GPIB, PXI) to using only PXI. Figure 11 shows the updated software.

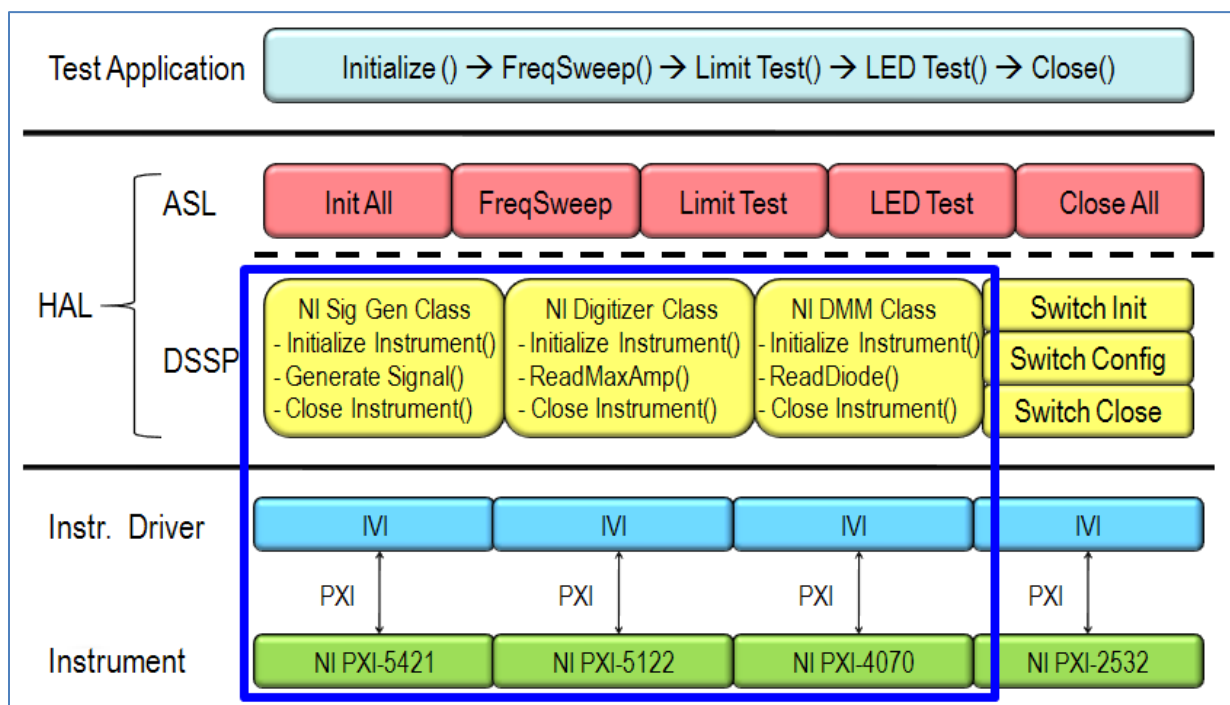


Figure 11. Updated Software (provides same measurement result using new DSSP)

The blue box in Figure 11 shows that you had to develop and test three new DSSP modules for the three new instruments. In this case, you did a one-to-one replacement of instruments. Refer to figures 4 and 5 for a comparison to the original system. The HAL prevented you from having to rewrite the test application.

Steps to Replace Instruments

Performance Requirements

The important consideration to make when replacing instruments is that the replacements must meet your requirements, usually by having equal or better measurement functions. For example, a replacement DMM should have an equal or better resolution and sampling rate (among other things).

Configuration

After you have procured suitable instruments, you need to update the configuration file with the hardware resource information. Usually this will require you to configure the instruments in whichever utility you use (MAX is used for NI-VISA and NI modular instruments).

DSSP Development and Testing

After you update the configuration file, you should start developing and debugging the new DSSP modules for the new instruments. It is recommended that you replace only one instrument at a time so you can debug your new DSSP with the existing “proven” DSSPs. Either way, you should use a “golden DUT” to determine instrument differences and compensate for them in the new DSSP module. Through the course of developing and testing the replacement DSSP modules, you start out by testing only one new module (and the accompanying hardware) to see if there are any differences for which the DSSP needs to compensate. In this example, the testing exposes a scaling difference between the Tektronix scope measurements and the NI digitizer measurements. The solution is to add linear scaling to the NI digitizer DSSP module as shown in Figure 12.

When updating the test hardware, and the corresponding DSSP modules, you do not necessarily have to do a one-to-one replacement. For instance, if you have a signal generation function and a DC voltage measurement function currently implemented with two separate instruments, the replacement instrument could be “synthetic” and implement both signal generation and DC voltage measurement. This is one of the advantages of an application-specific API and having two layers within the HAL.

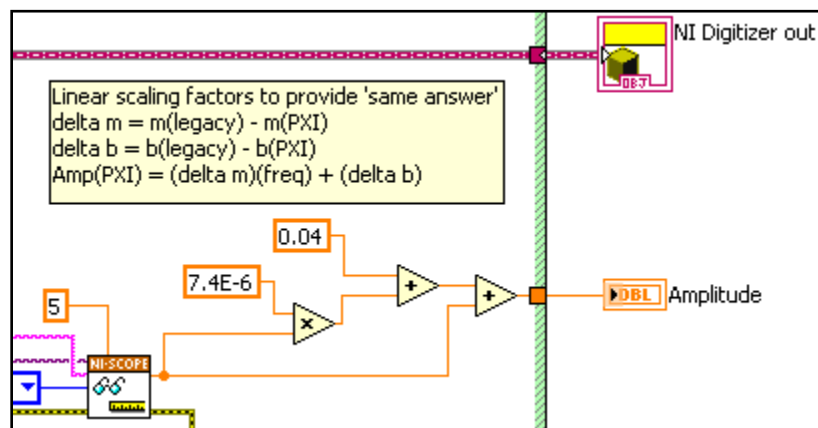


Figure 12. Linear Scaling in NI Digitizer DSSP Module

Only the instruments, the DSSPs, and the XML configuration file changed. The top level test application and test results did not change. The main benefits of this hardware replacement process are that the top level test application and the test results were not altered. Also, once an instrument has been integrated into the HAL, it can be interchanged and the DSSP can be reused by other test applications and extended if necessary.

HAL Features

HALs are useful because of the features that they implement. Table 3 lists the main features of HALs with a short explanation of each.

Feature	Description	Implementation	Example
Test Code and Hardware Separation	Software layers that separate the test application from the instruments and DUTs	HAL with two layers: ASL and DSSP	LED Test.vi for ASL and Ag Sig Gen.lvclass for DSSP
Measurement Compensation	Measurement results vary between instruments, and should be compensated for to give the test application the same measurement for the same actual value on the DUT	Compensate in DSSP module because it is “device specific”	Linear scaling in <i>NI Digitizer DSSP</i>
API Translation	Translates APIs for different driver types	DSSP modules translate different instrument driver APIs to a common ASL interface	Function parameter for diode measurement was “9” on Fluke and “6” on NI DMM
Adjustment for Timing Differences	Some newer instruments may respond faster than the test code can consume the result	Add delays in DSSP	No example in this implementation, but overall test time was reduced from 26 s to 3 s
Data Formatting	Different instruments expect and return different parameter formats	DSSP modules change data formats	- Scientific versus floating point - V versus mV - Degree C versus Degree F
Backward Compatible and Extensible API	Internal ASL standards maintain backward compatibility while extending to meet new requirements	Define internal API standards that are only extended, not deprecated – this allows an existing application to use an updated ASL	Use standard values for DMM measurements: 0 always means DC V, 1 always means AC V, and so on; extend DMM measurements by adding a new value
Error Filtering	HAL can check and respond to errors before passing results to test application	Write additional error handling code in either ASL or DSSP modules	If DSSP modules are not loaded, the parent DSSP class returns errors/warnings
Instrument Simulation	Provide simulated instrument results without having actual instrument	Develop DSSP with simulation software	
Fault Insertion	Inserting known faults can check software in higher layers	Fault insertion in DSSP can check ASL; ASL can check test application	

Table 3. Hardware Abstraction Layer Features

HAL Benefits

When addressing obsolescence, HALs yield the benefits of lower migration costs, faster migration time, higher code reuse, and easier maintainability.

Lower Migration Costs

The act of designing a user-defined HAL helps you examine thoroughly your test system requirements and overall design. By modularizing your test system software into different levels, you avoid the extensive test application redevelopment that is necessary in a monolithic (nonmodular) test application. In addition to minimizing redevelopment, a user-defined HAL reduces the redocumentation and revalidation burden, which can translate directly into lower migration costs.

Faster Migration Time

The lower migration costs are a direct result of decreasing the migration effort required. The decrease in effort also allows the migration process to go more quickly. Having a well-designed and well-documented explanation of your HAL requirements allows developers to work on new DSSP code while the existing DSSP layers are used in the “production” (or released) system. This means you are better prepared for a test hardware migration than you would be with a traditional monolithic test application.

Higher Code Reuse

When you start with future reuse in mind, you are already ahead of the curve when it comes to system architecture. Following the best practice of separating test logic from the ASL minimizes dependencies and maximizes reuse. This greatly reduces the overlap of responsibilities and redundant development efforts. For example, the limit test uses an upper and lower limit with a linear mask on a frequency/amplitude table (array). This can be reused by any instrument where a linear mask is tested by a 2D table (array). Isolating your ASL from instrument-specific code also keeps you from getting locked into a certain instrument model or manufacturer.

Easier Maintainability

The modularity of a user-defined HAL makes isolating and fixing bugs easier. In addition, you can focus any revalidation efforts on the module that was fixed, not the entire test application. When you design a user-defined HAL, you must document the requirements for each part, which, in turn, allows you to know what to test each part for. In many cases, the act of defining and designing your HAL exposes flaws in your original assumptions, which facilitates better system architecture. FreqSweep.vi in the ASL contains the looping code that executes a frequency sweep. This simplifies the requirements of the DSSP functions you need and gives you the power to use instruments that do not have a built-in frequency sweep capability. It also simplifies the timing between the output of the waveform at a certain frequency and the measurement of the maximum amplitude of the DUT at that frequency.

Summary

HALs separate the test application from the instrument hardware and hardware-specific software to streamline the upgrade process. The streamlined upgrade process minimizes the time and costs associated with migrating test applications. By isolating the software you have to modify, you reduce the cost and effort of requalifying code. The other case of migrating existing hardware to a new software application also benefits from HAL use. In addition to the separation of test code and hardware, the main HAL features are measurement compensation, API translation, and a backward-compatible and extensible API.

HAL options include standards-based (for example, IVI), user-defined, or vendor-defined and either an instrument-centric or application-specific API. You must weigh the different options and choose the most appropriate HAL type and API option. Because most test systems require a major upgrade over their useful lives, a HAL reduces the time and costs of migrating test software to new hardware by minimizing the impact of new test software development, revalidation, and redocumentation.

Appendix A - C-Based HAL Implementation Example

The C-based HAL outlined in this appendix takes advantage of the IVI-C architecture and the NI IVI Compliance Package (ICP) implementation framework to provide the same benefits as the LabVIEW OOP-based HAL. ICP is a distributable product that provides the software components needed to build interchangeable applications. ICP is available without charge on ni.com.

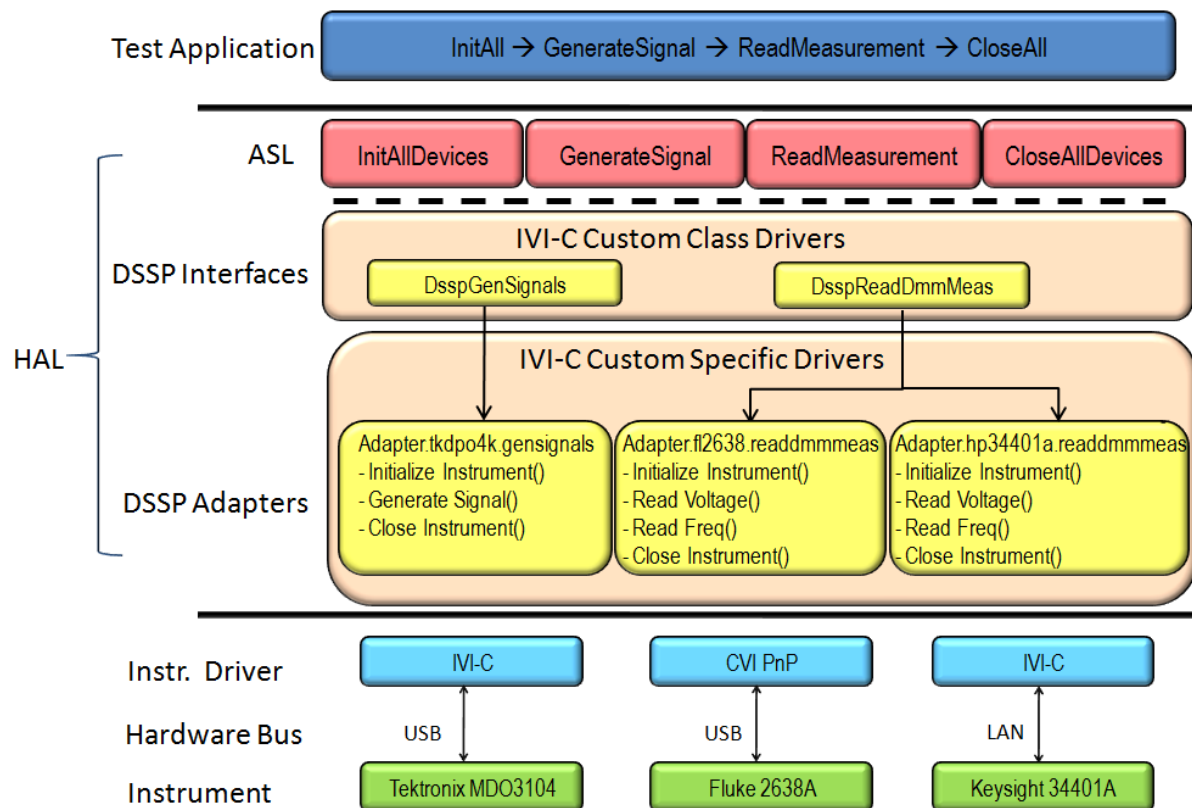
The HAL implements an interchangeable and configurable layer for high-level application operations. This layer is similar to the IVI standard instrument classes, which define interchangeable and configurable layers for device-level operations.

While IVI-C is an architecture defined by the IVI standard for implementing interchangeable and configurable layers for device-level operations, the IVI-C architecture and ICP are sufficiently generic so that you can use them to implement interchangeable and configurable layers for high-level application operations.

Building a HAL with IVI Compliance Package (ICP)

The following figure shows a detailed instance of the HAL architecture with specific functions listed and the instrument models displayed.

This example uses the **IVI-C Custom Class Driver** and **IVI-C Custom Specific Driver** concepts to implement components of the DSSP layer. Both concepts are explained in IVI-3.1: Driver Architecture Specification, Section 2.2. The configuration information is stored in the **IVI Configuration Store**, which is modifiable using MAX. This concept is explained in IVI-3.5: Configuration Server Specification, Section 2.4.



From the bottom layer up:

- This example includes three instruments. All three have pre-written device-level instrument drivers: two with IVI-C drivers and one with a LabWindows™/CVI™ Plug and Play driver. You can use the HAL

with any type of device-level programmatic interface, such as IVI, LabWindows/CVI Plug and Play, direct I/O, or other. The drivers are available without charge on ni.com/idnet.

- The DSSP layer includes three **IVI-C Custom Specific Drivers**. Each custom specific driver calls one of the device-level instrument drivers to implement the application-level functions required by the DSSP layer. Thus, the custom specific drivers play the role of the DSSP adapters.
 - In general, the APIs of the custom specific drivers are much smaller than the APIs of the device-level instrument drivers, because the custom specific drivers need satisfy only the needs of the specific application rather than all possible uses of an instrument.
 - The prefix of the custom specific driver must be more specific than the prefix of the device-level instrument driver. A qualifier that indicates the high-level operation should be appended to the instrument prefix. In the example, the *fl2638a* LabWindows/CVI Plug and Play driver is called by the *Adapter.fl2638.readmmmeas* custom specific driver.
 - The signatures of the functions exported by the custom specific driver must be 100% compliant with the DSSP Interface layer.
 - When replacing an instrument, you must write a new custom specific driver for the new instrument. The new custom specific driver must have the same API as the previous driver. The prefix of the new custom specific driver should include the same qualifier as the prefix of the replaced custom specific driver.
 - You can create custom specific drivers using the IVI Instrument Driver Wizard (**Tools»Create IVI Specific Driver or Tools»IVI Development»Create IVI Specific Driver**) in LabWindows/CVI.
- **IVI-C Custom Class Drivers** have only two custom class drivers because one of them, *Dssp.Interface.ReadDmmMeas*, is used for two separate IVI-C Custom Specific Drivers and physical instruments.
 - Using the IVI-C architecture, the custom class drivers dynamically load the custom specific drivers.
The custom class drivers for a particular application do not change when replacing instruments. Instead, the custom class drivers just call into different custom specific drivers. Use MAX to update settings in the IVI Configuration Store. (Refer to the following section about the IVI Configuration Store for details).
 - Consequently, instrument replacement using the HAL is done without changing the application source code or having to re-link the application program.
 - Always include the Initialize and Close functions in the custom class drivers.
 - The DSSP Interface can be created via IVI Custom Class Driver Wizard (**Tools»Create IVI Custom Class Driver**) in LabWindows/CVI.

Using the IVI Configuration Store

The **IVI Configuration Store** is an XML configuration file that is modifiable using MAX. For IVI-C based HALs, use the IVI Configuration Store to specify which custom specific driver you want to use in your application.

- In the IVI Configuration Store, each Logical Name is configured to point to a **Driver Session**. The Driver Session specifies the physical instrument to be used, the custom specific driver to be used, and also the configuration of certain optional parameters to be used when the device-level driver happens to be an IVI-C driver.
- To specify the custom specific driver, the Driver Session points to a **Software Module**. The software module specifies the DLL name (Module Path), prefix excluding qualifier (Prefix), and qualifier (Published API) of the custom specific driver.

- Using the Fluke 2638A instrument in the example, the IVI Configuration Store needs to include:
 - A Logical Name that corresponds to the role that the Fluke 2638A (or a replacement instrument) plays in the application.
 - The Logical Name should point to a Driver Session with a name such as *adapter.fl2638.readdmmmeas-session*.
 - The Driver Session should point to a Software Module with a name such as *adapter.fl2638.readdmmmeas-module*, and should specify the physical instrument (e.g., “GPIB0::3::INSTR”) using the **Hardware Asset** setting.
 - The Software Module should specify the DLL name (for example, *adapter.fl2638.readdmmmeas_32.dll*, assuming that it is default IVI location for driver DLLs), the prefix excluding the qualifier (*fl2638*), and the qualifier (*readdmmmeas*).
- When replacing an instrument, you should have already set up the Driver Session and Software Module for that new instrument and new driver. All you need to do is to change the Logical Name to point to that new Driver Session.

Now that you are familiar with the C-based HAL architecture, you should download and run the example (Hardware Abstraction Layer – C Example). Refer to the document “How to Run the Example” and follow the steps. Additionally, refer to the document “Implementing a C-based Hardware Abstraction Layer (HAL)” for instructions about implementing and expanding the example.