Using Abstraction Layers to Build Reusable, Maintainable Test Software for Aerospace and Defense Applications

Software intellectual property is the single most strategic asset to your test engineering team. According to the Defense Science Board, "many of the capabilities provided by our weapons systems are derived from the software of the system, not the hardware. This shift from hardware-enabled capabilities to software-enabled capabilities is increasing quickly." This applies to most modern commercial aviation technology, as well as defense and government fleets. Test systems are becoming more dependent on software, which makes these systems more flexible and extensible, but introduces development and maintenance challenges.

Aerospace and defense test organizations building new test capability must continue to meet tight schedule requirements, which means retaining a large team of in-house software engineers to create new test software for each program, contracting test partners to deliver test capability, or identifying a more effective way to use commercial off-the-shelf software to build a reusable architecture.

Software is such a valuable part of mission-critical systems that it remains highly-controlled and requires rigorous testing. But maintaining test software to ensure test readiness and conform to mandates can be laborious and expensive. Recertifying test software with the organizations and standards bodies that oversee it is especially difficult when the software is built in a large, monolithic structure.

The best way to increase reusability and reduce the time to modify or recertify test code is to build test software architectures in small, dedicated layers. The software that manages users and test sequences should be separate from test code modules, which should not include the code that sends settings and commands to the hardware. This way, test teams can repair and upgrade each layer or module individually while isolating other layers and maintaining the same inputs and outputs. The NI software platform includes programming environments and application software ideal for building tiered software architecture.

"By leveraging commercial off-the-shelf software technologies such as TestStand and LabVIEW, our Automated Test Engineering team was able to achieve up to 60 percent code reuse for each automated test and reduce overall test development time by up to 9X. Using the NI certification program, coupled with in-house training, we were able to develop the skills necessary to produce robust, flexible code and maintain it across over 200 test benches."—**L3 Technologies**

Abstraction layers are critical tiered software architecture components that translate each test software interlock. Read more to learn how abstraction layers increase code reuse and reduce certification costs.

NATIONAL INSTRUMENTS™

FUNDAMENTALS OF BUILDING A TEST SYSTEM

Hardware and Measurement Abstraction Layers

Grant Gothing, ATE R&D Manager, Bloomy Controls

CONTENTS

Introduction

The design and development of automated test equipment (ATE) presents a host of challenges, from initial planning through hardware and software development to final integration. At each stage of the process, changes become more difficult and costly to implement. Furthermore, because software typically follows hardware in the development cycle, many open-ended items are left for the software engineer to handle. Good planning goes a long way toward mitigating familiar risk, but it can't prevent every problem, especially in a fast-paced test development cycle where many issues arise at final integration. The idea that the software is more malleable than hardware, results in the phrase "just fix it in software!" However, hardware and software are tightly coupled and most issues typically require updates to both. This doesn't stop with the initial deployment, but continues for the system's life cycle.

As products get more complex, so do the systems required to test them. ATE instrumentation costs become important, so the ability to reuse instrumentation across several products is often a necessity. Furthermore, shortened development times require hardware and software to be developed in parallel, usually with poorly defined requirements. Then, once deployed, long product life cycles mean that failing or obsolete instruments, as well as product and test requirement changes, could produce more challenges for test equipment. Because of this, modularity, flexibility, and scalability are critical to a successful automated functional test system.

From a hardware standpoint, this is typically accomplished by using modular instrumentation and interconnects with interchangeable test fixtures. But how can you make the test software as adaptable as the hardware? Hardware abstraction layers (HALs) and measurement abstraction layers (MALs) are some of the most effective design patterns for this task. Rather than employing device-specific code modules in a test sequence, abstraction layers give you the ability to decouple measurement types and instrument-specific drivers from the test sequence. Because test procedures are typically defined using types of instruments (such as power supplies, digital multimeters [DMMs], analog outputs, and relays) rather than specific instruments, employing abstraction layers results in a test sequence that is faster to develop, easier to maintain, and more adaptable to new instruments and requirements. By using hardware abstraction to decouple the hardware and software, you can drastically reduce development time by giving hardware and software engineers the ability to work in parallel. The development of common APIs for sequence and low-level code implementation allows a system architect to maintain a repository of common functions, promoting standardization and reusability. This makes it possible for test developers to focus on the individual unit under test (UUT) sequence development and spend less time writing low-level code.

ATE SOFTWARE CHALLENGES

DEVELOPMENT	MAINTENANCE
Rushed development cycle Poorly defined requirements Evolving test procedure Software development begins before hardware design is complete Separation between software and hardware engineers	Long product life cycle ▪ Failing or obsolete instruments ▪ Instrumentation changes Product updates ▪ Test procedure changes ▪ New hardware required Manufacturing engineer is often not the original test developer

BENEFITS OF SOFTWARE ABSTRACTION

DEVELOPMENT	MAINTENANCE
Decouples hardware and software Disconnects sequence development from code (driver) development Provides common API for instrumentation Optimizes code reuse Reduces developement time Separates roles of architect versus test developer	Mitigates risk of obsolescence or hardware changes ▪ Reduces reliance on specific instruments ▪ Allows hardware changes without modifying test sequence Reduces code complexity for future test support/changes Increases compatibility of code across platforms

It's important to understand the difference between a HAL and MAL. A HAL is a code interface that gives application software the ability to interact with instruments at a general level, rather than a device-specific level. Typically a HAL defines instrument classes, or types and standard parameters and functions that those instruments must conform to. In other words, the HAL provides a generic interface to communicate with instruments from the instrument's point of view. A MAL is a software interface that provides high-level actions that can be performed on a set of abstracted hardware. These actions are a way of exercising multiple instruments to perform a task from the UUT's point of view. Together these make up a hardware abstraction framework.

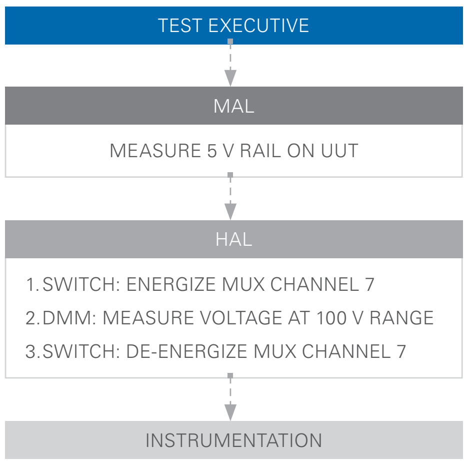

TEST EXECUTIVE
MAL
MEASURE 5 V RAIL ON UUT
HAL
1. SWITCH: ENERGIZE MUX CHANNEL 7
2. DMM: MEASURE VOLTAGE AT 100 V RANGE
3. SWITCH: DE-ENERGIZE MUX CHANNEL 7
INSTRUMENTATION

Figure 1. High-Level Overview of an Abstraction Framework

Printer dialogs are an excellent everyday use of a HAL/MAL. When you print from your computer, you don’t have to open a terminal and send the raw serial, USB, or TCP commands to your printer to initialize, configure, and send the data to print. A hardware driver implements methods to perform configuration and printing. Each printer manufacturer follows certain standards for implementing these methods into their drivers, so that their printers are easy to use. This common interface for executing tasks on a piece of hardware is the HAL. So do you write code to call the abstracted methods of the HAL to configure and print a document? No, when you select print, a print dialog is displayed. This dialog provides a common interface to adjust the configuration parameters, and send the printable data to the device. This is the MAL, as it gives you the ability to exercise all printers intuitively without having to understand the low-level functions of printer devices. Just like with printing documents, an ATE HAL defines a common set of low-level tasks that each instrument type must follow, and the MAL provides a common means of performing high-level actions that exercise the instruments.

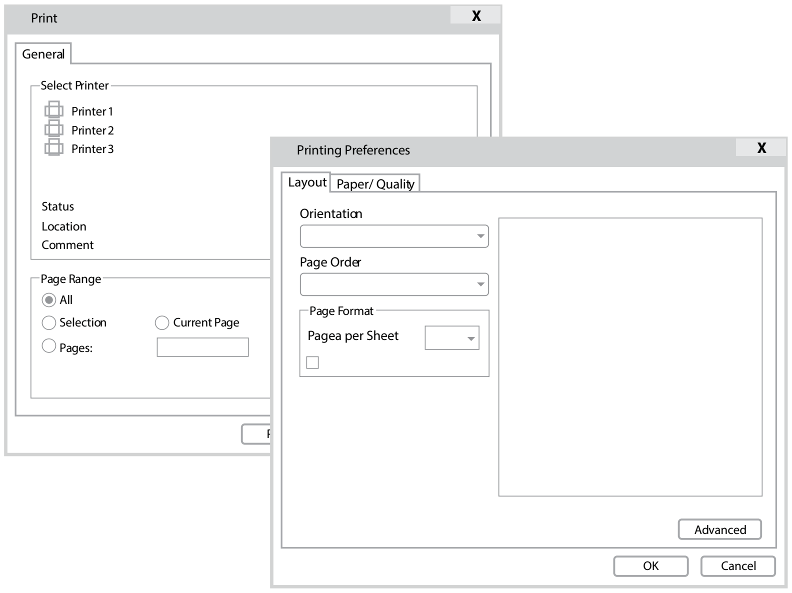


Figure 2. Printer dialogs are an excellent everyday use of a HAL/MAL.

Existing HAL/MAL

The test and measurement world has addressed HALs and MALs in many ways. Much of this can be used right out of the box, or integrated into a larger custom HAL/MAL approach to extend functionality with minimal effort. Here are a few of the most common examples.

ABSTRACTION	DESCRIPTION	TYPE	PROS	CONS
Vendor-Specific Driver Family Drivers (NI-DAQmx, Modular Instruments, Pickering PILPXI)	HAL	Vendor-specific family drivers provide generic interfaces for some groups of a vendor's common instruments. These driver sets can interface with dozens to hundreds of instruments for each particular family. Examples include NI drivers (such as NI-DAQmx, NI-DCPower, NI-DMM, NI-Scope, NI-SWITCH, and NI-FGEN), and Pickering PILPXI.	▪ Common intuitive interface for supported instruments ▪ Well documented and tested ▪ All available functions provided ▪ Low learning curve—the same driver can control all instruments in the family	▪ Valid only for each vendor's specific drivers ▪ Not all instruments support all functions
Industry-Standard Interfaces	HAL	IVI is a standard for instrument driver software that promotes instrument interchangeability and provides flexibility when interfacing with IVI-compliant instruments. The standard defines specifications for 13 instrument classes, which many manufacturers follow, allowing a single driver to control multiple types of instruments. Instrument classes include DMM, oscilloscope, arbitrary waveform/function generator, DC power supply, switch, power meter, spectrum analyzer, RF signal generator, counter, digitizer, downconverter, upconverter, and AC power supply.	▪ Available for a wide variety of instruments from USB to PXI ▪ Compatible with many boxed GPIB, serial, and LXI instruments ▪ Plug and play ▪ Standard programming model for all drivers ▪ High-level instrument API ▪ Allows simulated devices	▪ Only API is specified, not the implementation—Two "interchangeable" implementations may return different results for the same measurement ▪ Cannot be used with noncompliant instruments ▪ May not implement all functions required ▪ May expose functions that are not supported by an instrument
Switch Executive	MAL	Switch Executive is a switch management and routing application that allows compliant switch matrix and multiplexer instruments to be combined into a single virtual switch device. This virtual switch can be intuitively configured and actuated using named signal channels and routes.	▪ Intuitive switch route setup and operation ▪ Define channels and routes based on UUT- or test-centric names ▪ Define no-connect routes for added safety	▪ Requires switches to be NI- or IVI-compliant ▪ Doesn't work with relays controlled with NI-DAQmx

Table 1. Out-of-the-Box Software Abstraction Layers

Out-of-the-box abstractions provide a lot of functionality with minimal customization. However, they don't provide unification. IVI drivers and NI family drivers are great HALs for compliant instruments, but they still require test sequences to be developed from an instrument-centric point of view. Switch Executive does an excellent job of abstracting switch routes to a test-centric point of view, but it can be used for only NI- or IVI-compliant switch connections (no analog or digital I/O, DMM, Scope, power supply, and so on). By using a unified HAL/MAL, you can more effectively develop UUT-centric sequences that can interface with a wide variety of instrumentation and better handle changes to instrument channels and connections.

Although beneficial, HALs and MALs require a lot of foresight that typically comes from past experience. There are many different levels of abstraction to consider. Some are software and time intensive, and others are given out of the box. In general, the more abstracted from specific instrumentation and measurements you get, the more high-level framework planning and software development is required. Architecting a large abstraction framework is time-consuming, and can be risky without proper planning. Improper initial assumptions or implementation can have both positive and negative lasting consequences. It is important to find the right scope of hardware substitution for your particular needs. If you are unsure of how to proceed, start simple, keep it scalable, and use built-in abstraction when possible.

Background

To best understand how a HAL/MAL is implemented, you must understand the anatomy of automated test software. At the highest level, automated test software employs a test executive (or sequencer), such as TestStand. The executive calls a series of test steps, which most often are code modules or functions, developed in languages like G in LabVIEW software, C, .NET, or ActiveX. With a custom instrument-specific approach, these code modules have specific purposes, such as a switched DMM that uses the DMM and switch, or a power supply with ripple measurement that uses both the power supply and the scope. Although this can be beneficial, because it gives each developer the ability to code the specific functions needed, it requires a large amount of cross-functionality and can be difficult to develop, deploy, and manage. Furthermore, it requires every test developer to be well versed in the low-level software (such as LabVIEW).

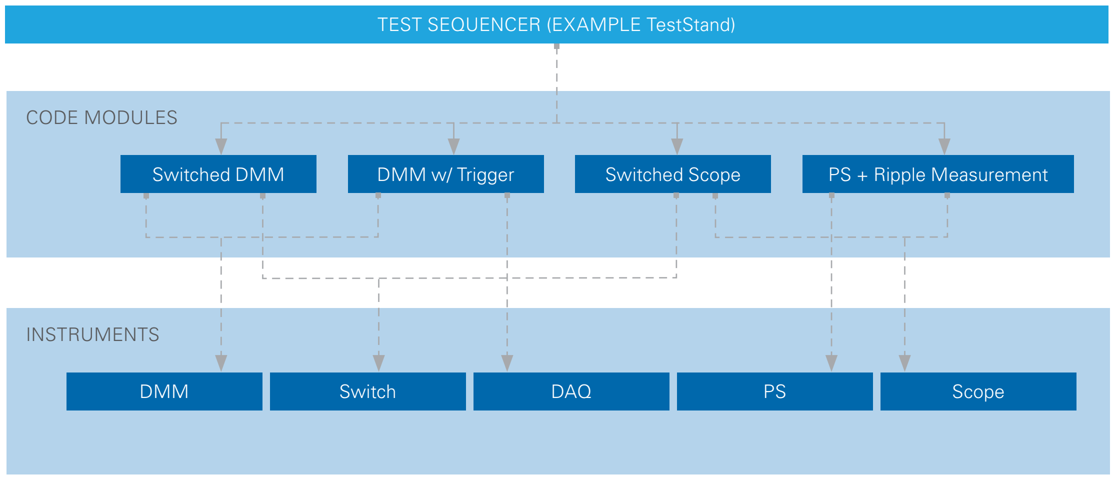


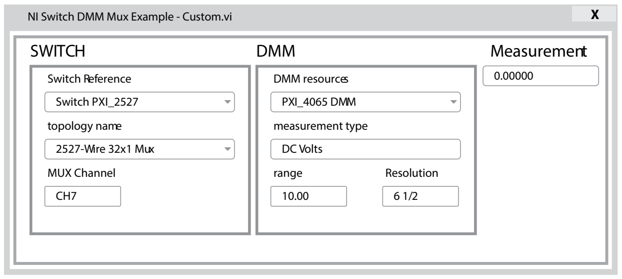
Figure 3. The Anatomy of Nonabstracted Automated Test Software

Without Abstraction

Without hardware or measurement abstraction, you must employ code modules that directly reference drivers to interface with instruments. This results in a test sequence that is closely coupled to specific instruments and specific driver code. Four inevitable problems occur without a HAL/MAL framework:

- Instruments need to change because of obsolescence or requirement changes—Without abstraction, you need to change the driver for each call to that instrument, which could be dozens of steps in a typical test sequence. Each instrument change causes a chain reaction of software changes.
- Driver functionality changes because of new requirements—If a driver needs to be updated, you may need to update every instance of that driver to match the new code, especially if the inputs or outputs change. Furthermore, directly calling driver code modules requires that every test developer understand the inner workings of each driver they use, especially in the case of multifunction action engines. By exposing all of this functionality, test engineers must also be well-versed software engineers.
- Test sequences are developed from the point of view of the instrumentation—By using instrument-specific drivers, all test sequences are developed using instrument-centric channel names (for example, you develop test sequences using instrument-centric names) rather than UUT- or test-centric names (for example, 5V_Rail, LED_Control, VDD). Because you developed test procedures from the UUT's point of view, this makes development and debugging difficult. Furthermore, any test changes require intimate knowledge of the instrumentation, wiring, and interconnects.
- Test sequence development occurs at the same time as hardware development—To achieve tight deadlines, software and hardware development often happen concurrently. Therefore, the instrumentation and channel details are not always known when developing test sequences. Without abstraction, you'll need to leave placeholders for drivers, channel numbers, and connections. Any hardware signals that change require updates to the test sequence.

For example, with the custom approach, a multiplexed DMM measurement code module may look something like the image below, a common switched DMM LabVIEW VI. The code module has a specific set of calls to specific instrument types. In this example, these are the NI Mux and NI DMM. This code module connects a switch based on an input channel and switch topology, measures using the DMM based on some input parameters, and then disconnects the switch. In the test executive, you must know what fields to fill out, and exactly what channels, topologies, and configurations are needed from the instrumentation's point of view. You must also make sure to pass the switch and DMM measurements to the code module appropriately.


NI Switch DMM Mux Example - Custom.vi

SWITCH
Switch Reference: Switch PXI_2527
topology name: 2527-Wire 32x1 Mux
MUX Channel: CH7

DMM
DMM resources: PXI_4065 DMM
measurement type: DC Volts
range: 10.00
Resolution: 6 1/2

Measurement: 0.00000

Figure 4. Front Panel of a Typical Multiplexed DMM Measurement Application in LabVIEW

From the perspective of the test executive, the code module is called to perform a specific function (multiplexed DMM). This function implements specific calls to the instruments for which it was developed. The block diagram below shows the nesting of command calls. In the diagram, the test executive contains a step that calls the code module. The code module employs drivers to talk to specific instruments. Each outer item is dependent on its internal calls.

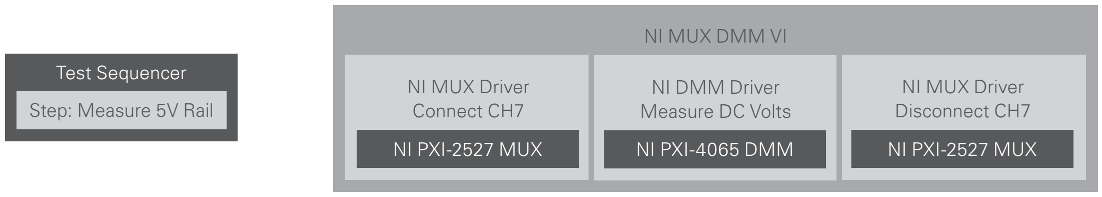


Figure 5. Nested Command Calls to Perform a Multiplexed DMM Measurement

If an instrument must change, every function in the line of dependencies must change. For instance, if the initial multiplexer lacks enough channels, and needs to be switched for a higher channel count matrix, a series of changes must take place because of the chain of dependencies:

1. **Instrument**—PXI-2527 mux is changed to a PXI-2532B matrix
2. **Driver**—NI Mux driver changes to NI Matrix (rows/columns instead of channels)
3. **Code Module**—NI Mux DMM VI must be changed to an NI Matrix DMM VI
4. **Function Call**—The test executive call to the code module must be updated
5. **Sequence**—Test sequence must be updated for every call to that code module

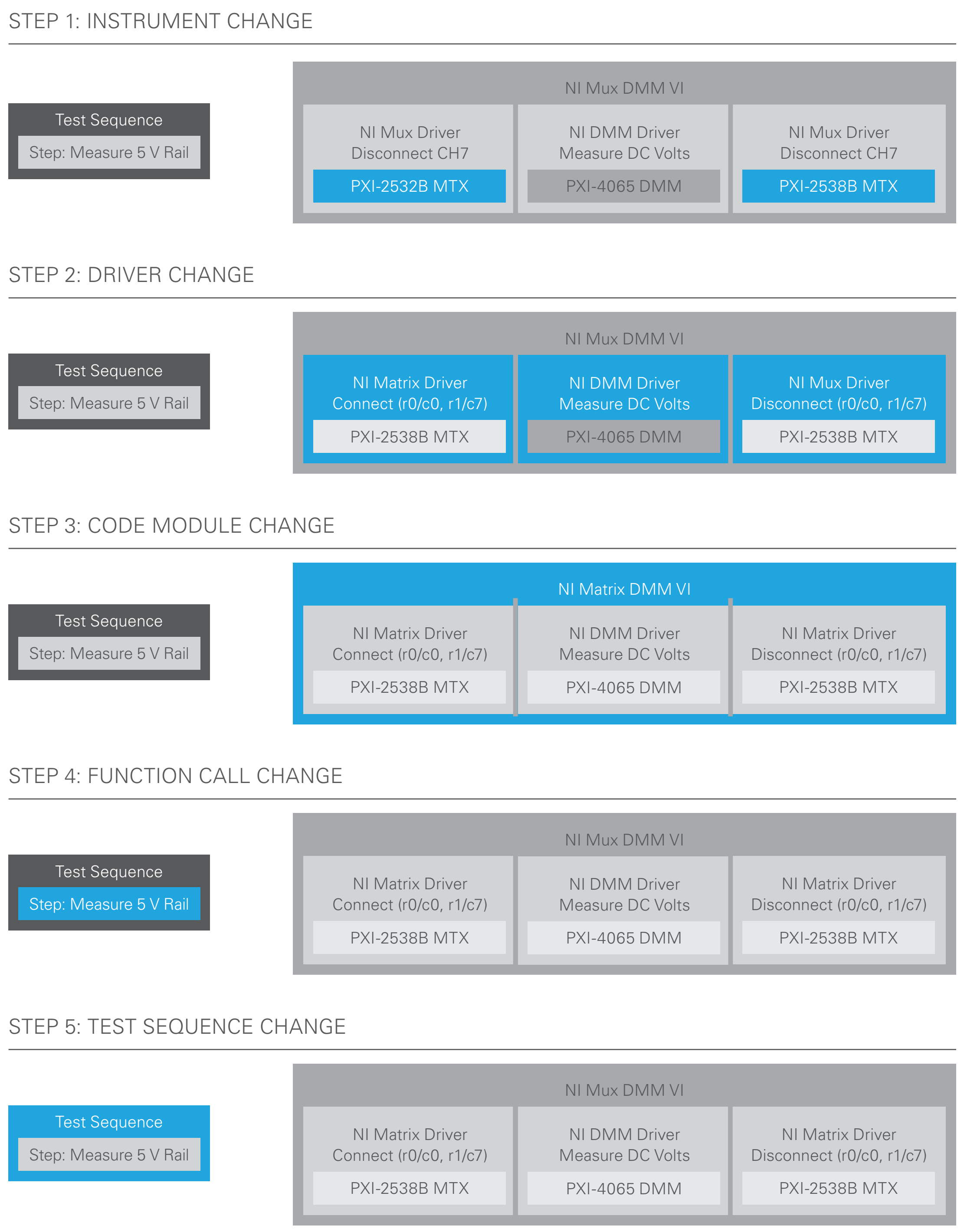


Figure 6. Nonabstracted Changes Required by Chain of Dependencies

With Abstraction

Hardware and measurement abstraction breaks the coupling between the test executive and the code modules that interact with the instruments. Instead of calling code modules that directly interact with specific instruments, the test executive interacts with the MAL. This defines actions or step types that perform common tasks based on generic instrument types. These actions are instrument-generic and typically have high-level names like "Signal Input," "Signal Output," "Connection," "Power," and "Load." They also take in test-specific parameters (rather than instrument-specific parameters) like signal name, connection name, power supply alias, voltage/current, and load method (CV, CC, CP). A mapping framework uses a configuration file to translate test-specific parameters of the generic actions into instrument-specific parameters like instrument references, channel numbers, matrix rows and columns, GPIB addresses, and instrument configuration constraints. The framework interfaces with the HAL to communicate with the specific instruments that the configuration file defines. It calls the appropriate methods of each specific instrument based off of the MAL action type with instrument-specific parameters pulled from the configuration file.

If you think of a single step as a cooking recipe (pancakes), the details in the configuration file would be the ingredients (eggs, milk, butter, flour), the actions would be the cooking functions (combine, mix, beat), the drivers would be the kitchen tools (bowl, mixer, griddle), and the framework would be the instructions that put it all together.

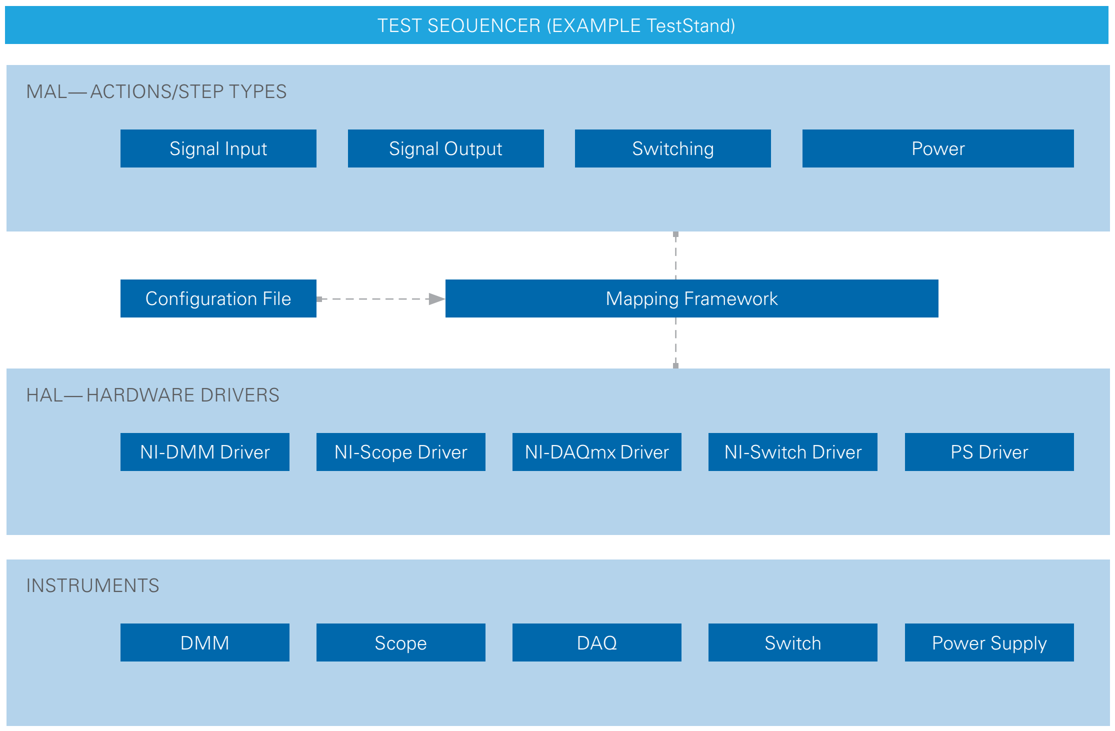


Figure 7. Anatomy of Abstracted Automated Test Software

This section continues to the multiplexed DMM example using abstraction. In this example, the test executive calls a generic step type, Signal Input, using a step-specific input parameter 5 V Rail. In this particular framework, Signal Input is defined as three device actions: connect signal route, read measurement device, disconnect signal route. This is passed to the mapping framework using the 5 V Rail parameter. The mapping framework reads the configuration file to find the instrument and channel details of 5V Rail. These correspond to a connection of the PXI-2527 mux channel 7, and a measurement of the PXI-4065 DMM in DC volts mode. The framework then calls the appropriate abstracted drivers, NI-Switch and NI-DMM, to communicate with the specific instruments that the configuration file defines.

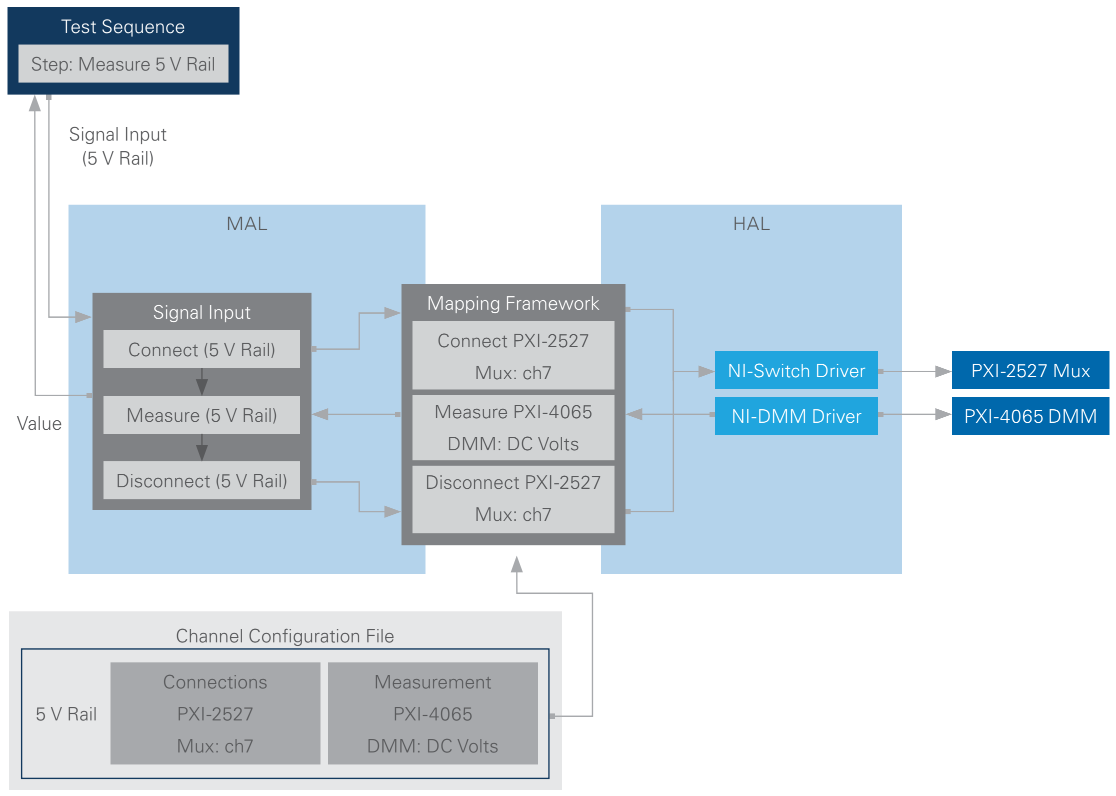


Figure 8. Function Calls for a DMM Measurement With an Abstraction Framework

Executing the same change as discussed in the nonabstracted example, where the PXI-2527 mux is replaced with a PXI-2532B matrix proves to be much easier when using a HAL/MAL framework. Because all of the instrument-specific details are stored in the configuration file and the HAL provides a common interface for interacting with similar instruments, only the configuration file needs to change. By replacing PXI-2527 Mux: Ch7 with PXI-2532B Mtx: r0/c0, r1,c7, the mapping framework automatically pulls the updated details and calls the new matrix with the new parameters. No test sequence or code module changes are required.

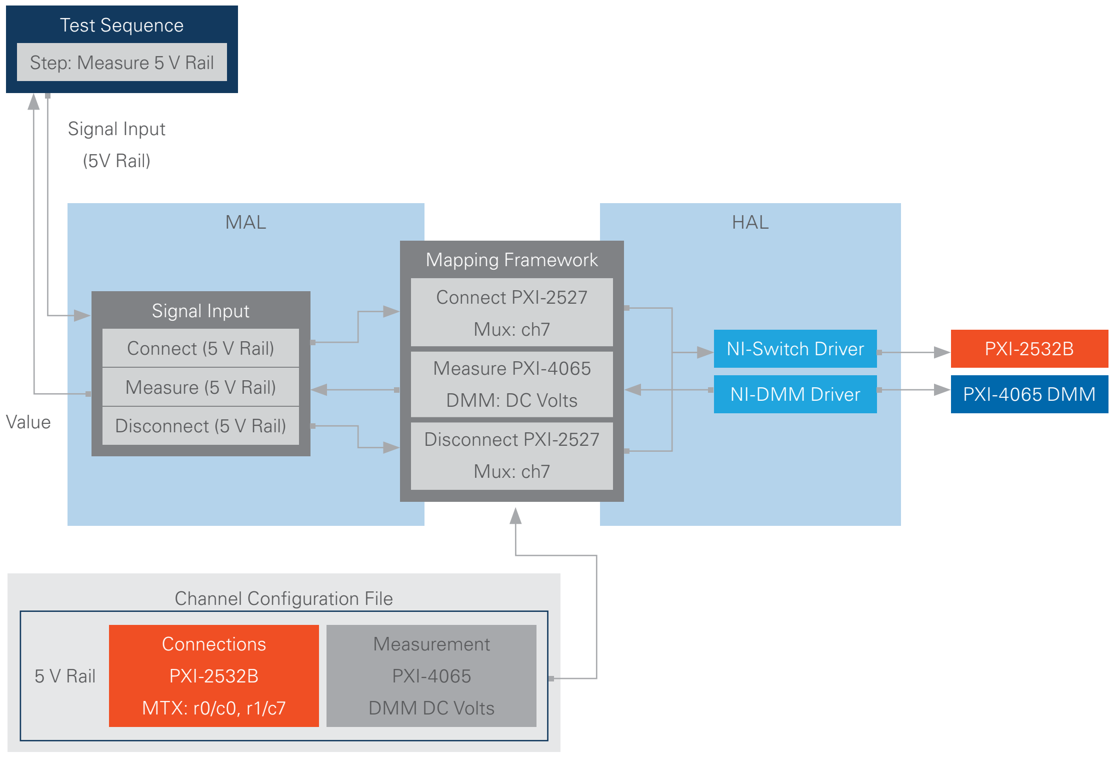


Figure 9. Abstraction makes it easy to update hardware with minimal software updates—just updates to the configuration file.

Approaches

The most important topic to consider when deciding on an abstraction framework is the scope of abstraction on which all other decisions are based. On one extreme, there is the case for no abstraction, where each hardware interface is a direct call to an instrument-specific driver. On the other extreme, you have complete abstraction, where every possible interface between components, communications protocols, measurements, and configuration formats has an abstract definition. This section explores some of the options that cover the range of possibilities.

Option 1: Instrument-Specific Driver

The instrument-specific driver approach is probably the most commonly implemented in automated test, mainly because it requires the least amount of coding, foresight, and planning. With this approach, low-level code modules are developed to interface with specific instruments. These are typically referred to as low-level drivers, or instrument drivers, which are then called by higher level code modules or directly by the test executive. The block diagram below shows each of the instrument drivers developed for a specific instrument. In this scenario, if the instrument changes, the driver and higher level calls must also change.

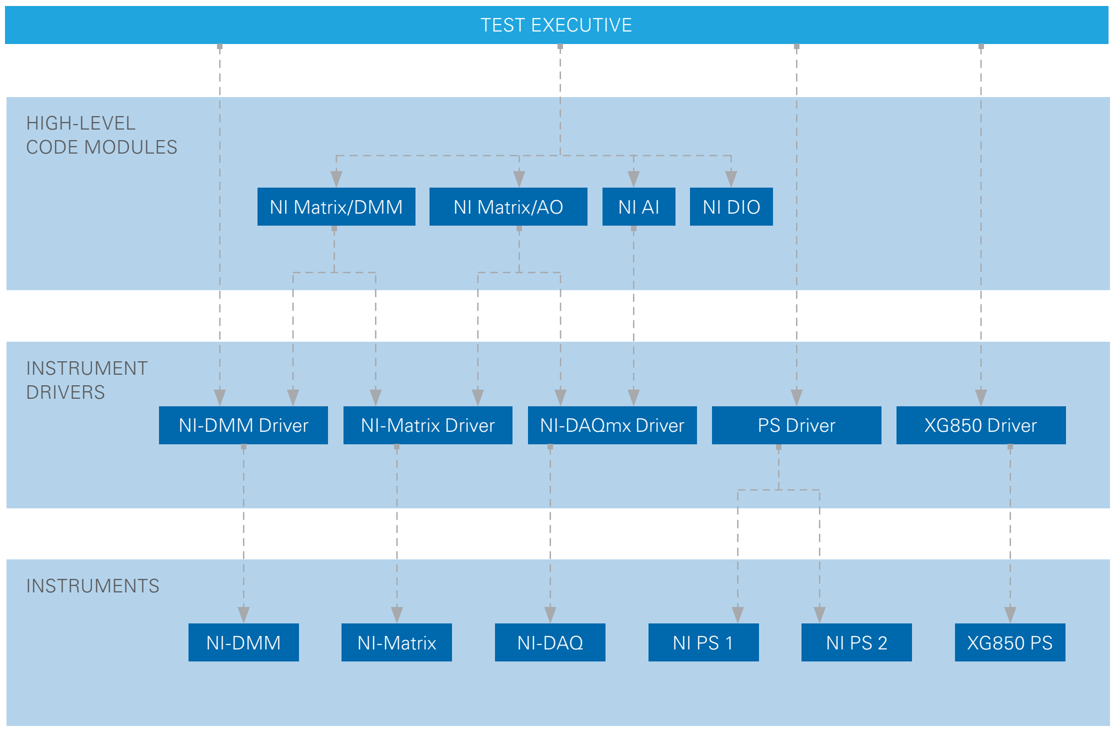


Figure 10. Overview of an Instrument-Specific Driver Method for Automated Test Software Without Abstraction

Although this method does not include any abstraction, there are still best practices you should follow to promote robust driver development and interactions:

- Develop or use instrument driver packages for interfacing with each instrument.
  - A low-level driver package implements all of the functions for initializing, interacting with, and closing a connection to an instrument.
  - Functions should be simple and single-purposed.
  - Drivers should be able to handle multiple instances of the same instrument type (such as two identical power supplies in the same system).
- Develop wrapper instrument drivers to simplify the instrument interface.
  - Pre-existing drivers contain dozens of functions that may be difficult to understand. You can wrap pre-existing full-featured instrument drivers into simpler wrapper instrument drivers to promote easy usability.
- Ensure all instrument interfacing goes through instrument drivers.
  - This provides a single point of entry for all instrument communications, which eases debugging, reduces race conditions, and allows the instrument state to be managed in a single location.
  - A wrapper instrument driver, if developed, should be the single entry point.
  - Drivers may be called directly by the test executive, or by higher level code modules.
- Do not implement test-specific functionality at the driver level.
  - Test-specific algorithms should be implemented by higher level code modules or in the test executive.

- Ensure instrument drivers are unaware of one another.
    - High-level code modules or the test executive calling individual instrument drivers should perform multi-instrument interactions.

Option 2: Out-of-the-Box HAL/MAL

The fastest way to incorporate abstraction into the instrumentation driver architecture is to use pre-existing HALs and MALs. Although the options for purchasing a fully integrated HAL/MAL abstraction framework are limited, many hardware vendors have already implemented some level of hardware abstraction into their instruments; Switch Executive is a MAL geared specifically toward switch connections and routing. By architecting your code modules around these pre-existing abstractions, you can increase ATE software adaptability and abstraction with minimal development effort.

Out-of-the-Box Hardware Abstraction

Pre-existing hardware abstraction uses common low-level interfaces that work with a variety of instruments. This reduces the number of required instrument-specific drivers and reduces the impact of instrument changes in a system. The test executive and higher level code modules can reference general drivers, which reduces development effort and the impact of instrument changes. When one of the abstraction types, defined below, is implemented, the I/O for a particular interface is fixed. Therefore, instrument changes do not typically cause code module changes.

You can use pre-existing hardware abstraction in two ways: instrument family drivers and communications standards. Instrument family drivers tend to be vendor-specific drivers that can control many variations of a particular instrument type within that vendor's catalog. Communications standards provide an industry agreed-on method for interfacing with certain types of instruments across multiple vendors. You may use these standards to develop instrument drivers that can control a variety of similar instruments.

Hardware Abstraction Through Instrument Family Drivers

Instrument family drivers are vendor-specific drivers that communicate with a common product line of instruments. Similar to IVI drivers, instrument family drivers provide communications to multiple different instruments using a common driver. Common examples include NI modular instruments (NI-DMM, NI-Switch, NI-DCPower, and NI-Scope) and Pickering PILPXI. Instrument family drivers promote interchangeability within the family for which they are developed. Although they do not support cross-vendor or cross-family reuse, these drivers are typically intuitive, easy to implement, and contain most, if not all, of the functions for each instrument.

Hardware Abstraction Through Communications Standards

Many instrument manufacturers follow industry standards for device communications. By following industry standards, manufacturers can make their instrumentation interoperable with other similar instruments. Two of the most common standards are the Standard Commands for Programmable Instruments (SCPI, often pronounced "skippy") and Interchangeable Virtual Instruments (IVI).

SCPI

SCPI defines a standard for syntax and commands to use in controlling programmable instruments in the test and measurement industry. With these commands, users can set and query common parameters of instruments. SCPI commands can be implemented over a variety of communications protocols, including GPIB, LAN, and serial. By developing a single SCPI-compliant driver, you can communicate with multiple instruments of the same type (DC power supply, electronic load, and so on) without having to develop instrument-specific drivers. When developing a SCPI driver, note that, although SCPI defines a common command and syntax standard, different vendors sometimes implement the standard with minor differences, making a 100 percent standard driver somewhat difficult. When selecting SCPI-compliant instruments and developing drivers, it is important to pay close attention to the command specifics of each instrument.

IVI

IVI is a standard for instrument driver software that promotes instrument interchangeability and provides flexibility when interfacing with IVI-compliant instruments. The standard defines an I/O abstraction layer using VISA. Because of the incorporation of SCPI into IVI, many instruments that are SCPI-compliant are by definition IVI-compliant. The IVI standard defines specifications for 13 instrument classes that many manufacturers follow, which gives a single driver of each type the ability to control multiple unique instruments from different vendors. Instrument classes include DMM, oscilloscope, arbitrary waveform/function generator, DC power supply, switch, power meter, spectrum analyzer, RF signal generator, counter, digitizer, downconverter, upconverter, and AC power supply. Many PXI and boxed instruments follow the IVI standard, and pre-existing drivers are available in many programming languages and test executives.

By developing test sequences and code modules using IVI drivers for IVI-compliant instruments, one vendor's instrument looks the same as another's. You may use a single driver set for each type to interface with many interchangeable instruments. If an IVI-compliant instrument is replaced with one of similar functionality, code and sequence updates are reduced as compared to using instrument-specific drivers. However, although IVI drivers can implement most functions of compliant instruments, some instruments may still require specific code for executing custom functions. Conversely, some instruments may not be capable of handling all IVI-compliant functions. Finally, although two instruments may execute identical IVI functions, they may not always achieve identical results. Always verify and test the functionality of instrumentation whenever changes are made.

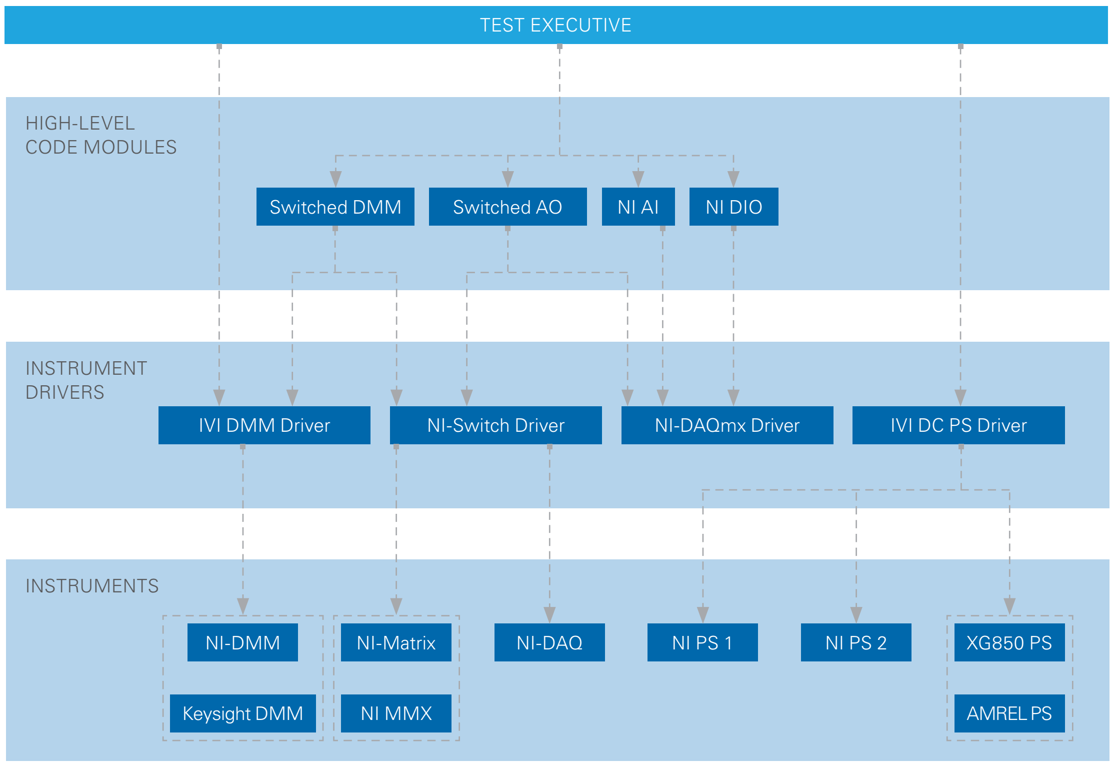
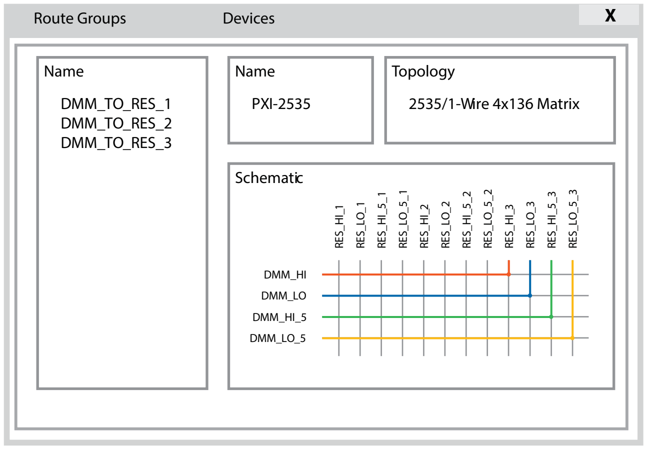


Figure 11. Overview of Automated Test Software With Out-of-the-Box Abstraction

Out-of-the-Box Measurement Abstraction

Although pre-existing hardware abstraction is relatively common, it allows abstraction from only an instrument point of view. Conversely, measurement abstraction is very limited. Because of the high level of customization across test systems, it is difficult to define a standard for measurement actions. The most well-known out-of-the-box measurement abstraction layer is Switch Executive, a switch management and routing application that allows compliant switch matrix and multiplexer instruments to be combined into a single virtual switch device. This virtual switch can be intuitively configured and actuated with user-named channels and routes. Although valid for only devices compliant with NI-Switch and IVI switch, Switch Executive provides an excellent method of defining switch routes from the point of view of the UUT or test.

First, Switch Executive provides a Graphical Configuration Utility for setting up switch channel names and routes within single instruments and across multiple instruments. Rows, columns, channels, and route groups can all be configured and named to intuitively set up a switching scheme.


Route Groups | Devices | X

Name
DMM_TO_RES_1
DMM_TO_RES_2
DMM_TO_RES_3

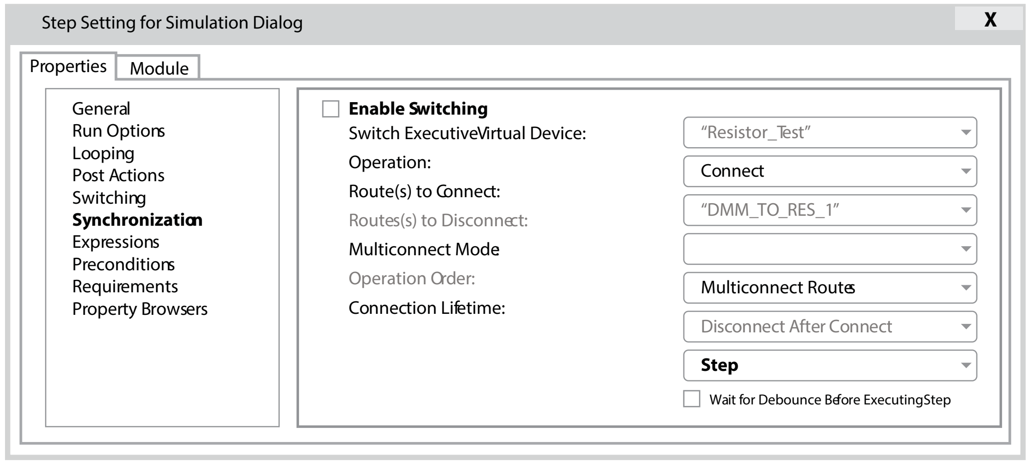
Name
PXI-2535

Topology
2535/1-Wire 4x136 Matrix

Schematic

Figure 12. Switch Executive MAL Configuration Interface

Next, Switch Executive integrates into LabVIEW and TestStand to provide powerful interfaces for setting and querying the preconfigured routes by name. When used with the TestStand test executive, Switch Executive can be used on a step-by-step basis to provide a named interface to the switch instruments before executing the step's code module.


Step Setting for Simulation Dialog | X

Properties | Module

General
Run Options
Looping
Post Actions
Switching
Synchronization
Expressions
Preconditions
Requirements
Property Browsers

☐ **Enable Switching**

Switch ExecutiveVirtual Device:	"Resistor_Test"
Operation:	Connect
Route(s) to Connect:	"DMM_TO_RES_1"
Routes(s) to Disconnect:	
Multiconnect Mode	Multiconnect Routes
Operation Order:	Disconnect After Connect
Connection Lifetime:	**Step**

☐ Wait for Debounce Before ExecutingStep

Figure 13. Switch Executive MAL Test Setup

Switch Executive is a useful MAL that abstracts switch connections to test-specific names rather than instrument-specific names. When used in conjunction with IVI-switch hardware abstraction, it proves to be an excellent example of an integrated HAL/MAL framework. However, it falls short when non-IVI switches or external digital-output-controlled relays are used. Furthermore, Switch Executive pertains only to switch routing, and does not extend to other measurement types. To achieve an integrated HAL/MAL framework beyond switching, custom code development is required.

Option 3: Integrated HAL/MAL Framework

An integrated HAL/MAL framework provides a structure for implementing high-level actions called by the test executive (MAL), interfacing with low-level drivers to communicate with instruments (HAL), and mapping the details between the two. This framework is implemented by three major types of code modules: actions, mapping framework, and hardware drivers. Each of these code module types are defined by a set of APIs. An API is a set of tools (functions, protocols, parameters, syntax) for software applications, which define how a code module should function and interact with the software around it. In a basic HAL/MAL framework there are four common APIs: Measurement API, Configuration API, Hardware Driver API, and Instrument API. The code modules, APIs, and their interactions are shown and described below.

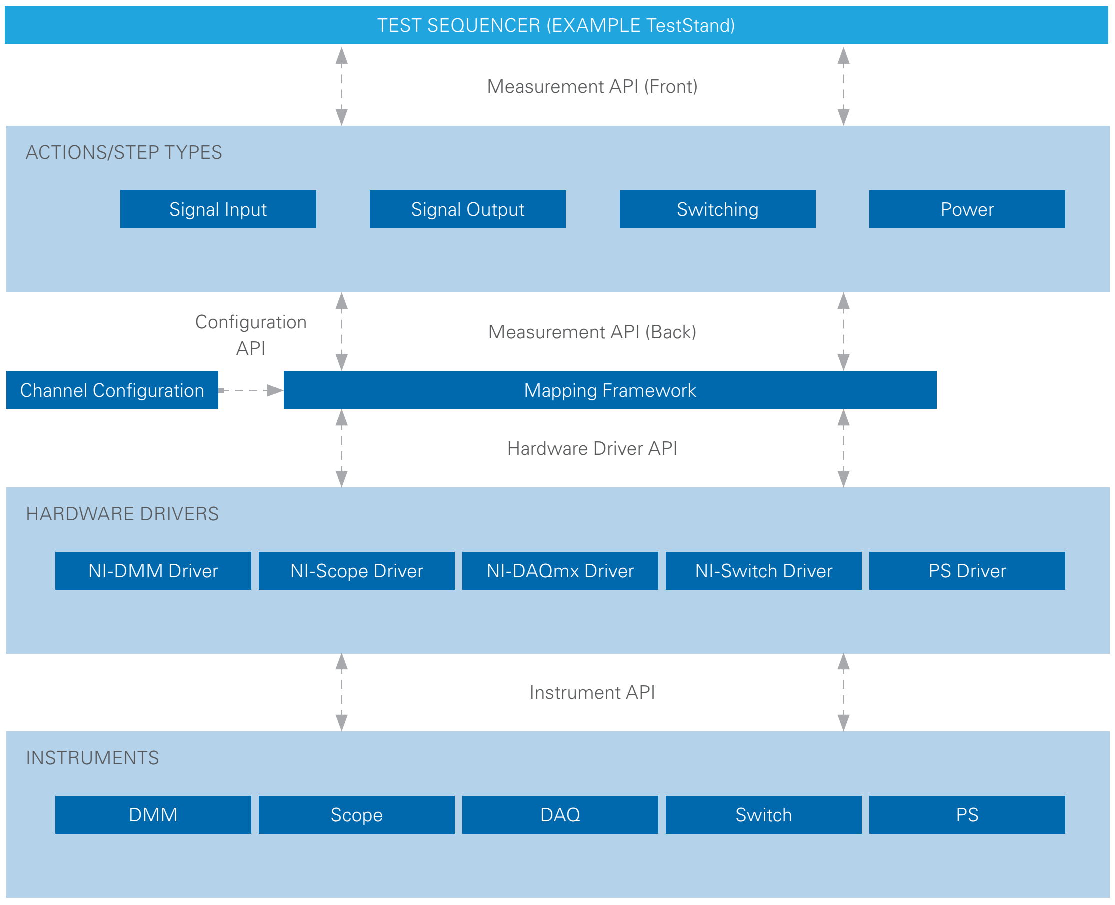


Figure 14. Overview of Automated Test Software With an Integrated MAL and HAL

The three types of code modules are:

- Actions/Step Types—The actions define the capabilities of the MAL. A specific action defines each measurement type (input or output). Actions can be as simple as a single function call to a single instrument type, such as making a switch connection. They can also be as complex as multiple function calls to multiple instruments, such as combining a switch connection with setting a power supply voltage, current, and enabled state. These code modules implement the Measurement API for defining their methods and parameters.
- Mapping Framework—The mapping framework is the internal code that links the high-level actions to the low-level instrument devices using defaults from the configuration file. The mapping framework code module interacts with the hardware drivers through the Hardware Driver API, and with the actions through the Measurement API.
- Hardware Drivers—The hardware driver code modules translate the generic device type function calls (DMM, power supply, switch, and so on) to instrument-specific communications (SCPI, IVI, NI-DCPower, and proprietary communications). Therefore the hardware drivers implement the Hardware Driver API on one end, and instrument-specific API on the other.

A HAL/MAL abstraction framework contains a minimum of the following four APIs:

- Measurement API—The Measurement API defines the high-level actions and their specific parameters. This is the MAL definition. The Measurement API defines a common framework that all actions must follow, and then allows each action to define its own API (parameters and methods) required to carry out its particular function. Each action must at a minimum implement the back-end Measurement API, which the mapping framework uses to link the human readable alias to specific switching and measurement instruments and the appropriate channels. Optionally, a front end to the API may be developed that provides a more intuitive interface to each action. This front end is typically a configuration dialog/wizard. An example Measurement API for a signal input would define a signal input alias and an output of the return value. The API would also define that, for the alias, a connection, measurement, and then disconnection is made.
- Configuration API—The mapping framework uses the Configuration API to fill in the details on how to translate from the Measurement API to the Hardware API. The Configuration API defines the parameters, syntax, and content of the configuration file or database. Only the mapping framework uses this API. For example, the Configuration API may dictate that the configuration file is a Microsoft Excel file and that each signal alias should have the following properties: name, type, connection details, instrument, instrument configuration, and scaling.
- Hardware API—The Hardware API is the abstracted API that defines what common parameters and methods a particular type of instrument must implement. This API defines the HAL. For example, the DMM Hardware API might dictate that all DMMs must be able to initialize, configure (voltage; current; resistance, range, resolution), measure (return value), and close.
- Instrument API—The Instrument API is defined by each individual instrument, and is therefore not an abstracted layer. Each instrument-specific hardware driver implements the necessary functions and commands for controlling its particular instrument. This is the same API that would be used in an instrument-specific code interface, and would implement the specific communications protocols and commands for that particular instrument.

To better understand the interactions between the code modules and APIs, revisit the multiplexed DMM example with a detailed explanation of the inputs and output of each code module.

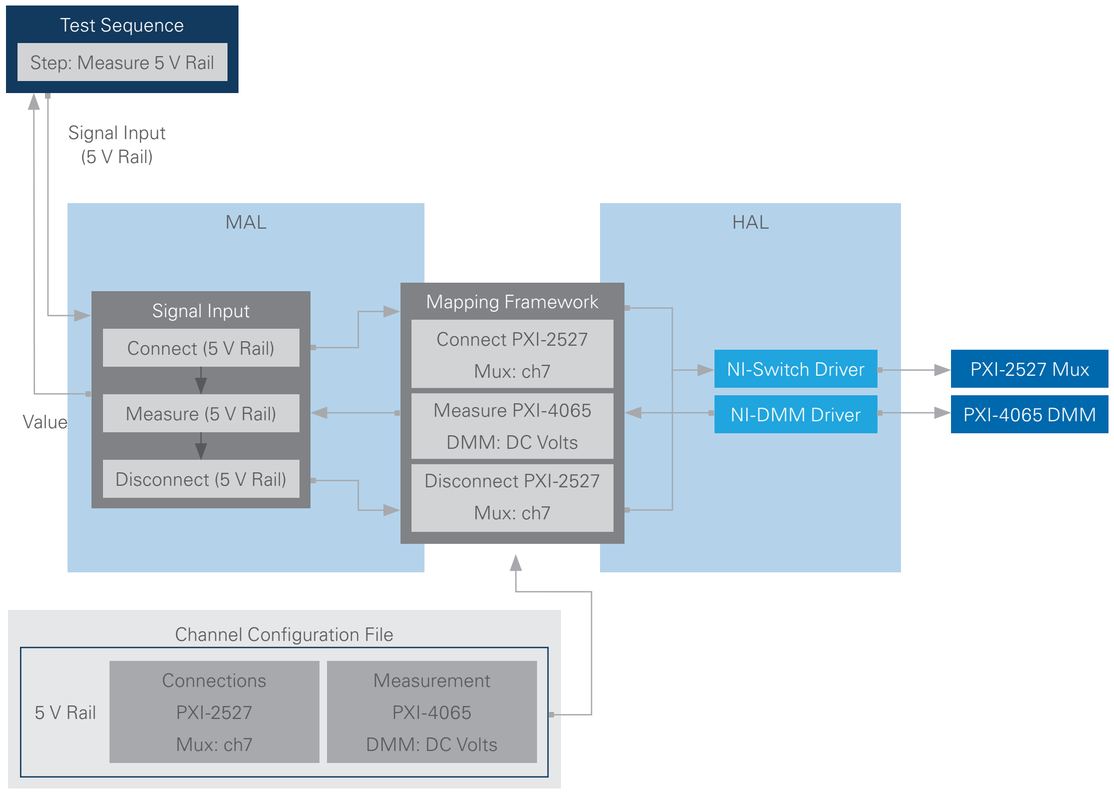


Figure 15. Multiplexed DMM Measurement With an Abstraction Framework

In the example, the signal input block is the action code module, which defines that a signal input should execute a Switch Device Connect function, a Measurement Device Measure function, and then a Switch Device Disconnect function. The Measurement API for this function defines that the code module requires an alias that it receives from the test executive, then passes to the mapping framework, and then gets a return value from the mapping framework to pass back to the test executive.

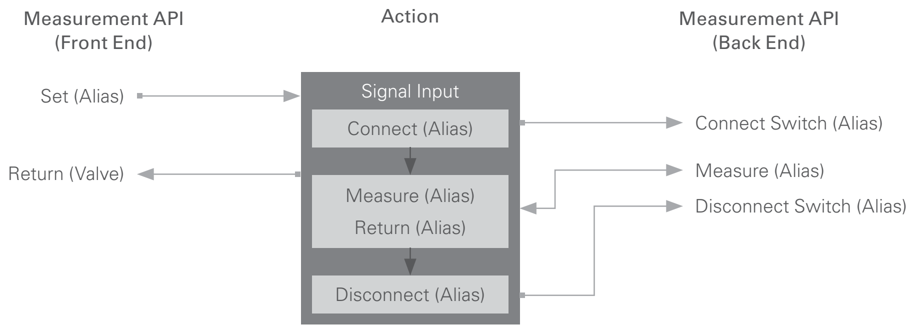


Figure 16. Example of MAL Action APIs for a Signal Input

The mapping framework receives the commands from the action through the Measurement API. It then parses the alias data from the configuration file through the Configuration API to obtain the correct instrument IDs and parameters. The Configuration API defines the file format, syntax, and fields for the system configuration. The mapping framework then passes the instrument-specific information to the appropriate drivers through the Hardware Driver API.

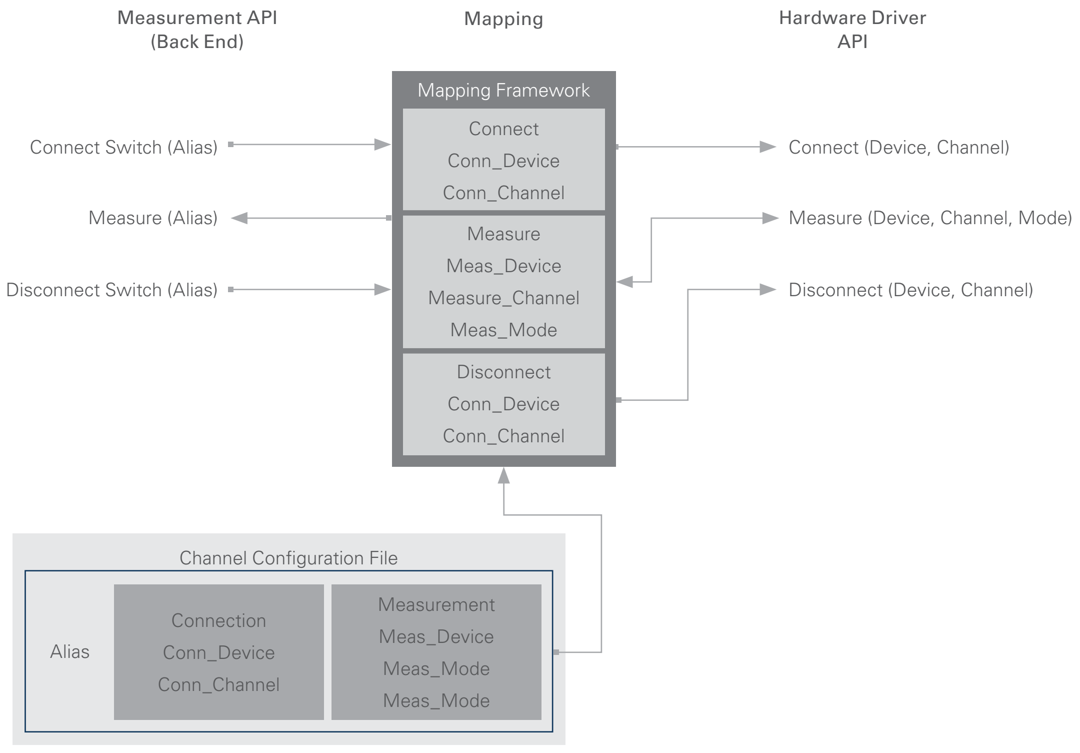


Figure 17. Example of Mapping Framework APIs for a Signal Input

The mapping framework calls the individual hardware drivers using the generic Hardware Driver API. Each driver then interprets the details of the generic setup and communicates with the specific instruments using their own out-of-the-box methods and parameters.

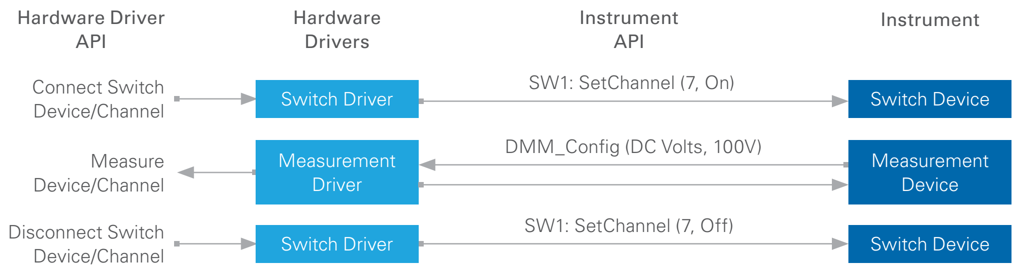


Figure 18. Example of Hardware Driver APIs for a Signal Input

Option 4: HAL/MAL Plugin Architecture

Plugins are potentially valuable additions to an integrated framework. A true plugin is simply a software component that can be modified after deployment without redeploying an entire application. Plugins are stored on disk separately from the main application and/or framework and are loaded dynamically at run time.

Although developing a plugin architecture introduces several challenges, it also simplifies software regression testing by clearly limiting the scope and risk of added or modified functionality. A framework developed without plugins must be rebuilt each time a new measurement type, instrument driver, or configuration format is needed. Because, without plugins, the entire source is built into a single EXE, there can be no guarantee that a seemingly trivial change to one instrument library did not inadvertently affect other application features. Testing must be thorough because it is difficult to know all possible effects of source modifications.

A plugin architecture provides the highest level of software modularity by giving a developer the ability to add or fix plugin code without modifying, or redeploying the underlying framework. This is achieved by writing a framework that depends only on abstract classes or modules and that loads the required concrete plugins dynamically, usually only as needed. Successful plugin architectures depend on thoughtful interface design. In other words, to make use of plugins in a test framework, the framework must know how to call any possible component that plugs in. If all plugins implement a consistent software interface, loading them at run time requires only that the framework or test application knows where to find them.

Although these are some of the more common processes, APIs, and code modules of an abstraction framework, they are certainly not the only ones. Each framework is unique, and has its own requirements, processes, and implementations. For some teams, this level of abstraction may be more than is required. However, in other cases, the system architect may need to inject additional layers of abstraction. The actual implementations of these APIs are also open to interpretation, based on the needs and abilities of the framework architect and users. Some engineers implement all abstractions with simple action engines, some use more advanced object-oriented programming, some use plugins, and others prefer a single code base. The key is to find the right extent of abstraction and implementation to fit your particular needs and abilities. It is also important to understand that not everything can be solved by abstraction, and sometimes instrument-specific code may still be required. Therefore, when developing an abstraction layer, make sure not to prevent custom code from being developed for advanced functions. You can do this by allowing instrument references to be obtained by higher level code modules or by the test executive. Advanced developers should never be hindered by a framework.

None ○ Some ◒ Full ●

ABSTRACTION OPTION	OPTION 1 NONE	OPTION 2 OUT OF BOX	OPTION 3 BASIC CUSTOM	OPTION 4 WITH PLUGINS
Allows individual instruments to be replaced with:				
Instrument with same communications protocol	●	●	●	●
IVI- or family-compliant instrument	○	●	●	●
Instrument with different communications protocol	○	◒	●	●
Change instrument channels/ wiring without modifying test sequence (modify config file)	○	◒	●	●
Measurements/tasks from point of view of test/UUT	○	◒	●	●
Add new instruments or measurements without modifying framework	○	○	○	●

Table 2. Feature Comparison for Abstraction Layer Options

Practical Scenario 1

You, a test engineer from a commercial product company, have been tasked with developing functional tests for the electronic subassemblies of a new product. There are three PCBAs and a final assembly that your need to test. An existing general-purpose ATE instrumentation platform exists but it is outdated, and previous test programs have been recently plagued by equipment failures and obsolescence. Fortunately, a new ATE platform has been designed as part of this program, and it allows interchangeable test heads to adapt the instrumentation to different assemblies.

Your task is to develop the test sequence and code module software to interact with the instrumentation and fixtures that hardware engineers are developing. You have some experience with a test executive (the same one used by the previous platform), and have been developing software applications for a few years. As part of the effort, there have been talks about using abstraction to help mitigate the obsolescence issues of the previous system. You must decide if this is the right way to go and how far to take it.

To Abstract or Not to Abstract…

The first decision you must make is whether to develop an abstraction framework, regardless of the level of abstraction. Given the out-of-the-box options, like IVI, the answer to this decision is almost always yes. The only time that abstraction is not worth the effort is if the project lifespan is 100 percent known, and changes will never be required, which is almost never.

Will You Need a HAL?

The next decision to be made is what level of hardware abstraction to use. This is where the decision gets more complicated, as many factors are at stake. Hardware abstraction is typically easier to understand, and therefore less costly to implement than a MAL. This is especially true if you can reasonably commit to using pre-abstracted drivers, such as IVI and product family drivers. However, as soon as you must use instruments that don't fall into a single driver, you may need to develop a generic interface for each instrument type. For instance, if your system has some IVI-compliant power supplies, as well as a noncompliant supply, you may want to develop an abstracted power supply definition that works with either type. Defining an abstract hardware definition typically requires past knowledge of how most instruments of that particular type work. You can then use that information to define the common methods and parameters for each instrument type within your system.

Aim for covering about 80 percent of the functions that you reasonably expect each device to use. Talk with your team to determine the core functions and parameters of each instrument type that have to be implemented by each abstracted instrument driver. For example, the team may determine that the core functions of all power supplies should be initialize, set voltage/current/enabled state, readback voltage/current/enabled state, and close. Although there may be other functions that one power supply could potentially use in the future, it may not always be worth it to include as part of your system's standard. If you don't know enough about a particular instrument type, or are unsure of what functions to require, start small. You can always add to the standard in the future, but it is difficult to change the parameters or details of a function after it is in use by multiple drivers.

The flowchart below can help you decide what level of hardware abstraction is right for you. If you are unsure of an answer, you can either assume toward more of an abstracted solution or toward the less abstracted solution. A more abstracted solution requires more upfront design, but may save time in the long run, while the less abstracted solution gets you up and running faster, but may be problematic in the future. One item to note is that the first question is if you require a MAL. This is because a MAL cannot be effectively implemented without a well-designed HAL.

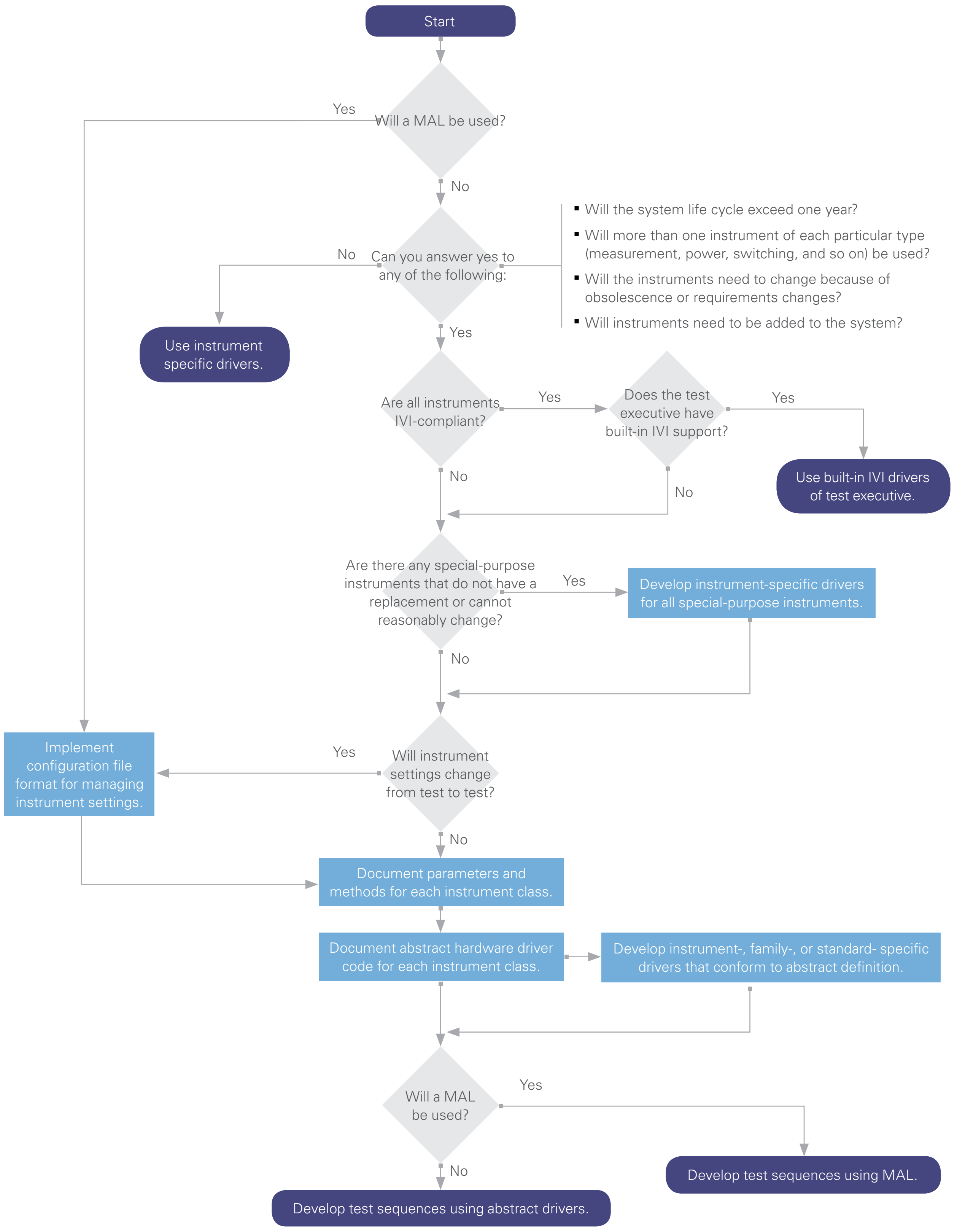

Start
Will a MAL be used?
Yes
No
Can you answer yes to any of the following:
No
Yes
Will the system life cycle exceed one year?
Will more than one instrument of each particular type (measurement, power, switching, and so on) be used?
Will the instruments need to change because of obsolescence or requirements changes?
Will instruments need to be added to the system?
Use instrument specific drivers.
Are all instruments IVI-compliant?
Yes
Does the test executive have built-in IVI support?
Yes
Use built-in IVI drivers of test executive.
No
No
Are there any special-purpose instruments that do not have a replacement or cannot reasonably change?
Yes
Develop instrument-specific drivers for all special-purpose instruments.
No
Will instrument settings change from test to test?
Yes
Implement configuration file format for managing instrument settings.
No
Document parameters and methods for each instrument class.
Document abstract hardware driver code for each instrument class.
Develop instrument-, family-, or standard- specific drivers that conform to abstract definition.
Will a MAL be used?
Yes
Develop test sequences using MAL.
No
Develop test sequences using abstract drivers.

Figure 19. Decision Flowchart to Determine What Level of Abstraction to Implement

Will You Need a MAL?

The first decision of a HAL is if a MAL will be required. This is because a MAL is nearly impossible without relying on hardware abstraction. Therefore, this question is really asking if you need an integrated abstraction framework. A HAL/MAL is ideal when there are multiple test developers who may not have low-level software experience. A few major questions can help guide the decision to develop a MAL:

- Will there be a software architect who can plan and support the framework? A HAL/MAL is difficult to support organically without an architect/owner.
- Will there be multiple test developers with minimal software experience? A big benefit of an abstraction framework is that it lowers the learning curve for test development.
- Will the system have a long life cycle that supports many products? This can be a big upfront investment, but the payoff is greater the more it is used.
- Do you feel comfortable developing and supporting a MAL? No abstraction is better than poorly defined and poorly implemented abstraction. When simple and elegant, a HAL/MAL can save a lot of time in the long run; but, when overly complex or poorly designed, it can be cumbersome and actually add development and debug time.

If you answer yes to most of these questions, then developing an integrated abstraction framework will probably pay off in the long run.

Practical Scenario 2

Even if all of the benefits of abstraction are known, there is still the major hurdle of cost versus payoff (where units are typically time). Although the first part of the abstraction decision is typically from a technical perspective, the cost/benefit decision has to be made at a higher business level.

How Much Will It Cost?

This is a difficult question to answer as much of it depends on past experience, coding abilities, and the level of abstraction required. However, you can estimate a rough order of magnitude for various components, as the table below shows.

CATEGORY	TASK	DESCRIPTION	HOUR ESTIMATE (LOW)	HOUR ESTIMATE (HIGH)
Planning	Architecture definition	Documentation of the types of actions, devices, and the general interfaces between them	24	48
	HAL definition per device type	Documentation of the inputs and outputs and methods of each type of device	8 (per device)	16 (per device)
	MAL definition per action	Documentation of the inputs and outputs and methods of each type of measurement/action	8 (per action)	16 (per action)
	Configuration definition	Definition of the format, syntax, and content of the configuration file or database	24	48
Implementation	Mapping framework development	Implementation of all of the software to map the configuration file to actions and abstract drivers—the majority of the underlying framework is developed here	60	120
	Abstract device driver development	Software development of the abstract device interface code, per device type—essentially building the instrument	4 (per device)	24 (per device)
	Instrument driver development	Software development of each instrument-specific driver that uses the HAL—fills in the template for each specific driver	4 (per instrument)	24 (per instrument)
	Action development	Software development for each action defined by the MAL—implements the front-and back-end APIs for interfacing with the test executive and the mapping framework	4 (per action)	24 (per action)
Total		Total time to develop framework (not including individual instrument drivers)—assumes five device types with one instrument-specific driver per device, and five actions	248	776

Table 3. Abstraction Framework Tasks and Costs

This shows that development time for a fully integrated HAL/MAL abstraction layer could be as low as 250 hours, and could exceed 750 hours. Depending on the level of abstraction, this could even exceed 1,000 hours.

What Can You Do to Reduce Cost?

When it comes to software development, cost is closely related to complexity. Complexity can be both good and bad, depending on its nature. The goal is to increase good complexity while avoiding bad complexity. Complexity can be good when it increases functionality. Each feature typically increases functionality. Code that is scalable, flexible, and modular tends to be more complex to achieve these goals. But this complexity is beneficial when implemented in an elegant way. Complexity that arises out of poor planning, redundant functionality, and unclean spaghetti code is bad because it increases development cost without increasing features.

You can reduce complexity in an ATE abstraction framework in four ways:

- **Plan your architecture up front.** As with most development processes, upfront planning and documentation can save a lot of time and hassle during development. By planning and documenting your APIs and code modules up front you can reduce cross-functionality and unnecessary interdependence, which makes your code more robust and reduces unnecessary complexity. You don't have to plan every nuance of every API and code module, but define the major interactions, parameters, and basic functions of the software.
- **Don't think too far ahead.** When developing a large architecture, the tendency is to overdesign and try to plan for all possible scenarios. Although a forward-thinking approach can be good, it is best to design for what is known. All too often, engineers design systems for the worst-case scenario that typically never happens. It's the last 20 percent that takes 80 percent of the time. You will end up spending more time trying to handle presumed edge cases, rather than focusing on the software that will be used most of the time.
- **Give in to the fact that you may not be able to abstract everything.** Abstraction is great, but trying to abstract away every possible interface is an exercise in diminishing returns. Instead, don't preclude custom hardware interactions as part of your framework to account for the times when a generic interface just isn't possible. Set realistic rules for your system that give you the ability to reduce abstraction layers. For example, restrict configuration files to a single format (ini, xls, database) to reduce the complexity of the mapping framework, or restrict actions to three independent hardware calls to prevent the need to implement a recursive Hardware Driver API call.
- **Keep it flexible, scalable, and modular.** Although flexibility, scalability, and modularity do add complexity, they are your best tools for developing large architectures. Here is where plugin architectures are extremely handy, because they define the low-level framework but let the details be implemented by unique code libraries. This means that new functionality can expand on old functionality without breaking pre-existing functions. A well-planned plugin architecture is the epitome of developing for what is known and expanding to new challenges as necessary.

Is It Worth the Effort?

Although the development of an abstraction framework can be time-consuming, even when implemented well, it is done because the payoff is often greater than the development effort. Several key factors can improve the payoff and make your framework more successful. Many of these payoffs can be quantified by the time or effort saved. The table below outlines some typical costs associated with tasks and compares the difference between a nonabstracted system and one that uses a HAL/MAL abstraction framework.

TASK	ESTIMATE (STANDARD)	ESTIMATE (ABSTRACTED)	WHY THE PAYOFF?
Test software platform learning curve for new test engineers	60 hours per engineer	40 hours per engineer	Mastering how to use an abstraction framework typically requires understanding the test executive as well as the framework. In either situation, the developer must understand how to interact with the test system hardware. When instrument-specific drivers are used, the engineer must know the details of each driver and how to use them. However, when learning an abstraction framework, the engineer needs to understand only the high-level actions to be performed, as the instrument details are left to the framework. Typically, these high-level actions are more intuitive and easier to implement than various instrument-specific drivers.
Development and debug of a basic functional test sequence (by an experienced engineer)	80 hours per sequence	40 hours per sequence	Test sequence development becomes much faster because the details of the hardware are stored in a single location, rather than in every driver call within the sequence. Tests interact with hardware from the UUT's point of view, allowing the sequence to be more intuitive and better match the test procedure. In general, an intuitive framework can cut development and debug time in half.
Test sequence development and debug by a new engineer	120 hours per sequence	60 hours per sequence	The payback on development time is amplified when a new or less-experienced engineer develops test sequences. Because the framework imposes a set of rules and functions, less-experienced engineers can better use pre-existing steps to develop sequences when compared to using instrument-specific drivers and code. Furthermore, an intuitive framework allows product-minded test engineers to develop sequences without having to be experts on the underlying software language.
Updating a test sequence for a failed/ obsolete instrument or new instrument requirement	8 hours for driver development plus 4 to 20 hours per sequence	8 hours for driver development plus <1 hour per sequence	When an instrument in the system needs to be replaced, the test must change to account for it. In a nonabstracted platform, this means that every instance of the driver call must be updated for the new instrument. The more the instrument is referenced, the longer this can take. When using an abstracted framework, engineers may need to develop a new instrument driver, but after that is done, only the configuration file/ database needs to be modified.
Moving a test sequence to a new ATE hardware platform	40 to 80 hours per sequence	<8 hours per sequence	Occasionally, entire systems get upgraded and all of the tests must be migrated to the new system. Typically these new systems have very different instrumentation. Whether using an abstraction framework or not, new drivers must be developed, however after those drivers exist, the test sequences must be updated to use them. With a nonabstracted sequence, this is very cumbersome, and can sometimes be easier to write the sequence again from scratch. However, an abstracted sequence can typically be updated in less than a day, all through the configuration file, without having to touch the test sequence software.

Table 4. Costs Associated With Tasks in Nonabstracted and Abstracted Systems

You can use these numbers to expand on the previous scenario with the commercial product company and see if or when it makes sense to develop an integrated abstraction framework.

First, assume that you develop all four test sequences on your own. You must start by developing the framework. In the standard, In the standard, nonabstracted scenario, you must develop instrument-specific drivers. In the second scenario, you focus on using out-of-the-box abstraction when developing the drivers. In the third scenario, you develop an integrated HAL/MAL.

TASK	DEVELOPMENT TIME (STANDARD)	DEVELOPMENT TIME (OUT-OF-THE-BOX ABSTRACTION)	DEVELOPMENT TIME (INTEGRATED HAL/MAL)
Framework/driver development	80 hours	100 hours	500 hours
Test development (4 tests)	80 x 4 = 320 hours	80 x 4 = 320 hours	40 x 4 = 160 hours
New Total	400 hours	420 hours	660 hours

Table 5. An integrated HAL/MAL requires the most up front development effort.

By the time you have completed initial development, the integrated HAL/MAL approach is around 240 hours more than the standard, but out-of-the-box abstraction has cost only about 20 hours more. However, no test program ends after initial development.

Six months later, R&D finds that a few more measurements are required and the 32-channel multiplexer in the system is no longer sufficient, so it is replaced with a 4 x 128 matrix. You must now develop a new driver and update each test sequence to use the matrix instead of the mux. However, if you used a pre-existing abstracted driver, you would not need to do any driver development to handle the new matrix, and the function calls in the sequence wouldn't need to change—only the details. By using an integrated HAL/MAL, the sequence updates would only need to be done in the channel configuration file.

TASK	DEVELOPMENT TIME (STANDARD)	DEVELOPMENT TIME (OUT-OF-THE-BOX ABSTRACTION)	DEVELOPMENT TIME (INTEGRATED HAL/MAL)
New driver development	4 hours	0 hours	0 hours
Update 2 simple test sequences for new matrix	2 x 4 = 8 hours	2 x 2 = 4 hours	2 x 1 = 1 hour
Update 2 complex test sequences for new matrix	2 x 16 = 32 hours	2 x 12 = 24 hours	2 x 2 = 2 hours
Additional Hours	44 hours	28 hours	7 hours
New Total	444 hours	448 hours	667 hours

Table 6. The integrated HAL/MAL method is much easier to update, but still requires more development effort.

Even now, the integrated abstraction layer hasn't paid off yet, although the out-of-the-box hardware abstraction has almost broken even. Now imagine that a new test program comes along that requires you to test four more assemblies. Unfortunately, you are too busy to develop these sequences on your own, and two new test engineers are brought onboard. You must train them on the system and have them develop the sequences.

TASK	DEVELOPMENT TIME (STANDARD)	DEVELOPMENT TIME (OUT-OF-THE-BOX ABSTRACTION)	DEVELOPMENT TIME (INTEGRATED HAL/MAL)
Training/learning curve	60 x 2 = 120 hours	50 x 2 = 100 hours	40 x 2 = 80 hours
Test development (4 tests)	120 x 4 = 480 hours	100 x 4 = 400 hours	60 x 4 = 160 hour
Additional Hours	600 hours	500 hours	240 hours
New Total	1,044 hours	948 hours	907 hours

Table 7. The integrated HAL/MAL approach pays off in the long run when more tests are developed or significant changes are made.

At this point, the initial 500-hour investment in the framework has paid off by about 100 hours over the standard development practice. As new tests are developed, changes are made, and the product life cycle continues, there will be a continual return on the initial investment.

There are also many more subjective payoffs to using abstraction that are difficult to put a number on. The calendar time to develop tests is greatly reduced as well, because a HAL/MAL makes it much easier to develop software before the hardware is fully defined. By maintaining a standard framework, you ensure a single repository where new drivers and measurements can be added, bugs can be managed, and code divergence among engineers can be reduced. Standardization helps keep everyone (test engineers, manufacturing engineers, and technicians) aligned, allowing better support of systems. Although there are countless other advantages, as described in detail in this document, let your abilities and ROI calculations help you understand what level of abstraction makes sense for you.

Next Steps

TestStand

TestStand is industry-standard test management software that helps test and validation engineers build and deploy automated test systems faster. TestStand includes a ready-to-run test sequence engine that supports multiple test code languages, flexible result reporting, and parallel/multithreaded test. Although TestStand includes many features out of the box, it is designed to be highly extensible. As a result, tens of thousands of users worldwide have chosen TestStand to build and deploy custom automated test systems. NI offers training and certification programs that nurture and validate the skills of over 1,000 TestStand users annually.

Learn more about TestStand

About Bloomy

Bloomy provides products and services for electronics functional test; avionics, battery, and BMS hardware-in-the-loop (HIL) testing; aerospace systems integration lab (SIL) data systems; as well as world-class LabVIEW, TestStand, and VeriStand application development. Bloomy is a 24-year NI Alliance Partner, placed in the top Platinum and Select tiers by NI since the program's inception.

Learn more about **Bloomy's UTS Software Suite**, which includes an integrated HAL/MAL