NATIONAL INSTRUMENTS

COMPLETE GUIDE

Fundamentals of Building a Test System

ARTICLES

Modeling the Total Cost of Ownership of an Automated Test System

Selecting Instrumentation

Automated Test System Power Infrastructure

Switching and Multiplexing

Test Executive Software

Hardware and Measurement Abstraction Layers

Rack Layout and Thermal Profiling

Mass Interconnect and Fixturing

Software Deployment

System Maintenance

NATIONAL INSTRUMENTS™

FUNDAMENTALS OF BUILDING A TEST SYSTEM

Modeling the Total Cost of Ownership of an Automated Test System

CONTENTS

Introduction

Development Costs

Deployment Costs

Operation and Maintenance Costs

Financial Analysis Approaches

Practical Scenario

Conclusion

Introduction

Most organizations do not consider production test a top priority, but it is a necessity to prevent major quality issues in the products that represent the company brand in the hands of customers. The costs, however, can be significant and are often greatly misunderstood, especially when there's no easy way to quantify the positive business impact of high-quality products or shortened time to market. But best-in-class organizations are unfazed by this "necessary evil" viewpoint, because they seek to understand the total cost of developing, deploying, and maintaining test systems to get ahead of this perception. And the cost of automated test, in reality, is far more complex than the capital cost of a test rack or even the operator's hourly rate.

In this guide, learn about the tools and insight you need to evaluate your test organization, propose changes where significant cost savings are available, and improve the profitability of your company year over year with smarter investment decisions.

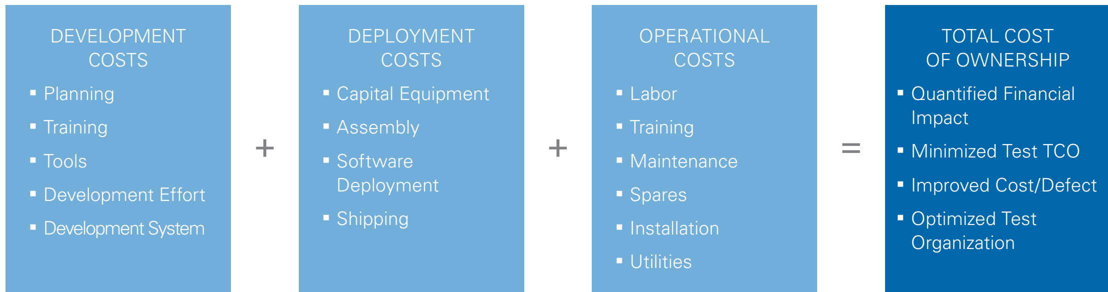


Figure 1. Proper modeling of total cost of ownership uncovers all the lifetime costs of certain test assets and provides a financial framework for justifying future strategic investments.

Development Costs

For most applications, the development costs associated with building a customized automated test system are the smallest in relation to the deployment and operation and maintenance costs. This is typically because only one system is built to serve as a proof of concept for performance benchmarking and test coverage assessment. However, the total cost for developing a test system can vary significantly, depending on the end goal. An organization that is creating a new product often develops and compares multiple test systems with different architectures and instrumentation to identify the optimal approach.

The R&D (engineering) team responsible for a product designs and builds the majority of development systems, and the costs, therefore, are rolled into this budget or cost center. More mature test departments work with their R&D teams to influence the design of products, often referred to as design for test, or DFT, and also work to develop the test systems. This is a best practice but not always possible for test organizations.

For test systems built to test the functionality of a single device or component, the level of effort involved with requirements gathering, instrumentation selection, fixturing, and software development are relatively finite. If, however, a test department is designing a multipurpose, standardized test system to verify the functionality of multiple devices or components, development costs can be greater. You must spend more time identifying all permutations of functionality that the system must accomplish, device under test (DUT) fixturing must be flexible, and the software must be more scalable to make it easy to implement changes when adding new devices to the product portfolio.

Other efforts, such as writing a hardware or measurement abstraction layer or mass interconnect system, require significantly more upfront development cost, but should pay a return on investment for test organizations that either deal with rapid technology change or face instrumentation end-of-life (EOL) issues for long life-cycle systems.

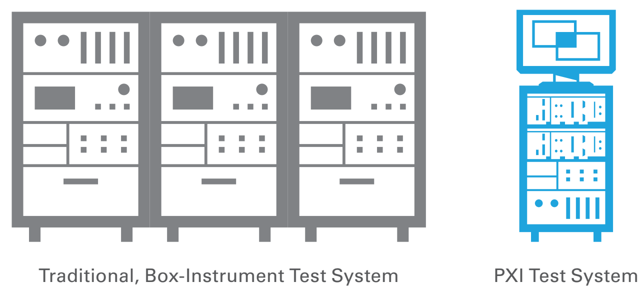
The main costs associated with developing an automated test system are:

- **Planning Effort—**Entails the time and expenses required to properly identify all viable options for the test system. It includes time spent at vendor websites, product demonstration sessions, evaluations, trade shows, and discussion forums.
- **Developer Training—**Includes the time and training course fees associated with learning a new set of software development tools (integrated development environment [IDE] or test executive) and hardware platforms (for example, rack and stack with SCSI or PXI).
- **Development Tools—**The cost associated with purchasing development licenses for the test software (IDE or test executive).
- **Development Effort—**The time associated with the hardware and software development of a proof-of-concept test system.
- **Development System—**The capital cost associated with purchasing the initial proof-of-concept or demonstrator test system for benchmarking against current or other new systems.

Deployment Costs

When you put a product into production, you must scale up the proof-of-concept or demonstrator test system to meet the volume demands of the product. The throughput (units tested per amount of time) of the test system directly impacts the number of systems required to meet demand, and product management and the sales channel determine the forecasted volume. Alongside coverage of test functionality, the required number of test systems is the factor that you should consider most during the development phase because this directly impacts the total deployment cost.

Another factor that increases a test system's deployment cost is shipping. Smaller organizations find this less challenging because the manufacturing test and R&D departments can be collocated in the same building or at least in close geographic proximity. However, even some smaller companies opt to contract the manufacturing and test of their products if they lack the ability or expertize to manufacture and test their devices or components. Larger companies, however, can have manufacturing test and R&D departments located in separate regions within the same country, and even in a completely different country. This can increase deployment costs dramatically, especially if the manufacturing test system is large and/or heavy. Slower freight shipping methods can help to reduce this cost, but only in circumstances where time is not a factor. A best practice is to consider the physical size and weight of any test system during the development phase, especially when comparing two options, as this can bear a significant downstream cost.



Traditional, Box-Instrument Test System

PXI Test System

Figure 2. When selecting between two test systems with similar performance, select the smaller, lighter test system to reduce deployment costs.

The main costs associated with deploying an automated test system are:

- **Capital Equipment—**The number of test systems required, which is determined by the product demand and test system throughput, directly impacts this cost.
- **System Assembly—**The time required to assemble the individual components into a test system, which includes building a 19 in. or 21 in. instrumentation rack or other mechanical enclosure, installing all test instrumentation, connecting cabling and wiring, installing switching and mass interconnect, and fixturing.
- **Software Deployment—**These costs are associated with compiling or building a collection of software components and then exporting these components from a development computer to target machines for execution.
- **Shipping and Logistics—**The size and weight of the test system as well as the quantity of test systems required for the production or manufacturing facility directly impact this cost. The distance travelled and the time window required to receive the shipment also impacts the cost. Depending on the ruggedness of the system, special packaging may be necessary.

Operation and Maintenance Costs

The final and often overlooked or underappreciated costs associated with a test system are the operation and maintenance costs. These are typically not attributed to the R&D team that originally designed the product or device but almost always roll up to the manufacturing or production team; this separation of cost centers makes cross-department collaboration a common pain-point. In situations where a company chooses to contract out the manufacturing and test of its products, the contract manufacturer incurs the individual costs and negotiates a flat or hourly rate for the service.

The costs associated with operating and maintaining an automated test system are:

- **Hourly Operation—**The labor costs for test system operators and support technicians to ensure the systems are up and running during manufacturing. The number of test systems and the skill level required to operate the system directly impact this amount.
- **Operator Training—**The time required for each operator to learn how to use a test system. Costs typically are limited to the amount of time that each operator must attend training, regardless of format (manual, online, or in-person). Companies with a variety of test systems must decide on their staffing strategy between a model of every operator can operate every test system and each operator specializes on a single test system.
- **Maintenance—**The cost associated with keeping the test system and instrumentation in working order. It often includes the cost for annually calibrating equipment, as well as a forecasted cost for replacing instrumentation upon failure. How easy the system is to service can also impact this.
- **Spare Inventory—**The cost required to keep spare instrumentation in the event of unplanned downtime (for example, instrument failure) or planned downtime (for example, calibration). Each test system requires spare instrumentation; companies with multiple unique test systems, because of high product mix, require a larger set of spare instrumentation and parts for their test fleet to ensure high uptime.
- **Installation—**Test systems that consume a lot of power or produce a lot of heat need special, high-power electrical work or cooling towers to be installed to ensure proper performance.
- **Utilities—**The cost associated with powering, cooling, and housing (floor space) the test system. The price per square foot of the manufacturing floor and electricity rate can vary significantly based on geographic location.

Financial Analysis Approaches

Because development and deployment costs are amortized over multiple years and operation and maintenance costs occur in the future, you must use a financial model to determine the total cost of ownership for a test system. For traditional investment scenarios, a project will generate revenue and profit. In this situation, however, there is no revenue or profit but rather a relative savings of one test system in comparison to another. Consider a similar situation that involves investing in high-efficiency lighting or building insulation, which costs money upfront but money will be made through reduced utility expenses in the long run.

- **Payback Period (PP)—**This is the amount of time it takes to recuperate the money you invest in a project. The calculation has two parts. First, you must determine the upfront costs by finding the difference in developing and deploying the new test system and deploying more of the old system. Because the old system has already been developed, there are no associated costs. Second, this difference is divided by the annual savings in operation from the new system's efficiency (throughput).

$$\text{Payback Period (PP) [yr.]} = \frac{\text{Upfront Cost [\$]}}{\text{Annual Savings [\$/yr.]}}$$

- **Return on Investment (ROI)—**This is the ratio of the money earned to the money invested over the life of a project, expressed in a percentage. The calculation is more involved as it requires you to calculate the projected total cost of ownership for both the old and new options, and then find the difference in the two. You then divide this result by the total cost of the more cost-effective option, and subtract 1 (100%) from the quotient to find the resulting percentage.

$$\text{Return on Investment (ROI) [\%]} = \frac{\text{Total Net Savings [\$]}}{\text{Total Cost [\$]}} - 1$$

- **Additional Models—**To determine the viability of projects or financial investments, you can use many additional financial models such as internal rate of return (IRR), net present value (NPV), and modified internal rate of return (MIRR). But most of the advanced modeling that comes with these drops out when comparing two options against one another, and you can simplify the analysis to PP and ROI.

Practical Scenario

The following practical scenario helps demonstrate how you can use financial analysis of total cost of ownership to make an informed decision about purchasing a new test system architecture instead of keeping an old approach.

Overview

Company B is a $200 million manufacturer of IP-based satellite communication systems. Their current production test system is architected using traditional rack-and-stack box instruments. Company B develops and deploys these test systems to a contract manufacturer who charges them a flat rate of $30 per hour to perform product test.

The following best characterizes the current test system:

- Fully functional and full test coverage
- Moderate capital cost
- Organization is fully trained on how to operate
- Throughput is less than optimal

Because Company B recently invested in a larger sales channel and was able to enter new markets for their radar products, their production capacity must increase from 10,000 units to 25,000 units per year.

Figure 3. Increase in demand of 15,000 products, year-over-year.

Their engineering team worked with NI to specify a new PXI-based test system that should result in a 3X improvement in test time per DUT. However, a new solution would require upfront development and deployment costs, so the business impact of the migration must be modeled relative to purchasing additional testers based on the previous architecture before a decision can be made.

Existing Rack-and-Stack System

NRE Capital Investment:	N/A
NRE Development Time:	N/A
Capital Expense:	$100,000 per system
# Existing Test Systems:	10
Test Time:	40 minutes per device
Volume/Throughput:	1,000 devices per year

New PXI-Based System

NRE Capital Investment:	$90,000
NRE Development Time:	$150,000
Capital Expense:	$120,000 per system
# Existing Test Systems:	N/A
Test Time:	13 minutes per device
Volume/Throughput:	3,000 devices per year

Other Financial Variables

Amortization Schedule:	5 years
Replace Existing Systems:	No, keep in operation
Operation Cost per Hour:	$30 (contract manufacturer)
Required Throughput:	25,000 units per year

Development and Deployment Costs

The most common assumption during this evaluation process is that it is more economical to buy additional test systems based on the existing architecture, because the organization is already fully trained and there are no incurred development costs. The system is already architected, and it just needs to be replicated. The new system, however, requires planning, architecting, training, and other nonrecurring engineering (NRE) costs during the development period.

The throughput advantage of the new system, however, cannot be ignored; throughput directly determines the number of additional or new test systems that must be purchased to reach the forecasted increase in volume. In this scenario, scaling up the number of existing test systems requires 15 additional systems whereas buying new PXI-based systems requires only five to meet the production volume.

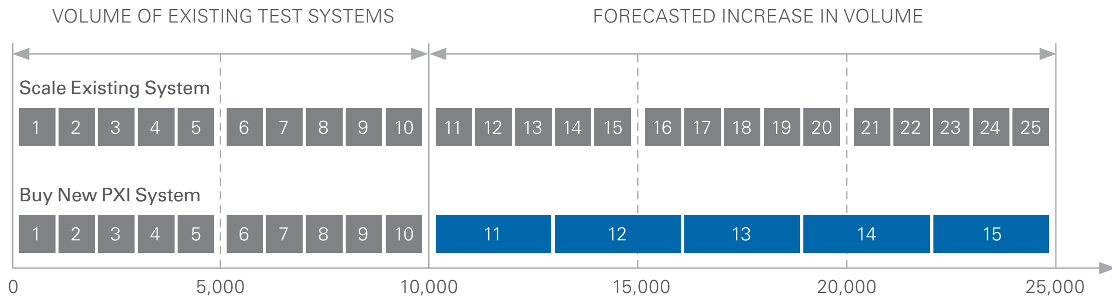


Figure 4. The 3X throughput improvement of the new PXI test system greatly reduces the number of systems required to meet additional product demand.

After determining the number of test systems required for each approach, you can compare the total cost associated with the development and deployment and directly understand the impact of throughput, capital expense, and NRE.

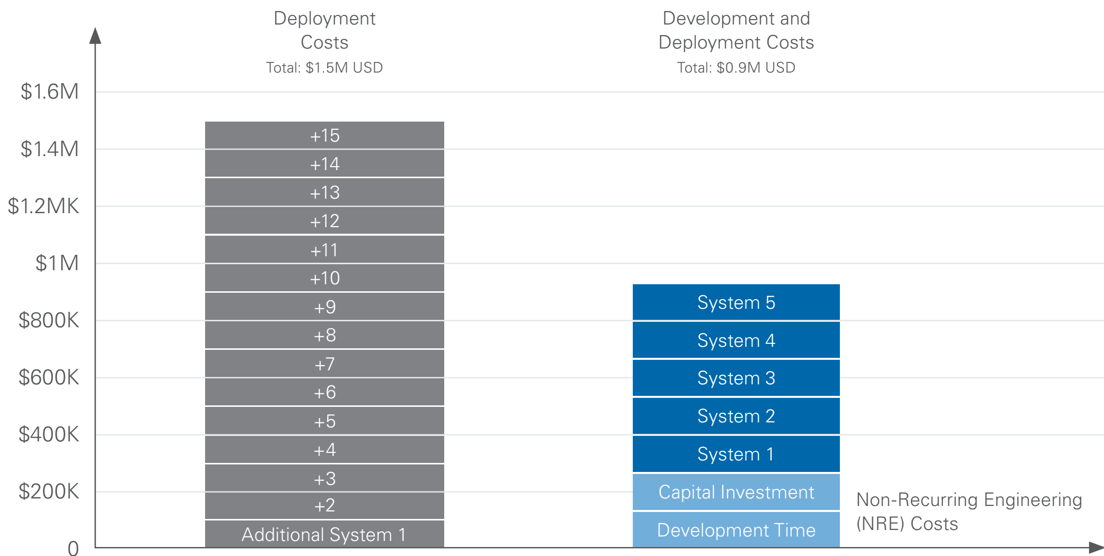


Figure 5. Even though the new PXI-based test system incurs NRE development costs, the total cost of development and deployment for the new system is $600,000 less expensive.

For this given scenario, when comparing the development and deployment costs, buying a new solution is more cost-effective than scaling up the existing test system. The biggest driver of the inflated costs for scaling up the existing system is the low throughput of the system. Throughput alone increases the deployment costs by requiring three times as many test systems to meet the required volume.

But what happens if the variables change? Model different what-if scenarios to ensure that it is a profitable outcome, even in the worst-case scenario.

Some hypothetical scenarios to model:

- What if development time of the new system takes twice a long, and is therefore twice as expensive?
- What if the capital expense increases by 10 percent because of currency inflation?
- What if the throughput improvement is only 1.5X instead of 3X?
- What if the sales volume is revised from 25,000 to only 20,000 units?
- What if additional floor space is limited?
- What if additional power or cooling must be installed at the test facility?
- What if the previous instrumentation is now EOL?

Operation and Maintenance Costs

After you have developed and deployed the required number of test systems, you must operate and maintain them over the length of the project or product life cycle. The costs associated with operating and maintaining a test system are normally attributed to the company's manufacturing group, whereas the development and deployment of a test system are attributed to the R&D (engineering) group. Without guidance from leadership, the engineering team will likely default to cost optimize around development and deployment without considering the implications for the operation and maintenance costs.

In the example above where only development and deployment costs are considered, the new test system is more economical than purchasing additional test systems based on the previous architecture. Now analyze the operation and maintenance costs of the two options over the first five years of the project to understand the impact on the total cost of test.

In this situation, Company B has contracted the manufacturing and test of its products. The contract manufacturer charges Company B $30 per hour to operate the test system.

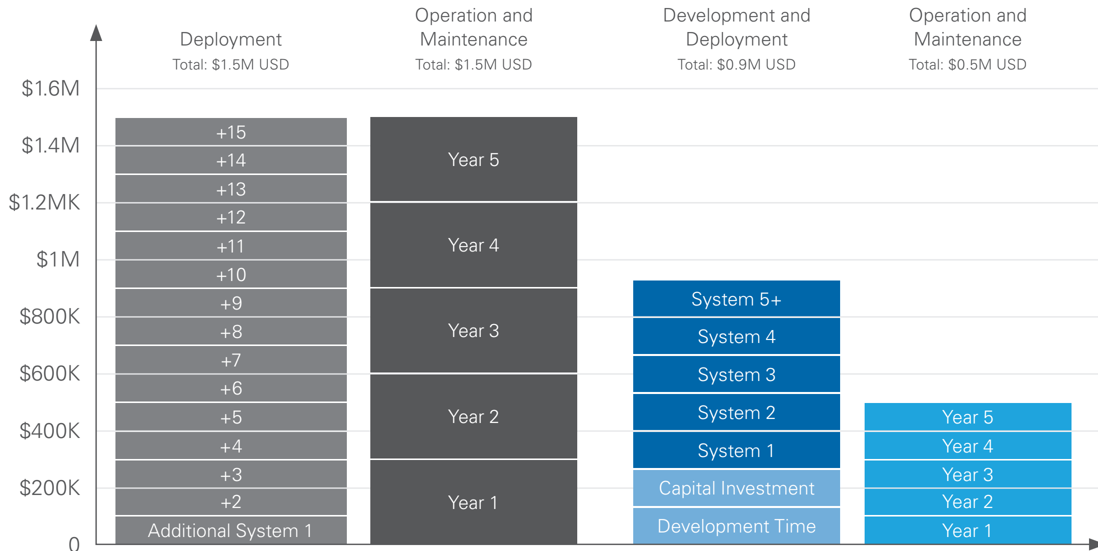


Figure 6. In addition to having much lower development and deployment costs, the operation and maintenance costs of the PXI-based test system are much lower than the previous system's.

Total Cost of Ownership

Although in this scenario the PXI option is the best choice, it is still important to determine the total cost of ownership to effectively model the financial benefit of the new system. This five-year analysis provides insight into variables such as PP, ROI, total savings, and reduction in cost of test on a per part basis. For this analysis, the development and deployment costs are amortized equally (flat) over a five-year period.

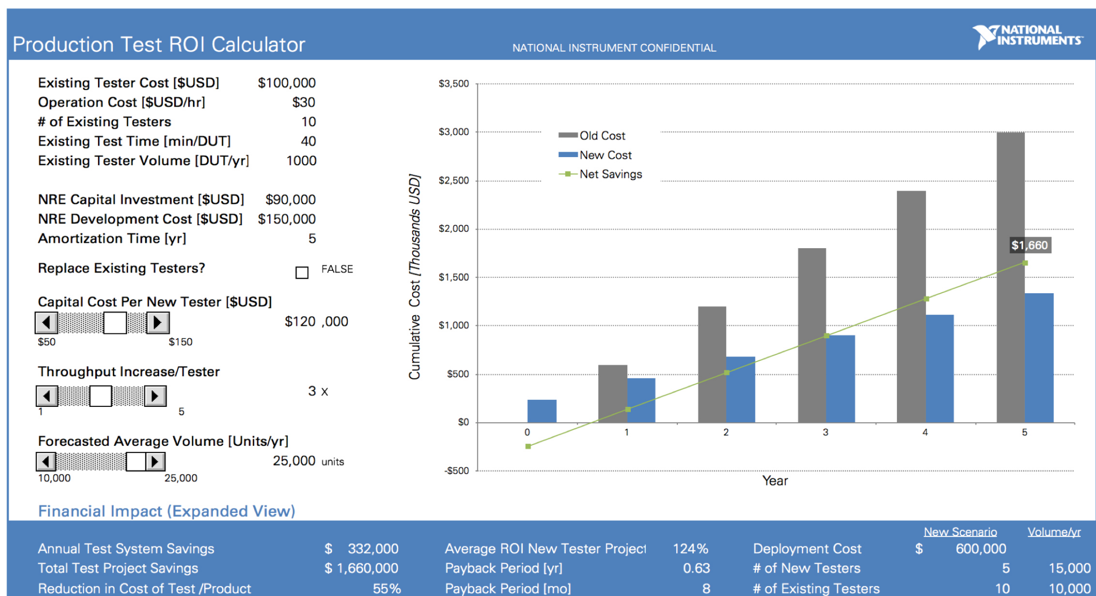


Figure 7. The new test system generates a total savings of $1.66 million in five years with a payback period of 11 months in comparison to scaling up the existing solution.

Scenario Summary

In the case of deciding between these two options for a test system, there are many factors to consider. The common assumption is that scaling up the old solution is easier and cheaper, but further analysis reveals that investing in a newer, higher performance system is a superior financial decision. The biggest factor in the financial advantage of the PXI system is the 3X improvement in throughput—this allowed Company B to purchase one-third as many test systems to accomplish the same task, which saves them money on the capital investment. Over the five-year period, this also reduces the operation and maintenance costs that they pay to the contract manufacturer significantly, resulting in a PP of 11 months and a 124 percent ROI on the project.

Conclusion

As device complexity and time-to-market pressures continue to soar, the total cost of ownership for an automated test system will continue to play an important role in a company's profitability. To realize this goal, you must look beyond the initial capital cost of the test system to ensure that all relevant costs are factored into your purchasing decisions. This guide focuses on automated production test, but you can extrapolate and apply the same concept to other phases of bringing a product from initial concept to the end user, including R&D, characterization, verification, and validation.

As the developers of the PXI platform, LabVIEW graphical system design software, and TestStand test management software as well as a founding member of the PXI Systems Alliance, NI has 40 years of experience helping companies to develop automated test systems for industries ranging from semiconductor production to aerospace and defense. Our direct field engineer team in more than 50 countries worldwide is committed to helping companies, large and small, ensure the highest product quality while reducing the cost of test. To take the next step, contact your local NI representative.

2016 National Instruments. All rights reserved. LabVIEW, National Instruments, NI, NI TestStand, and ni.com are trademarks of National Instruments. Other product and company names listed are trademarks or trade names of their respective companies.

NATIONAL INSTRUMENTS

FUNDAMENTALS OF BUILDING A TEST SYSTEM

Selecting Instrumentation

CONTENTS

Introduction

Analog and RF Instrumentation

Digital Instrumentation

Form Factor

Next Steps

Introduction

Engineers can generally agree on the high importance of the adage "pick the right tool for the job." Using the wrong tool can waste time and compromise quality, whereas the right tool can deliver the correct result in a fraction of the time.

When building automated test systems, the primary tools at your disposal come in the form of measurement instruments. These instruments include known commodities like digital mulitmeters (DMMs), oscilloscopes, and waveform generators as well as a variety of new and changing categories of products like vector signal transceivers and all-in-one oscilloscopes. To select instrumentation, a skilled test engineer must be knowledgeable and proficient in navigating:

- Technical measurement requirements of the device under test (DUT)
- Key instrument specifications that will influence an application
- Various categories of instrumentation available and the trade-offs in capabilities, size, price, and so on
- Nuanced differences between product models within a given instrument category

Picking the right tool for the job is much easier said than done, specifically when it comes to navigating and evaluating the many trade-offs at play. In this guide, see the major categories of instruments available, and learn about common selection criteria to help you narrow in on the best choice for your application.

Analog and RF Instrumentation

The landscape of analog and RF test instruments is very broad with thousands of models across hundreds of product categories. At the same time, it is also predictably governed by the laws of physics—specifically, the fundamentals of noise and bandwidth manifest in the form of amplifier technology and analog-to-digital converters (ADCs) that are used to create instruments. These fundamental physics limitations create a very discernable trade-off in the precision of a measurement compared to the speed at which it can be acquired. Shown below is a view of how that speed versus resolution trade-off has evolved over time as technology has progressed in both traditional and modular instruments.

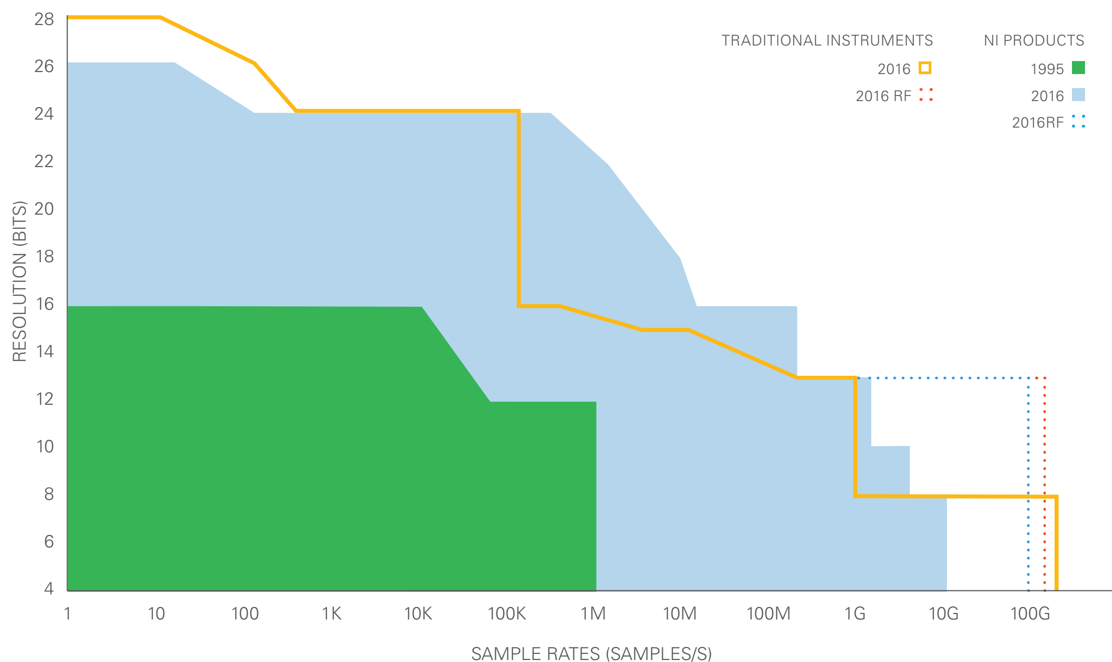


Figure 1. Resolution Versus Sample Rate for Instrumentation

Analog and RF Instrument Categories

The curve in Figure 1 represents examples from a variety of different instrument categories. DMMs provide high accuracy at low speeds at the top left of the chart, oscilloscopes provide high-frequency acquisitions at lower resolutions on the bottom right of the chart, and DAQ products offer higher channel densities and lower cost in the bottom left.

To narrow in on which category of instrument to begin looking into, first consider a couple of key questions about your measurement task:

- **What is the direction of the signal?** (input, output, or both)
- **What is the frequency of the signal?** (DC, kilohertz, megahertz, or gigahertz)

Given the answers to those two key questions on directionality and speed, there's generally a natural starting point for the instrument category you should consider, which Table 1 can best describe.

	DC AND POWER	LOW-SPEED ANALOG	HIGH-SPEED ANALOG	RF AND WIRELESS
INPUT, MEASURE	Digital Multimeter	Analog Input, Data Acquisition (DAQ)	Oscilloscope, Frequency Counter	RF Analyzer Power Meter (Spectrum Analyzer, Vector Signal Analyzer)
OUTPUT, GENERATE	Programmable Power Supply	Analog Output	Function/Arbitrary Waveform Generator (FGEN, AWG)	RF Signal Generator (Vector Signal Generator, CW Source)
INPUT AND OUTPUT ON THE SAME DEVICE	DC Power Analyzer	Multifunction Data Acquisition (Multifunction DAQ)	All-in-One Oscilloscope	Vector Signal Transceiver (VST)
INPUT AND OUTPUT ON THE SAME PIN	Source Measure Unit (SMU)	LCR Meter	Impedance Analyzer	Vector Network Analyzer (VNA)

Table 1. Analog Instrumentation Categories

This chart, although helpful, is far from an exhaustive list of instrument types, especially regarding vertical or specific-purpose instruments. Some noteworthy areas that the table does not cover include:

- Specialty DC instruments such as electrometers, microohmmeters, nanovoltmeters, and so on
- Audio band analysis and generation (also known as dynamic signal analyzers)
- Specialty analog products including pulse generators, pulser/receivers, and more

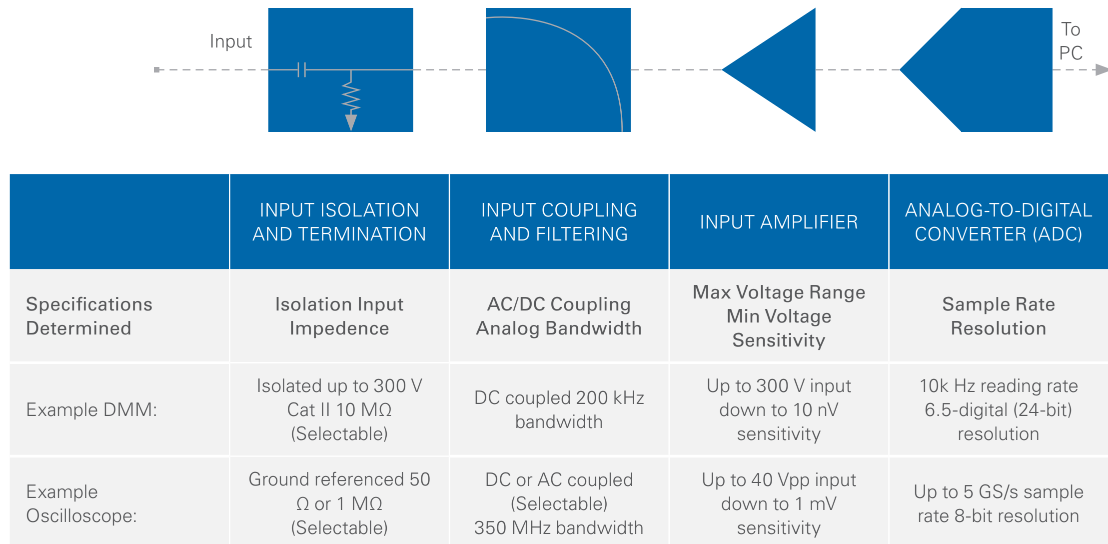
Key Specifications to Consider

After you have narrowed a measurement task to a specific instrument category, the next step is to weigh the trade-offs among products within that category regarding requirements including:

- **Signal ranges, isolation, and impedance—**First, make sure an instrument's input signal range is large enough to capture the signals of interest. Additionally, consider an instrument's input impedance, which affects the loading and frequency performance of the measurement setup, and an instrument's isolation from ground, which impacts noise immunity and safety.
- **Analog bandwidth and sample rate—**Next, make sure that the instrument can pass through the signals of interest based on their analog bandwidth (represented in kilohertz, megahertz, or gigahertz) and that the ADC can sample fast enough to capture the signal of interest (represented in samples per second such as kilosamples per second, million samples per second, or gigasamples per second).
- **Measurement resolution and accuracy—**Finally, evaluate multiple aspects in an instrument's vertical specifications that influence the quality of the measurement such as ADC resolution (digital quantization of analog signals, generally between 8-bit up to 24-bit), measurement accuracy (maximum measurement error over time and temperature, generally expressed in percent or parts per million), and measurement sensitivity (the smallest detectable change, generally expressed in absolute units such as microvolts)

Instruments that don't comprise in these functional dimensions of range, accuracy, and speed will likely present other trade-offs in terms of price, size, power consumption, and channel density—all of which influence an instrument's utility.

Figure 2 shows a simplified view of the analog input path of a generic measurement instrument with four key input stages, the instrument specs those stages influence, and the example instrument specifications of a typical DMM and a typical oscilloscope as influenced by that stage.



	INPUT ISOLATION AND TERMINATION	INPUT COUPLING AND FILTERING	INPUT AMPLIFIER	ANALOG-TO-DIGITAL CONVERTER (ADC)
Specifications Determined	Isolation Input Impedence	AC/DC Coupling Analog Bandwidth	Max Voltage Range Min Voltage Sensitivity	Sample Rate Resolution
Example DMM:	Isolated up to 300 V Cat II 10 MΩ (Selectable)	DC coupled 200 kHz bandwidth	Up to 300 V input down to 10 nV sensitivity	10k Hz reading rate 6.5-digital (24-bit) resolution
Example Oscilloscope:	Ground referenced 50 Ω or 1 MΩ (Selectable)	DC or AC coupled (Selectable) 350 MHz bandwidth	Up to 40 Vpp input down to 1 mV sensitivity	Up to 5 GS/s sample rate 8-bit resolution

Figure 2. Analog Instrumentation Input Stages

The above simplification can be a helpful construct to sift through instrument specifications, which are often presented using a variety of different nomenclatures across instrument categories and across instrument vendors. These stages are often interdependent in influencing key specifications. For instance, the input amplifier can also influence the input bandwidth and the effective resolution of an instrument. Similarly, the input impedance of an instrument can have major effects on the bandwidth.

Analog and RF Instrument Categories

As you compare the measurement requirements of your DUT and the capabilities of the instruments you'll use to test them, keep in mind the following critical ratios.

Test Accuracy Ratio = 4:1

When testing a component, such as a voltage reference, make sure that the accuracy of your measurement equipment is substantially larger than the accuracy of the component being measured. If this criterion is not satisfied, measurement error can be significantly caused by both the DUT and the test equipment, making it impossible to know the true source of error. Because of this, the concept of test accuracy ratio (TAR) is employed to illustrate the relative accuracy of the measurement equipment and the component under test.

Acceptable values for TAR are four and above, depending on the test being performed and the test certainty that is required.

$$TAR = \frac{\text{Wanted Accuracy of the Component Under Test}}{\text{Accuracy of Measurement Equipment}}$$

Bandwidth Ratio = 5:1

Rise time and bandwidth are directly related, and one can be calculated from the other. Rise time defines the time a signal takes to go from 10 to 90 percent of its full-scale value. As a guideline, use the following equation to figure out the bandwidth of your signal based on its rise time:

$$Bandwidth = \frac{0.35}{\text{Rise Time}}$$

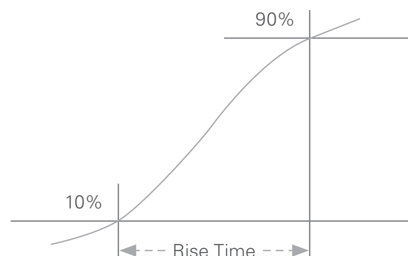


Figure 3. Analog Signal Rise Time

Ideally, you should use a digitizer with three to five times the bandwidth of your signal as calculated in the equation above. In other words, your digitizer's rise time should be 1/5 to 1/3 of your signal's rise time to acquire your signal with minimal error. You can always backtrack to determine your signal's real bandwidth based on the following formula:

$$T_m = \sqrt{T_5^2 + T_d^2}$$

T_m = *measured rise time,* T_5 = *actual signal rise time,* T_d = *digitizer's rise time*

Time Domain Sampling Ratio = 10:1

Whereas bandwidth describes the highest frequency sine wave that can be digitized with minimal attenuation, sample rate is simply the rate at which the ADC in the digitizer or oscilloscope is clocked to digitize the incoming signal. Sample rate and bandwidth are not directly related; however, there is a general rule for the wanted relationship between these two important specifications:

Digitizer's real-time sample rate = 10 times input signal bandwidth

Nyquist theorem states that to avoid aliasing, the sample rate of a digitizer needs to be at least twice as fast as the highest frequency component in the signal being measured. However, sampling at just twice the highest frequency component is not enough to accurately reproduce time-domain signals. To accurately digitize the incoming signal, the digitizer's real-time sample rate should be at least three to four times the digitizer's bandwidth. To understand why, look at the figure below and think about which digitized signal you would rather see on your oscilloscope.

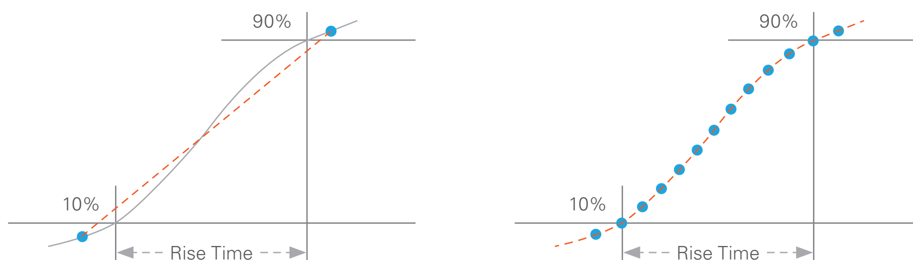


Figure 4. The image on the right shows a digitizer with a sufficiently high sample rate to accurately reconstruct the signal, which will result in more accurate measurements.

Although the actual signal passed through the front-end analog circuitry is the same in both cases, the image on the left is undersampled, which distorts the digitized signal. On the contrary, the image on the right has enough sample points to accurately reconstruct the signal, which results in a more accurate measurement. Because a clean representation of the signal is important for time domain applications such as rise time, overshoot, or other pulse measurements, a digitizer with a higher sample benefits these applications.

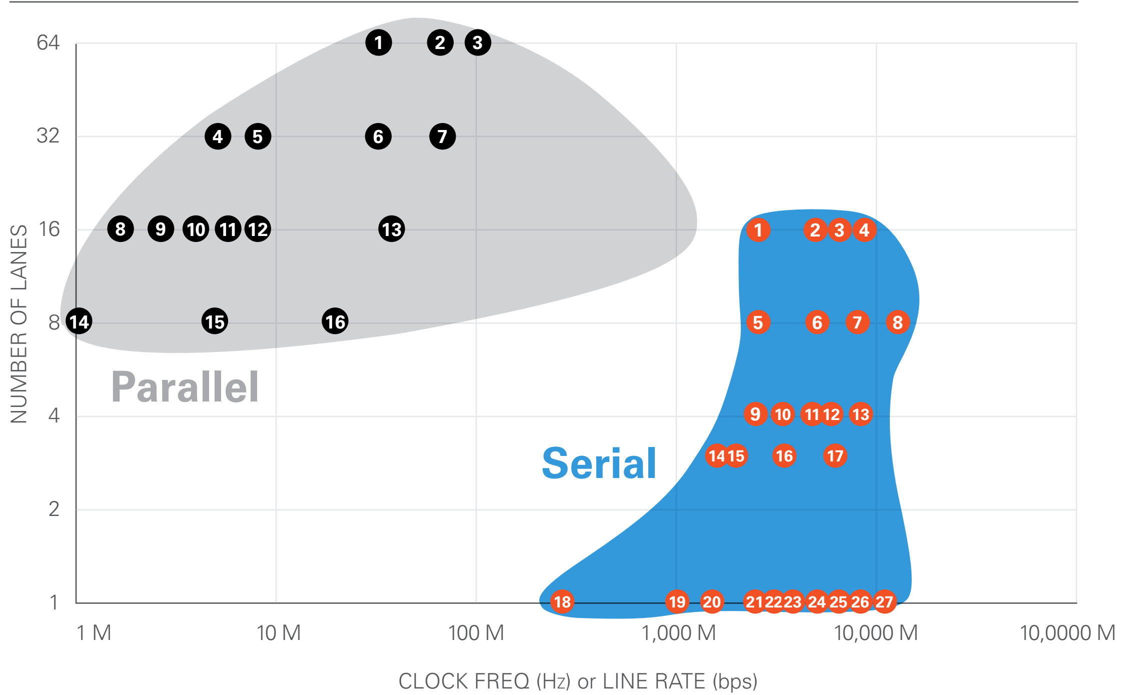
Digital Instrumentation

In the context of electronic functional test, digital instrumentation serves the purpose of interfacing with digital protocols and testing the electrical characteristics and communication link characteristics of those protocols. One of most critical aspects influencing the available instrumentation for a given task is parallel versus serial digital communications.

Parallel Versus Serial Standards

Serial standards have been gaining in popularity because of the physical limitation on the clock rates of parallel buses at around 1 GHz to 2 GHz. This is because of skew introduced by individual clock and data lines that cause bit errors at faster rates. High-speed serial buses send encoded data that contains both data and clocking information in a single differential signal, allowing you to avoid speed limitations in parallel buses. Serializing the data and sending at faster speeds allows pin counts of integrated circuits (ICs) to be reduced, which helps decrease size. Furthermore, because the serial lanes can operate at a much faster clock speed, they can also achieve better data throughput than what was possible with parallel buses.

BUS STANDARDS



Parallel Bus

1. PCI 64-bit/33 MHz
2. PCI 64-bit/66 MHz
3. PCI 64-bit/100 MHz
4. Front Panel Data Port
5. EISA
6. PCI 32-bit/33 MHz
7. PCI 32-bit/66 MHz
8. IDE (ATA PIO 0)
9. ATA PIO 1
10. ATA PIO 2
11. ATA PIO 3
12. ATA PIO 3
    ISA 16-bit/8.33 MHz
13. Ultra-2 wide SCSI
14. RapidIO Gen1.1
15. GPIB
16. SCSI
    ISA 8-bit/4.77 MHz

Serial Bus

1. PCIe Gen1x16
2. PCIe Gen2x16
3. Serial RapidIO Gen2
4. PCIe Gen3x16
5. PCIe Gen1x8
6. PCIe Gen2x8
7. PCIe Gen3x8
8. JESD204B
9. PCIe Gen1x4
10. Serial RapidIO Gen1.3
11. PCIe Gen2x4
12. DisplayPort
13. PCIe Gen3x4
14. HDMI 1.0
    DVI
15. HDMI 1.3
16. HDMI 2.0
17. SD-SDI
18. Gigabit Ethernet
19. SATA 1.0
20. Serial FPDP
    PCIe Gen1x1
21. SATA 2.0
    3G-SDI
    JESD204A
    10 Gigabit Ethernet
22. PCIe Gen2x1
    USB 3.0
23. SATA 3.0
24. PCIe Gen3x1
25. USB 3.1

Figure 5. This chart shows well-known bus standards and their respective numbers of lanes versus line rates. The serial standards are capable of much higher line rates than the parallel standards, leading to higher throughput.

Digital Instrument Categories

As with analog instrumentation, you can quickly narrow your options for digital instrumentation using a couple of key questions:

- **What task do you need to accomplish?** (digital interfacing, custom digital interfacing, or electrical and timing test)
- **How fast is the link?** (static and kilobit per second range, megabit per second, or gigabit per second)

	STATIC, LOW SPEED	SYNCHRONOUS AND HIGH-SPEED PARALLEL (100 MBITS/S RANGE)	HIGH-SPEED SERIAL (10 GBITS/S RANGE)
INTERFACE (STANDARD)	Low-Speed Standard Interface Card (I2C, C) Synchronous Protocol Interface (ARINC 429, CAN, GPIB, I2C, SPI)		Interface Card (10 Gigabit Ethernet, Fibre Channel, PCI Express, and so on)
INTERFACE (CUSTOM)	Digital I/O (GPIO)	Digital Waveform Generator/ Analyzer, Pattern Generator	FPGA-Based High-Speed Serial Interface Aurora, Serial Rapid I/O, JESD204b
ELECTRICAL TEST AND TIMING TEST (BASIC INTERFACE)	Pin Electronics Digital, Per-Pin Parametric Measurement Unit (PPMU)		BERT, Oscilloscope

Table 2. Digital Instrumentation Categories

Hardware Versus Software Timing

You can implement digital communication schemes using two main methods: software timing and hardware timing. Software-timed applications do not use any type of clock for input or output. The software controls the I/O, and a programming language controls the timing through software. This programming language typically runs on an OS, which could take up to milliseconds to execute software calls. For software timing, you use the OS timer to determine the rate of timed actions. Generally, low-speed applications, such as monitoring and controlling alarms, motors, and enunciators, use software timing.

You can choose from two types of software-timed communication: deterministic control and nondeterministic control. Using a real-time OS, you can achieve precision of up to 1 μs; however, real-time OSs do not make communications faster, only more deterministic. Non-real-time systems, such as Microsoft Windows, are nondeterministic. In these systems, the time taken for software commands to execute in hardware is inconsistent and could take multiple milliseconds. Factors such as computer memory, processor speed, and other applications running on the OS could affect the execution time.

Hardware-timed devices, in contrast, use the rising or falling edges of a clock for deterministic generation or acquisition. You can use this kind of timing to acquire or generate digital data at rates in the gigabit per second range with very high determinism, and you can reliably output data at predetermined locations.

Applications that use hardware timing include the following:

- Chip testing
- Protocol emulation and testing
- Digital video and audio testing
- Digital electronics testing

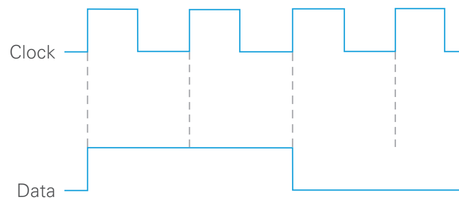


Figure 6. With hardware-timed operations, you can take advantage of real-time, deterministic digital signal output.

Clock Rate

An important consideration for hardware-timed digital applications is clock speed. The maximum speed that a device can achieve is difficult to compensate for if it is inadequate. You can achieve up to 200 MHz sampling rates for single-ended signals and up to 200 MHz for differential signals using NI high-speed digital I/O devices, thus enabling tests including protocol, digital audio and video, and digital electronic. For scenarios where a device might not meet the necessary clock rate requirements on a serial data stream, you can use serializers/deserializers (SERDES) to acquire higher frequency digital signals. However, depending on the type of SERDES you use, incorporating a SERDES might reduce the number of available lines.

Form Factor

In addition to understanding the analog front end required to physically make the correct measurement, you need your instruments to be stable, repeatable, fast, and PC-connected—that's part of the job. This brings you to a decision regarding the setting/environment:

- **For the bench and lab—**Accuracy, repeatability, low-level control, ease of setup, and ability to automate for repetitive tests
- **For the manufacturing floor—**Speed, throughput, accuracy, optimization through programming interface, and debugging

Clearly, there are similarities and differences in how you'd select instrumentation across the lab versus the manufacturing floor. You typically evaluate instrument form factors across a set of key success criteria for the end deployment. Below is a typical set of evaluation criteria you might see for a manufacturing environment.

FUNCTIONAL NEEDS	TEST ENGINEERING NOTES
Instrumentation, I/O needed?	
Processing, compute needed?	
Data throughput, storage?	
Synchronization?	
Future requirements?	
Number of systems deployed over number of years?	
Years of planned sustainment?	
Number of global sites replicating?	
Environmental stability of deployment scenarios?	
How is the initial setup, configuration, and repair managed?	
Rack mounted?	
Size, weight, and power?	
Fixture and connectivity?	

Table 3. Hardware Deployment Checklist

Selecting Bus Type

Today, USB, PCI Express, and Ethernet/LAN have gained attention as attractive communication options for instrument control. Some test and measurement vendors and industry pundits have claimed that one of these buses, by itself, represents a solution for all instrumentation needs. In reality, it is most likely that multiple bus technologies will continue to coexist in future test and measurement systems because each bus has its own strengths.

Bandwidth

When considering the technical merits of alternative buses, bandwidth and latency are two of the most important bus characteristics. Bandwidth measures the rate at which data is sent across the bus, typically in megabytes per second. A bus with high bandwidth can transmit more data in a given period than a bus with low bandwidth. Most users recognize the importance of bandwidth because it affects whether their data can be sent across the bus to or from a shared host processor as fast as it is acquired or generated and how much onboard memory their instruments will need. Bandwidth is important in applications such as complex waveform generation and acquisition as well as RF and communications applications. High-speed data transfer is particularly important for virtual and synthetic instrumentation architectures. The functionality and personality of a virtual or synthetic instrument is defined by software; in most cases, this means data must be moved to a host PC for processing and analysis. Figure 7 charts the bandwidth (and latency) of all the instrumentation buses examined in this guide.

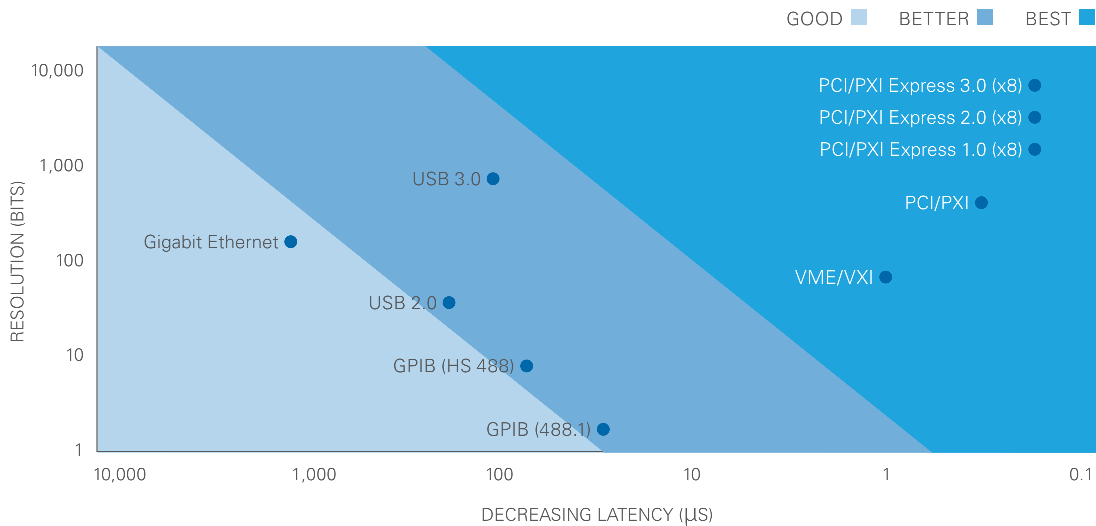


Figure 7. Bandwidth Versus Latency for Instrumentation Buses

Latency

Latency measures the delay in data transmission across the bus. By analogy, if you were to compare an instrumentation bus to a highway, bandwidth would correspond to the number of lanes and the speed of travel, while latency would correspond to the delay introduced at the on and off ramps. A bus with low (meaning good) latency would introduce less delay between the time data was transmitted on one end and processed on the other. Latency, while less observable than bandwidth, has a direct impact on applications where a quick succession of short, choppy commands is sent across the bus, such as in handshaking between a DMM and switch, and in instrument configuration.

GPIB

The IEEE 488 bus—commonly known as GPIB—is a proven bus designed specifically for instrument control applications. GPIB has been a robust, reliable communication bus for 30 years and is still the most popular choice for instrument control because of its low latency and acceptable bandwidth. It currently enjoys the widest industry adoption with a base of more than 10,000 instrument models with GPIB connectivity.

With a maximum bandwidth of about 1.8 Mbytes/s, it is best suited for communicating with and controlling stand-alone instruments. The more recent, high-speed revision, HS488, increased bandwidth up to 8 Mbytes/s. Transfers are message-based, often in the form of ASCII characters. Multiple GPIB instruments can be cabled together to a total distance of 20 m, and bandwidth is shared among all instruments on the bus. Despite relatively lower bandwidth, GPIB latency is significantly lower (better) than that of USB and especially Ethernet. GPIB instruments do not autodetect nor autoconfigure when connected to the system; though GPIB software is among the best available, and the rugged cable and connector are suitable for the most demanding physical environments. GPIB is ideal for automating existing equipment or for systems requiring highly specialized instruments.

USB

USB has become popular in recent years for connecting computer peripherals. That popularity has spilled over into test and measurement with an increasing number of instrument vendors adding USB device controller capabilities to their instruments. Though most laptops, desktops, and servers may have several USB ports, those ports usually all connect to the same host controller, so the USB bandwidth is shared among all the ports.

Latency for USB falls into the better category (between Ethernet at the slow end and PCI and PCI Express at the fast end), and cable length is limited to 5 m. USB devices benefit from autodetection, which means that unlike other technologies, such as LAN or GPIB, USB devices are immediately recognized and configured by the PC when a user connects them. USB connectors are the least robust and least secure of the buses examined here. External cable ties may be needed to keep them in place.

USB devices are well suited for applications with portable measurements, laptop or desktop data logging, and in-vehicle data acquisition. The bus has become a popular communication choice for stand-alone instruments because of its ubiquity on PCs and especially its plug-and-play ease of use. The USB Test and Measurement Class (USBTMC) specification addresses the communication requirements of a broad range of test and measurement devices.

PCI

PCI and PCI Express achieve the best bandwidth and latency specifications among all the instrumentation buses examined here. PCI bandwidth is 132 Mbytes/s, with that bandwidth shared across all devices on the bus. PCI latency performance is outstanding—benchmarked at 700 ns, compared to 1 ms in Ethernet. PCI uses register-based communication. Unlike the other buses mentioned here, PCI does not cable to external instruments. Instead, it is an internal PC bus used for PC plug-in cards and in modular instrumentation systems, such as PXI, so distance measures do not directly apply. Nonetheless, the PCI bus can be extended by up to 200 m by the use of NI fiber-optic MXI interfaces when connecting to a PXI system. Because the PCI connection is internal to the computer, it is probably fair to characterize the connector robustness as being constrained by the stability and ruggedness of the PC in which it resides.

PXI modular instrumentation systems, which are built around PCI signaling, enhance this connectivity with a high-performance backplane connector and multiple screw terminals to keep connections in place. Once booted with PCI or PXI modules in place, Windows automatically detects and installs the drivers for modules. In general, PCI instruments can achieve lower costs because they rely on the power source, processor, display, and memory of the PC that hosts them rather than incorporating that hardware in the instrument itself.

PCI Express

PCI Express is similar to PCI. It is the latest evolution of the PCI standard. Therefore, much of the above evaluation of PCI applies to PCI Express as well.

The main difference between PCI and PCI Express performance is that PCI Express is a higher bandwidth bus and gives dedicated bandwidth to each device. Of all the buses covered in this guide, only PCI Express offers dedicated bandwidth to each peripheral on the bus. GPIB, USB, and LAN divide bandwidth across the connected peripherals. Data is transmitted across point-to-point connections called lanes at 250 Mbytes/s per direction for Gen 1 link. Each PCI Express link can be composed of multiple lanes, so the bandwidth of the PCI Express bus depends on how it is implemented in the slot and device. A x1 (by 1) link provides 250 Mbytes/s, a x4 link provides 1 Gbyte/s, and a x16 link provides 4 Gbytes/s dedicated bandwidth. PCI Express achieves software backward compatibility, meaning that users moving to the PCI Express standard can preserve their software investments in PCI. PCI Express is also extensible by external cabling.

High-speed, internal PC buses were designed for rapid communication. Consequently, PCI Express is an ideal bus choice for high-performance, data-intensive systems where large bandwidth is required and for integrating and synchronizing several types of instruments.

Ethernet/LAN/LXI

Ethernet has long been an instrument control option. It is a mature bus technology and has been widely used in many application areas outside test and measurement. 100BASE-T Ethernet has a theoretical max bandwidth of 12.5 Mbytes/s. Gigabit Ethernet, or 1000BASE-T, increases the max bandwidth to 125 Mbytes/s. In all cases, Ethernet bandwidth is shared across the network. At 125 Mbytes/s Gigabit Ethernet is theoretically faster than Hi-Speed USB, but this performance quickly declines when multiple instruments and other devices are sharing network bandwidth. Communication along the bus is message-based with communication packets adding significant overhead to data transmission. For this reason, Ethernet has the worst latency of the bus technologies featured in this guide.

Nonetheless, Ethernet remains a powerful option for creating a network of distributed systems. It can operate at distances up to 85 m to 100 m without repeaters and has no distance limits with repeaters. No other bus has this range of separation from the controlling PC or platform. As with GPIB, autoconfiguration is unavailable on Ethernet/LAN. You must manually assign an IP address and subnet configuration to your instrument. Like USB and PCI, Ethernet/LAN connections are ubiquitous in modern PCs. This makes Ethernet ideal for distributed systems and remote monitoring. It is often used in conjunction with other bus and platform technologies to connect measurement system nodes. These local nodes may themselves be composed of measurement systems relying on GPIB, USB, and PCI. Physical Ethernet connections are more robust than USB connections, but less so than GPIB or PXI.

LAN eXtenstions for Instrumentation (LXI) is an emerging LAN-based standard. The LXI standard defines a specification for stand-alone instruments with Ethernet connectivity that adds triggering and synchronization features.

Despite the conceptual convenience of designating a single bus or communication standard as the ultimate or ideal technology, history shows that several alternative standards are likely to continue to coexist, because each bus technology has unique strengths and weaknesses. Table 4 compiles the performance criteria from the previous section. It should be clear that no single bus is superior across all measures of performance.

	BANDWIDTH (MBYTES/S)	LATENCY (µS)	RANGE (M) (WITHOUT EXTENDERS)	SETUP AND INSTALLATION	CONNECTOR RUGGEDNESS
GPIB	1.8 (488.1) 8 (HS488)	30	20	Good	Best
USB	60 (USB 2.0)	Analog Output	5	Best	Good
PCI (PXI)	132	0.7	Internal PC Bus	Better	Better Best (for PXI)
PCI EXPRESS (PXI EXPRESS)	250 (x1) 4,000 (x16)	0.7 (x1) 0.7 (x4)	Internal PC Bus	Better	Better Best (for PXI)
ETHERNET/ LAN/LXI	12.5 (Fast) 125 (Gigabit)	1,000 (Fast) 1,000 (Gigabit)	100 m	Good	Good

Table 4. Bus Performance Comparison

You can exploit the strengths of several buses and platforms by creating hybrid test and measurement systems that combine components from modular instrumentation platforms, such as PXI and stand-alone instruments, that connect across GPIB, USB, and Ethernet/LAN. One key to creating and maintaining a hybrid system is implementing a system architecture that transparently recognizes multiple bus technologies and takes advantage of an open, multivendor computing platform, such as PXI, to achieve I/O connectivity.

The other key to successfully developing a hybrid system is ensuring that the software you choose at the driver, application, and test system management levels is modular. Though some vendors may offer vertical software solutions for specific instruments, the most useful system architecture is one that breaks up the software functions into interchangeable modular layers so that your system is neither tied to a particular piece of hardware or to a particular vendor. This layered approach provides the best code reuse, modularity, and longevity. For example, Virtual Instrument Software Architecture (VISA) is a vendor-neutral software standard for configuring, programming, and troubleshooting instrumentation systems comprising GPIB, serial (RS232/485), Ethernet, USB, and/or IEEE 1394 interfaces. It is a useful tool because the API for programming VISA functions is similar for a variety of communication interfaces.

With hybrid systems, you can combine the strengths of many types of instruments, including legacy equipment and specialized devices. Despite the appeal of finding a one-size-fits-all solution for instrumentation, reality requires that you fit the instruments and associated bus technologies to your specific application needs.

Timing and Synchronization

You can find a good example of integrated timing and synchronization between instruments in the PXI platform, a modular standard for test and measurement. PXI Express maintains the 10 MHz backplane clock as well as the single-ended PXI trigger bus and length-matched PXI star trigger signal that the original PXI specification provides. PXI Express also adds a 100 MHz differential clock and differential star triggers to the backplane to offer increased noise immunity and industry-leading synchronization accuracy (250 ps and 500 ps of module-to-module skew, respectively). NI timing and synchronization modules are designed to take advantage of the advanced timing and triggering technology featured in its PXI and PXI Express chassis.

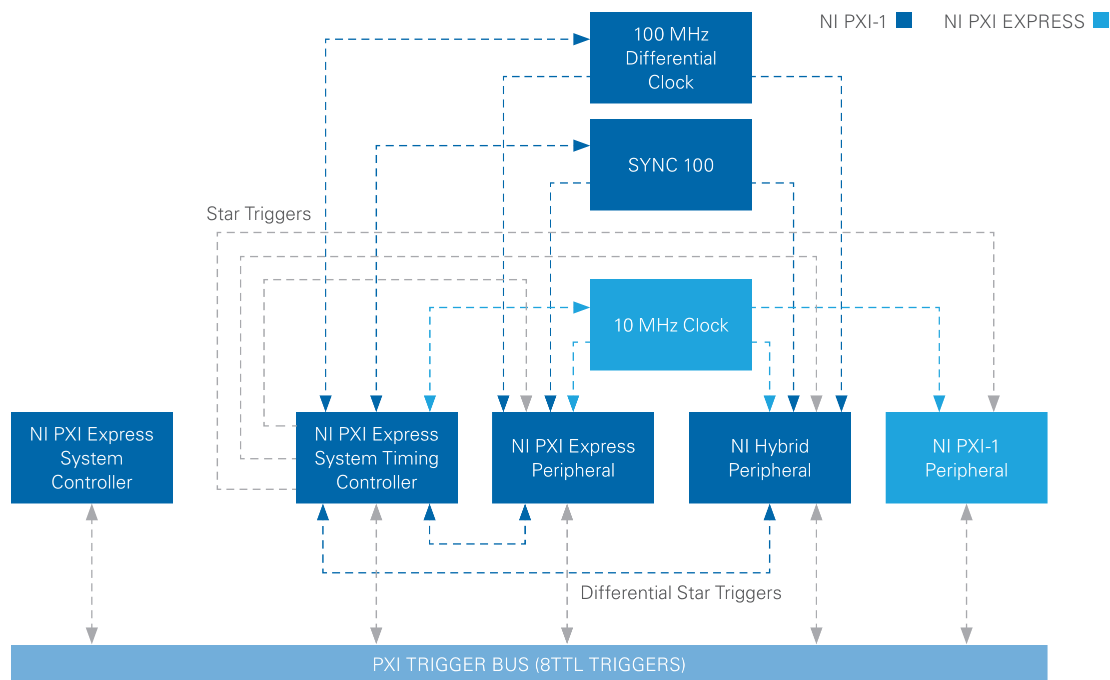


Figure 8. Example of PXI Chassis Timing and Synchronization Features

Next Steps

Learn more about the basics of using test and measurement instrumentation by reading the white paper series, Instrument Fundamentals. This series covers topics ranging from analog sampling theory to grounding considerations for improved measurements.

2016 National Instruments. All rights reserved. National Instruments, NI, and ni.com are trademarks of National Instruments. Other product and company names listed are trademarks or trade names of their respective companies.

NATIONAL INSTRUMENTS

FUNDAMENTALS OF BUILDING A TEST SYSTEM

Automated Test System Power Infrastructure

CONTENTS

Introduction

Introducing Power to the System

Geographic Location Considerations

Elctromagnetic Interference or Line Filters

Power Budget

Power Distribution Unit

Uninterruptable Power Supply

Power States

Grounding

Best Practices for Components

Introduction

Powering an automated test system, or automated test equipment (ATE), is different from powering the PC and lamp on your desk. Test systems are composed of many heterogeneous internal components, some of which require large amounts of current and power, and these systems are often deployed globally into facilities with differing power sources and quality. Many test system components are sourced from multiple vendors and must be integrated by the test engineers, which complicates matters even more. Choosing the right components and making the right design decisions is much simpler when you follow best practices in power layout and equipment selection.

A power layout ensures all components operate properly by avoiding bottlenecks where a component may need more power than the power distribution can provide. This is especially important for components that could compromise operation of the whole system if starved of power. This guide covers test system power planning by listing the steps and considerations for creating a power layout.

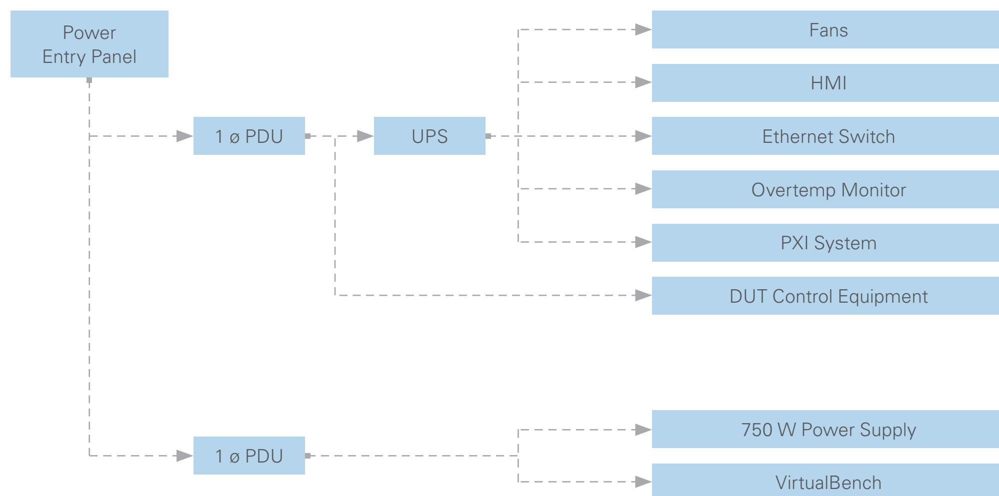
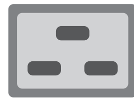


Figure 1. A power layout includes all equipment in the test system and maps the flow of power from the source to the test system to the end consumer.

Introducing Power to the System

A best practice for introducing power to the ATE system is to use a power entry panel, or power inlet panel. This allows you to isolate the internal power cabling from the point at which the main voltage is applied. With a power entry panel, you can outfit your test system with the proper power connector rated for the voltage and current that will be powering the system. NI power entry panels use a number of connector types and power ratings for a variety of power requirements and geographic coverage. Figure 1 shows examples of power panel connectors. A good power panel should also have built-in circuit protection, including a circuit breaker and fuses, which protects the system from power surges or incorrect supply power. A great power panel has a built-in electromagnetic interference (EMI) filter, surge suppression, and other connectivity to pass signals into the system.

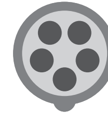


Low-Power Configuration

- IEC 60320 (C20)
- 1ø (16 A) 100-240 V

Medium-Power Configuration

- IEC 60309
- 1ø (32 A) 100-240 V



High-Power Configuration

- IEC 60309
- 3ø (16 A) 380-480 V (Red:3P+N+E)

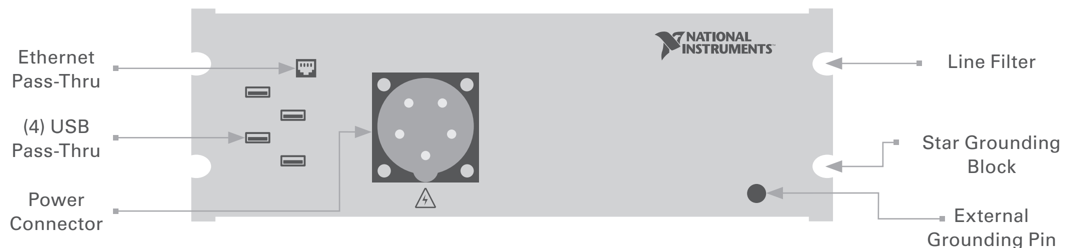


Figure 2. A power entry panel provides connectivity for incoming power to the system. Power entry panels can have one of a number of standard power connector types and good power panels have additional features like filtering or kill-switch relays.

Geographic Location Considerations

The geographic location of the tester or test facility is a critical detail in choosing the power panel for your test system. Additionally, when planning a new test system, consider the power standards and grid infrastructure, safety requirements, and ease of deployment, which are all factors that location can affect.

Power Grid Standards

The line power available from the public grid differs from country to country. Countries around the world have standardized on different RMS voltage levels, AC power frequency, connectors, and current ranges in their power grids. There are several types of power configurations in public grids:

- **Single-phase power** uses a single active line that conducts an AC power and a neutral line. Common voltage levels of these lines vary from 100 V to 240 V. For instance, line power in Japan is 100 V, while power is delivered between 220 V and 240 V. The United States and Canada transmit power in public grids at 110 V to 120 V.
- **Dual- or split-phase power** is composed of two active lines that supply power at a given positive and negative offset voltage and one neutral wire. A common implementation of dual-phase power in the United States is 120 V with a 180 degree offset between active lines. Having two wires that are transmitting power with voltage levels of 120 V and -120 V allows you to have two single-phase sources of power with 120 V of potential by using each active line and the neutral line or one single-phase source with 240 V of potential by using the two active lines.
- **Three-phase power** is made up of three active lines that are 120 degrees offset from one another and one neutral wire. Most US buildings use 208 Y/120 V power, which has three lines that conduct 120 V power and a constant power circuit output of 208 V. Many industrial building use 480 Y/277 V, which provides 480 V for larger machinery.

Global Deployment

Test systems are often designed and deployed in separate or multiple locations. Having a single system deployed in multiple locations introduces new sets of system requirements. Deploying a system to Malaysia is different from deploying a system to a factory in the same country or even the same building. For example, you may build a test system for automotive engine control units at an R&D facility in Detroit but deploy it to factories in Mexico. Consider the power grid standards and quality when designing the system and confirm that all safety and regulatory certifications needed to deploy in Mexico are met before you ship the system. Here is a checklist of items to think through when designing a test system that will be globally deployed:

- Power grid voltage standard and configuration
- Power grid quality and reliability
- Materials compliance like RoHS
- Energy compliance like CE, PSE, or KC
- Trade compliance and import/export regulations

If you plan to deploy the test system to countries or regions outside the test system's country of origin, know the available power in the location(s) that the test system will be deployed and if you need to convert that power to operate the equipment in your test system. In the example above, the test systems were going to Malaysia and Mexico. Luckily, the power grids in both the United States and Mexico provide power at 110 V to 120 V and 60 Hz. This gets a bit more complex for a test station designed in Germany and deployed to Mexico where mains voltages are different.

Power converters and uninterruptable power supplies (UPSs) can help you to condition standard power to meet the needs of the system. For example, a test system that includes equipment that accepts only 120 V may need to include a power converter to turn 230 V single-phase power into a single-phase 120 V supply for the instrumentation. Better yet, evaluate and select equipment that has global input voltage to avoid the hassle altogether.

Certification

Many countries have specific required electrical safety standards like CE in Europe, PSE in Japan, or KC in Korea. Compliance testing for electrical test equipment usually includes emissions level and frequency, touch safety, and surge protection. The most important reason to get these markings is to be able to deploy systems to other countries or certify a factory for operation. Do the necessary research to know the required certifications in each country in which the test system will operate. Ignoring the certifications could make it problematic to service test systems in the future. Individual components cannot be imported unless they have these marks, so it is difficult to replace or repair parts that lack proper certification.

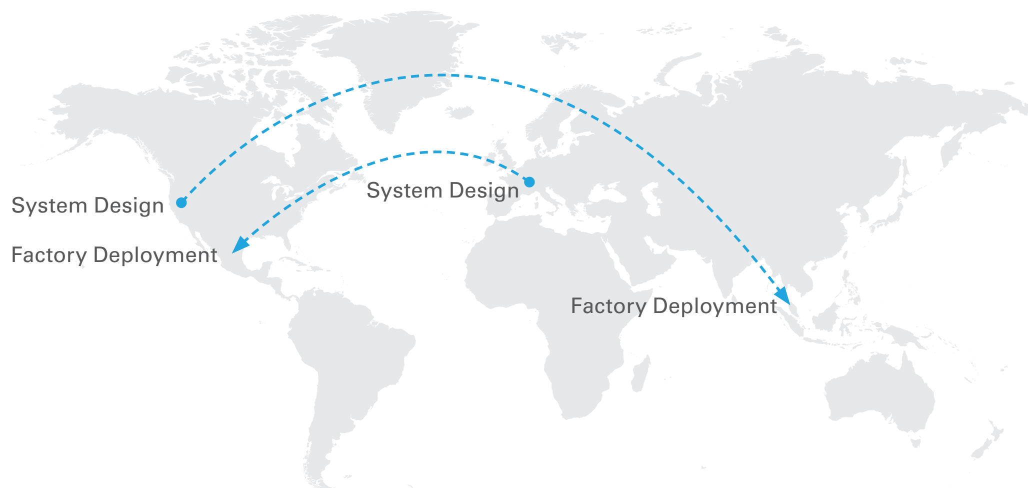


Figure 3. Designing and deploying test systems in multiple countries requires you to be flexible in designing your system. Consider power standards and certifications whenever you are developing a test system that could be used in multiple locations.

Electromagnetic Interference or Line Filters

Power grids carry high-energy signals that emit electromagnetic noise. Most noise from power lines is relatively consistent and you can plan for it in advance. No grid is perfect, however, so there will most likely be some nonstandard noise in the power signal. Nonstandard noise can affect the measurements taken by instrumentation in the system or cause the system to violate certification requirements. EMI and line filters are the most common ways to protect the test system from unexpected noise sources emanating from power transmission lines. A line filter is specified for a given voltage level, a maximum current level that should not be exceeded, and an operating range for frequencies it filters from the signal. For example, a line filter may be designed for 250 V, 10 A, and operate from 150 kHz to 1 MHz. Be sure to choose the right line filter for the power and unwanted noise frequencies in your test system. NI power entry panels include EMI/line filters to protect sensitive measurement equipment.

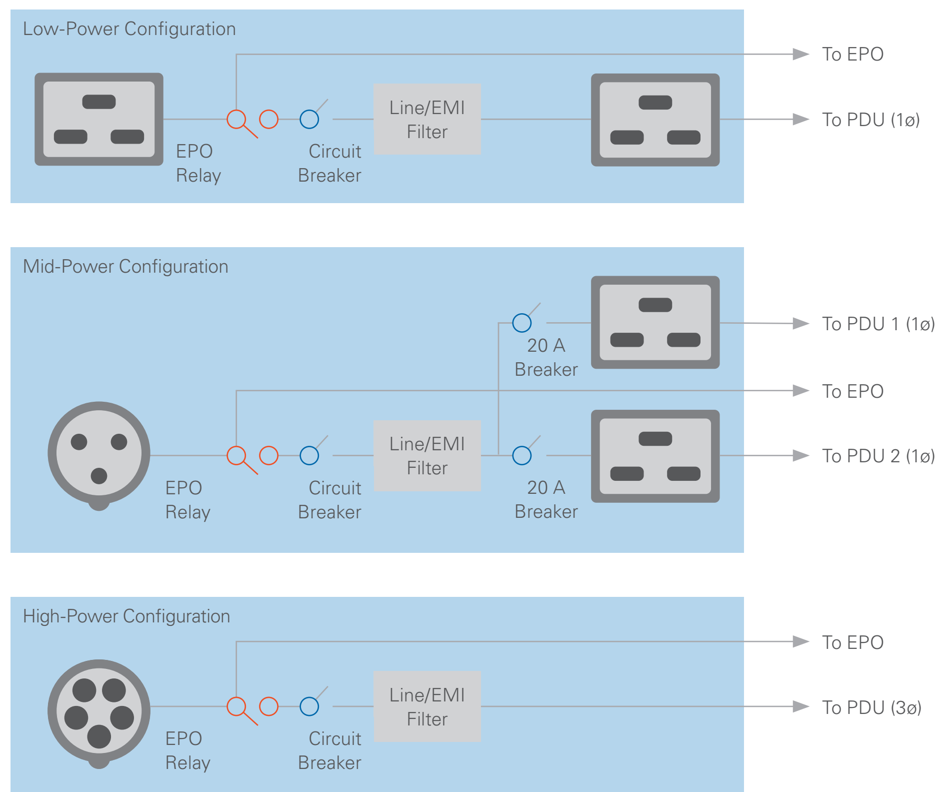


Figure 4. A circuit breaker and Line/EMI filter are critical to protecting the equipment in your test system and ensuring proper and accurate performance of your instrumentation. Example power entry panels are shown for low-power, mid-power, and high-power configurations.

Power Budget

A power budget is a critical part of planning resources and components for a test system. A given piece of equipment must have access to the proper amount of current at the correct voltage level. Budgeting must be performed for the entire system and at each point that power is distributed within the system. After determining the amount of power required, you can apply a few standard rules to the calculated values to right-size the power allocations in the test system.

System Power Budget

A system power budget begins with finding the maximum power requirements of all equipment included in the test system. The sum total should contain the expected power draw of all components in the test system, including voltage, current, and watts of power. In many cases the most important part of power budgeting is the current. Only a certain amount of current can flow through a given transmission line in the system, so current often has to be carefully distributed throughout the system using a power distribution unit (PDU).

The power draw of a given device is generally published in the user manual and sometimes includes a number of power requirements at different conditions. Occasionally, devices have specified typical power consumption and a maximum or worst-case power consumption specification. As a best practice, use the maximum power requirement as a conservative safety measure, and then subtract a given percentage, usually 30 to 40 percent, for a more realistic measure. Figure 5 shows the maximum power requirement of a stand-alone instrument that would be integrated into a test system.

Power Requirements

Caution: The protection providd by the VirtualBench hardware can be impaired if it is used in a manner not described in the *NI VB-8034 Safety, Environmental, and Regulatory Information* document.

Voltage input range	100 VAC to 240 VAC, 50/60 Hz
Power consumption	150 W maximum
Power input connector	IEC C13 power connector
Power disconnect	The AC power cable provides main power disconnect. Do not position the equipment so that it is difficult to disconnect the power cable. Depressing the front panel power button does not inhibit the internal power supply.

Figure 5. The VirtualBench all-in-one instrument specifies maximum power required at times of high-energy usage as opposed to typical or average power.

As a quick and simple example consider the test system in Table 1. First collect the maximum power consumption of each piece of equipment in the test system. Make sure to account for subsystems and bottlenecks. PDUs have a maximum current limit—in this case 16 A—so plan accordingly. The next step is to correct these values for typical power required as opposed to maximum. This means taking about 60 to 70 percent of that value. In this case, that gives you approximately 1,920 W for this test system using 70 percent as a conservative measure. It may also be a good idea to add about 20 percent of this full value as a means of expanding or adding new functionality to the system in the future without having to add more power to the system.

EQUIPMENT	MAXIMUM POWER CONSUMPTION	AVERAGE POWER UTILIZATION	CURRENT AT 110 V
PDU 1			
Fans	50 W	35 W	.03 A
HMI	100 W	70 W	.06 A
Ehternet Switch	25 W	17.5 W	.02 A
Overtemp Monitor	10 W	7 W	.01 A
PXI System	526.9 W	369 W	3.4 A
DUT Control Pumps	1,000 W	700 W	6.4 A
PDU 1 Total		**1,198.5 W**	**11.0 A**
PDU 2			
VirtualBench	150 W	105 W	1.0 A
750 W Power Supply	1,100 W	770 W	7.0 A
PDU 2 Total		**875 W**	**8.0 A**
System Total		**2,073.5 W**	**19.0 A**

Table 1. Start calculating a power budget by collecting the maximum power consumption of all system components, applying an average power utilization factor, and adding them together. Remember to account for bottlenecks and subsystems.

Three easy best practices can significantly simplify power budgeting:

- 1. Base your system power requirements on about 60 to 70 percent of the maximum required power of each component.
- 2. Add about 20 percent to the final power calculation from rule one as a safety buffer to account for high-activity periods and any necessary future expansion of the test system.
- 3. Remember that some items connect through PDUs and UPSs, so there are power subsystems within the larger system.

Subsystem Power Budget

A step not included in solving for the power budget above is how to account for subsystems within the large test rack. A subsystem can be any subset of the equipment in the larger test system that all share a common power source. This may be a number of instruments using a single bank of a PDU or a modular instrumentation system like PXI.

A benefit of modular instrumentation is that it can simplify power management. If all the instruments included in the PXI chassis were separate in the test system, you would have to account for each of them individually. PXI chassis provide high-quality and safe power to all instruments in the chassis and come in several power and instrumentation slot options. When adding a PXI system to your power budget, you can take one of two options:

- 1. Use the maximum power consumption of a full PXI system as specified by the PXI chassis. For example, the maximum power consumption of a PXIe-1085 PXI Chassis is 791 W, which would translate to a budgeted power consumption of 554 W after applying an average utilization factor of 70 percent.
- 2. Add the maximum power consumptions of all modules in the PXI system to get a very accurate power budget number. See Figure 6 for an example of performing a detailed PXI system power budget.

Additionally, a modular instrumentation system is significantly more efficient than a traditional set of instruments because it removes the shared components like monitors and cooling that would have to be powered within a test system.

As an example of an accurate power budget for a PXI system, consider a PXIe-1085 PXI Chassis with 24 GB/s system throughput that includes a PXIe-8880 PXI Controller, six PXIe-4139 precision system source measure units (SMUs), two PXIe-5162 PXI Oscilloscopes, a PXIe-6570 digital pattern instrument, two PXIe-4081 7 ½-digit digital multimeters (DMMs), and four PXIe-2527 multiplexer switch modules. See a representation of how the PXI system power budget is calculated in Figure 6.

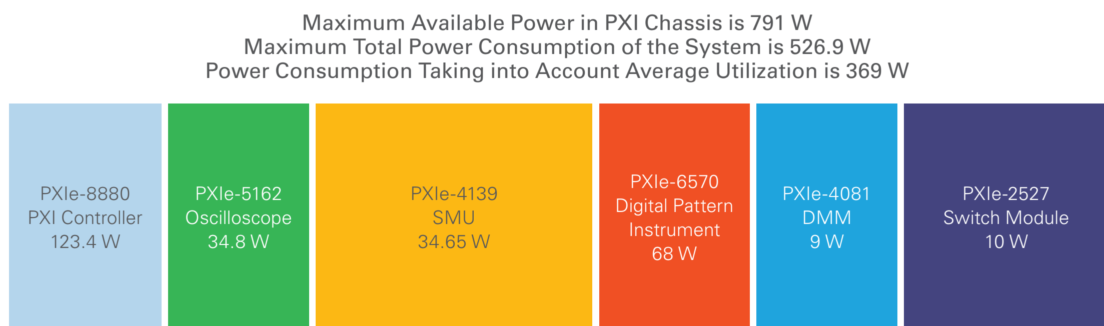


Figure 6. The total power consumption of a PXI chassis is the sum of all modules in the chassis. You can see above a full chassis of instrumentation that will, in the worst-case scenario, consume 526.9 W.

Power Distribution Unit

A PDU's main purpose is to take an input power signal and distribute it to a number of outputs that power components of the system. These internal power outlets from the PDU have a rated voltage and current and are often available for both alternating and direct current. The best PDU options have a number of features:

- Remote on/off gives operators the ability to make changes in the power state with the power mechanism and EPO. In this way, the operator has full control of the system state from an easily accessible location. Operators also can disable the power in the system from the local and global EPO mechanism.
- Built-in circuit protection like fuses can protect valuable and fragile equipment from unexpected power events, which could save tens, or even hundreds, of thousands of dollars.
- Bank sequencing can ensure that specific equipment powers on first before other banks power on. For instance, a PXI chassis that is connected to an external controller, or extended from another master PXI chassis, needs to start before the host controller. In this case, the PDU should enable a bank of outlets that include the slave PXI chassis before starting the bank that includes the master PXI chassis.
- Multiple banks that handle a given amount of power allow you to balance power loading on the PDU to prevent over-current conditions that could damage equipment in the test system. For instance, a PDU that has three banks of power outlets that can deliver 16 A on each bank prevents any one piece of equipment connected to the PDU from experiencing more than 16 A. It also means that you must be aware to distribute the current required for the equipment across multiple banks.
- DC supplies can provide power to items like status LEDs or cooling systems that run off of DC power with remote on/off and bank sequencing as well. Some of the items are even useful in the enable power state of a system, making a remote powering function necessary.

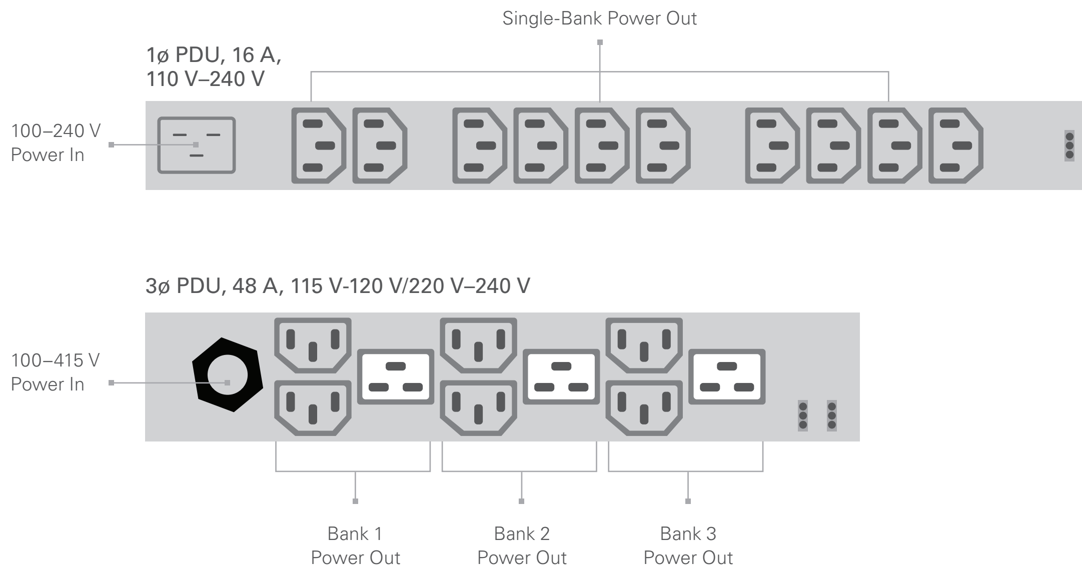


Figure 7. PDUs can have different connectivity and architectures. The PDU above has a single bank that can supply devices with up to 16 A, and the PDU below has three banks that can supply 16 A each for up to 48 A.

Powering Critical Components in the Test System

Make sure that critical components like the host controller and sensitive instrumentation in your test system get power from the UPS. Some components of the test system are more important than they might seem at first glance. Without the cooling systems continuing to run after a power incident, the host controller may overheat. Without power to the touch panel monitor on the test system, a technician has no way of troubleshooting the failures or logging data of the power incident. Think about the items that you want to be operational, even after a power outage or emergency.

Powering System Overhead and Support

Remember overhead and infrastructure like temperature control, network connectivity, and user interface elements of the test system when allocating power. Having an outage in your production because of overheating or lack of network connectivity is just as detrimental as failures in test instrumentation.

Uninterruptable Power Supply

A good test system designer takes into account the quality of the grid and designs the system to avoid undefined behavior during power loss and brownouts. You can use UPSs to power critical components in the test system during these events and sometimes during normal operation as well.

A UPS can deliver power with a dependable voltage and current supply. It can also act as a battery power supply after a power outage or significant brownout. A UPS is a critical part of a rugged test system, especially one in a location with an unreliable power grid.

There are two major types of UPSs:

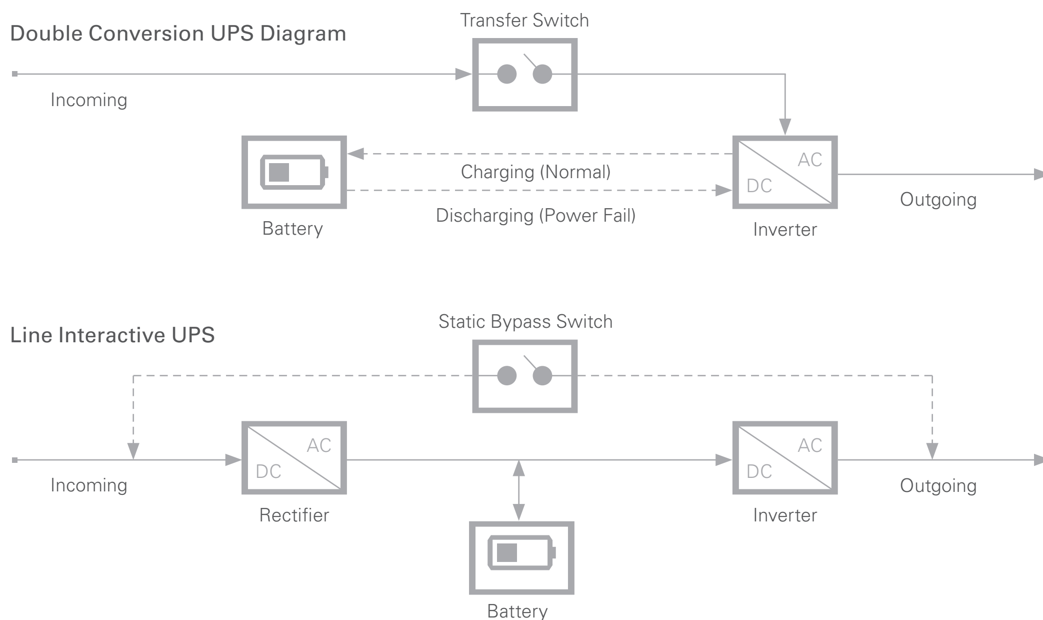


Figure 8. A UPS is used to provide clean, reliable power that also allows for graceful shutdown in the case of a blackout or brownout. A double conversion UPS is always charging a battery that provides consistent power to the system.

- **Line interactive UPS**—In a line interactive UPS, the active power line input is connected directly to the power output. The UPS then monitors incoming power to ensure that the power does not sag below a given threshold. If the power line does sag too low, it switches to a battery that is charged by running the UPS in reverse to power the output signal. In this case, the test system is receiving line power during operation without any conditioning and the UPS takes over power delivery in the case of a power failure.
- **Double conversion UPS**—Double conversion UPSs connect the incoming power line to a battery that charges continuously and then supplies power to the output line of the UPS. The power supply of a double conversion UPS is very consistent because it is delivering power from the onboard battery. A double conversion UPS has the added benefit of always being prepared to act as a backup power supply with a fully charged battery to allow critical systems to shut down gracefully if a blackout or significant brownout occurs without having to switch power supplies. Although these UPSs are slightly less efficient, they provide an added value of always providing stable and accurate power inside the test system, which makes double conversion UPSs good choices for ATE applications.

Power Quality and Reliability

No power grid is perfect, yet most electrical devices are designed to operate under ideal power conditions. When the power from the grid varies from the power the system is designed to use, the system behaves in an unexpected manner. Instruments can take bad measurements or source incorrect signals. Devices and systems can switch on and off and lose important settings or default to incorrect settings. This unexpected behavior can lead to bad test results, damaged devices under test (DUTs), or worse. A double conversion UPS has the added benefit of constantly providing filtering by charging the internal battery with incoming power and providing a highly reliable, clean power source.

Blackouts and Brownouts

Blackouts occur when the power that the grid supplies completely turns off. Blackouts are fairly rare where there are well-developed power grids, and managing behavior of a system during these conditions can go two ways: (1) run some or all parts of the system off of a battery for a short time so that it can shut down properly or (2) let it turn off because of the lack of power.

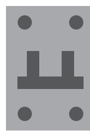
Brownouts and power surges are far more common in the grid, especially in facilities like factories with large power consumptions, and are more difficult to handle because they can cause indeterminate behavior in the system. A brownout can be any sag or glitch in voltage or current in the grid that causes less power to be delivered to the test system. A surge is a momentary instance of additional voltage difference or current than the grid normally provides.

UPSs have an internal battery that allows for time between a blackout, or severe brownout, and a new power source, like a generator, coming online to provide enough power for essential equipment in the test system. Essential equipment includes the host controller and user interface and any other critical equipment. The time that the battery provides allows the system to maintain essential data and avoid corruptions or unsafe software states.

Power States

It is often necessary for a test system to have multiple running statuses to allow for debugging maintenance, power saving, and safety. A good approach to test system design is to implement four states of operation:

- **Off**—A system is entirely disabled with no power passing through the line filter or any internal test system components.
- **Enabled**—Power is passing through the line filter and into any directly powered equipment. Usually, all equipment is powered through a PDU. In the enabled state, only primary or master outlets on the PDUs would likely be activated. In some cases, DC supplies on the PDUs are also activated to power system support and other components. For example, in the enabled state, an Ethernet router and real-time system controller could power on so that technicians can monitor the health of the test system.
- **On**—A change to this state begins the main power on sequence of the test system. All PDUs receive power and enable outlets to other system equipment. In many cases, it is helpful or necessary to stage the power sequence when certain system components depend on others to be running when they start. Read more about PDUs in the Power Layout section.
- **Emergency Power Off (EPO)**—The EPO immediately cuts power to the test system when a user or system monitor recognizes an unacceptable operating condition.



- System Power Disabled
- No Facility Power to the Line Filter

- System Power Enabled
- PDU Master AC Outlets Enabled
- PDU DC Outlets Enabled (Individually Controllable)
- PDU Standard Outlets Disabled
- UPS Disabled

- System Power Enabled
- PDU Master Outlets Enabled (DC Supplies—Fans, System Controller, ENET Router)
  – Individual DC Outlets Enabled (2 Bank Power-Up Sequence)
- PDU Standard Outlets Enabled (2 Bank Power-Up Sequence)
- UPS Enabled

- System Power Disabled
- No Facility Power to the Line Filter

Figure 9. A test system requires multiple power states including off, enabled, on, and EPO to ensure efficient operation.

Although the grounding path described is usually sufficient, grounding each piece of equipment in the test system individually can guarantee safety. A power entry panel from NI has a star grounding block, as seen in Figure 2, which is connected to other ground blocks throughout the rack. The grounding stud on the outside of the power entry panel can then be attached to a true earth ground outside the chassis. Additionally, each piece of equipment will generally have a grounding stud that can be directly connected to a ground. You can see the grounding screw of an NI PXI Chassis in Figure 12. Attaching each piece of equipment to the distributed grounding blocks throughout the chassis ensures that each one is grounded safely and that all ground leads are very short, which leads to less electromagnetic noise.

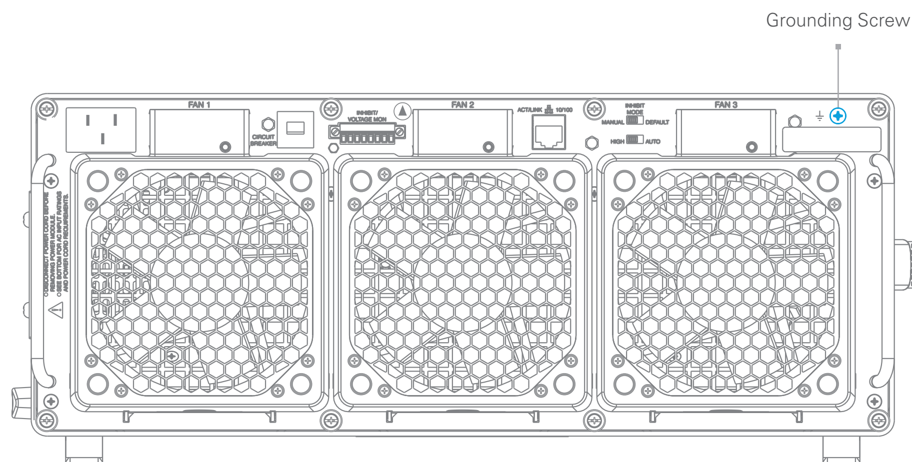


Figure 12. The PXIe-1085 PXI Chassis has a ground screw that allows you to directly ground the chassis and all instruments to an external grounding block. Grounding each piece of equipment in a rack is a best practice for guaranteeing safety.

Make sure that electrical connections to ground plains are short. Long ground loops can cause standing waves that result in radio frequency emissions within the system. If long transmission lines are needed to connect to ground plains, couple the signal with the ground signal in a twisted pair configuration to cut down on electromagnetic noise. Include both the positive and negative references of the signal if it is floating, or not referenced to ground.

Emergency Power Off

When a test system encounters a serious issue or an emergency is taking place in the facility, operators need the ability to quickly and cleanly power off the test system. EPO mechanisms are included on test systems to simplify connectivity and inhibit power switching. Operators might use an EPO to reset a system in an error state, prevent damage to a DUT, or even prevent harm to themselves. EPO functionality is also required by the safety standards bodies such as IEC and UL.

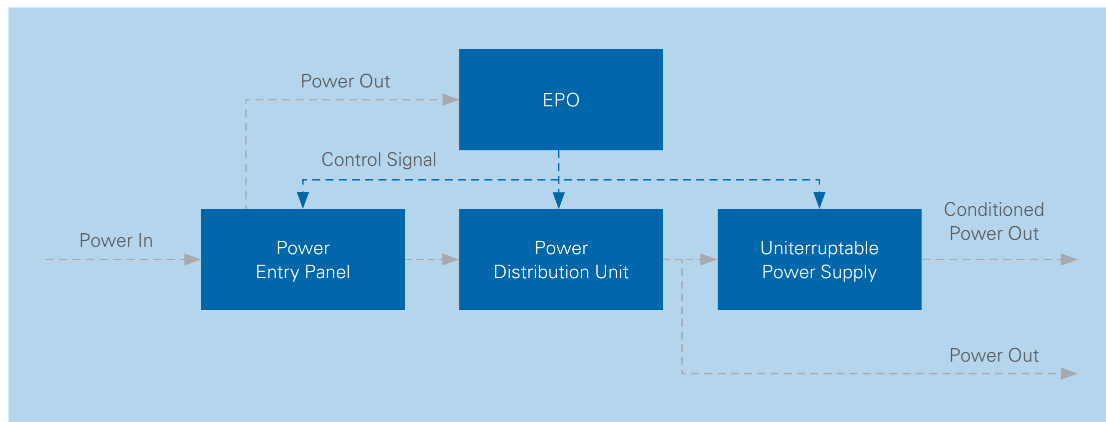


Figure 10. The EPO is connected to all equipment in the test system and can disable all connected equipment when necessary to maintain safety.

An EPO is generally a physical mechanism like a button or switch that is easily accessible to an operator and, when pressed, cuts power to all test system equipment. Ideally, the EPO panel has connectivity with all equipment in the test system to ensure that everything is powered off quickly. Most EPOs put systems into an off state that requires them to be reset to the enabled state before they can be reactivated and all equipment can power on. This prevents systems from unexpectedly restarting after a power off when there is an unsafe condition.

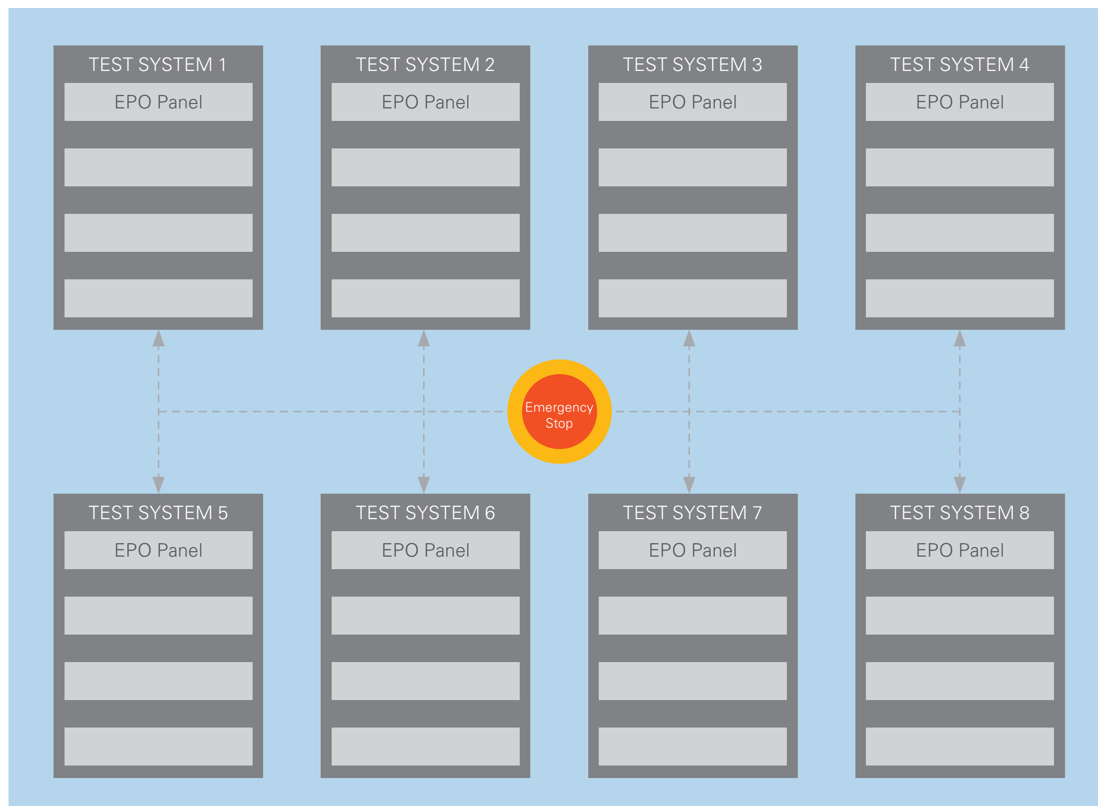


Figure 11. In some cases, a global EPO is necessary to disable all test systems and equipment in a facility. A global EPO is a single power off mechanism that enables the local EPOs of all individual systems.

Grounding

Grounding is a critical part of test system design for two main reasons: safety and measurement quality.

Ensuring that your test system has proper grounding to guarantee safety means giving all equipment in the test system a proper path for current to flow to a true or earth ground. The power entry panel must be connected to a power source that has a proper ground. You should then be able to choose any piece of equipment in the test system that is an end power consumer and follow its path to ground back to the power entry panel. Follow the ground current path for the Ethernet switch in the power layout of the example test system in Figure 1. The Ethernet switch ground is connected to the UPS ground, which should be connected to the ground of the PDU, which should be connected to the ground of the power entry panel. By creating a path for current that forms as a product of ground loops to flow to ground, you avoid building up dangerous charge in the system that could arc and cause damage or discharge into an operator or DUT.

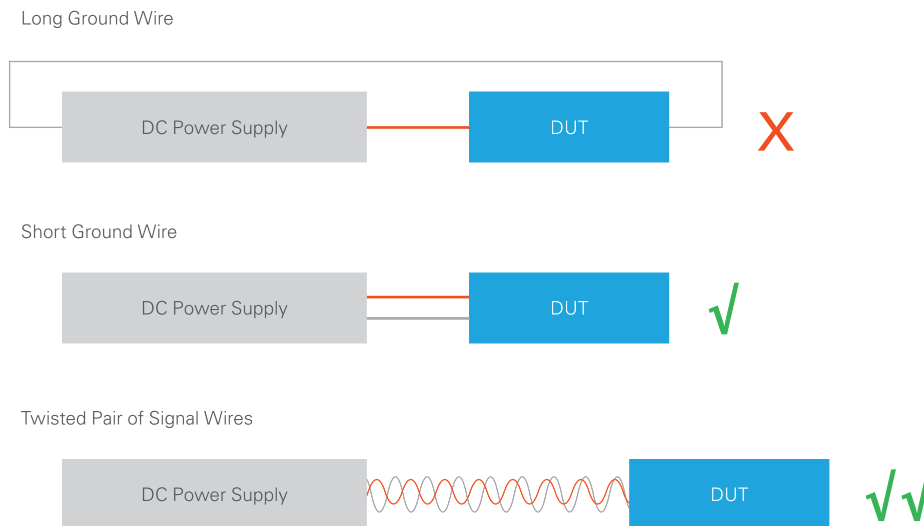


Figure 13. Having long, unmatched ground wires in your system can cause significant ground loops and act as an antenna for noise signals. Using short ground lines is better, but still has the possibility of picking up unwanted noise. For the best performance, use twisted pairs of signal and ground wires in your system.

Learn all you need to know about making the right connections for your measurements by reading the white paper Comprehensive Guide for Field Wiring and Noise Considerations.

Best Practices for Components

There are nearly infinite ways to source materials and construct a test system. When building a system that you have to maintain over time, consider long-term support and the extensibility of the system to add future requirements. To achieve these results, it is best to source system components from commercial vendors that support products and consumers with long-term supply strategies. It can seem like commonsense to work with a vendor for items like PDUs, UPSs, system controllers, and instrumentation, but that same strategy can pay off long term on smaller items like interconnects and cables. A committed vendor that can supply connectors along with vendors supplying test instrumentation can keep your system running with reasonable effort for a decade.

On the rare occasion that special requirements or extenuating circumstances make using a commercial product impossible, many companies are experts in custom equipment and solutions for test systems. Keep in mind that these solutions are often for a single consumer and are more likely to change or become obsolete over time.

©2016 National Instruments. All rights reserved. National Instruments, NI, ni.com, and VirtualBench are trademarks of National Instruments. Other product and company names listed are trademarks or trade names of their respective companies.

NATIONAL INSTRUMENTS

FUNDAMENTALS OF BUILDING A TEST SYSTEM

Switching and Multiplexing

CONTENTS

Introduction

Switching Architectures

How to Select the Best Switch for Your Application

Next Steps

Introduction

Many automated test applications require routing signals to a variety of instruments and devices under test (DUTs). Often the best way to address these applications is to implement a network of switches that facilitate this signal routing between the instrumentation and the DUTs. Switching not only handles this signal routing, but it is also a low-cost way to increase the channel count of expensive instrumentation while increasing the flexibility and repeatability of your measurements.

When adding switching to an automated test system, you have three main options: design and build a custom switching network in-house, use a stand-alone box controlled via GPIB or Ethernet, or use a modular platform with one or more instruments such as a digital multimeter (DMM). Switching is almost exclusively used alongside other instruments, so tight integration with those instruments is often a necessity. An off-the-shelf, modular approach can meet these integration challenges inherent in most common test systems. This guide will outline best practices for integrated switching and multiplexing into your test system.

Switching Architectures

Switching can be a cost-effective and efficient option for expanding the channel count of your instrumentation, but it is not always the best option. There are four main types of switching architectures:

1. No Switching
2. Switching in Test Rack Only
3. Switching in Test Fixture Only
4. Switching in Test Rack and Test Fixture

The following table outlines the strengths and weaknesses of all four switching architectures.

Below ○ Average ◒ Above ●

	Flexibility	Throughput	Cost	Low-Level Measurements (mV, μA, mΩ)
No Switching	○	●	○	●
Switching in Test Rack	●	◒	●	○
Switching in Test Fixture	○	◒	◒	◒
Switching in Test Rack and Fixture	●	◒	◒	◒

Table 1. Pros and Cons of Various Switching Architectures

No Switching

In the first architecture, no switches are used to route signals between the devices under test (DUTs) and the instruments in the test system. Such systems typically have a single instrument channel dedicated to every test point.

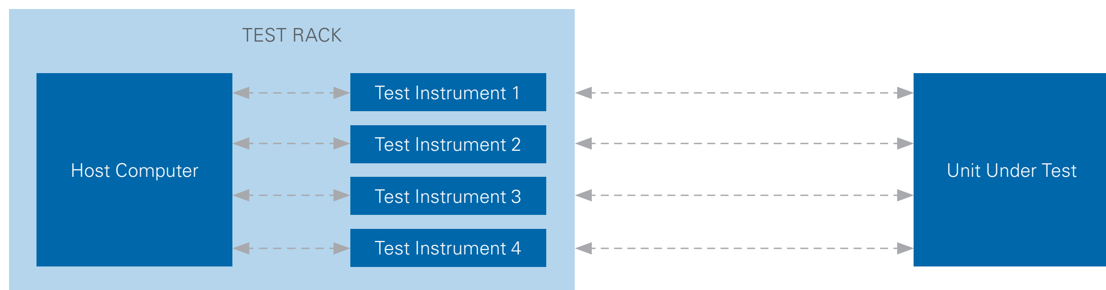


Figure 1. A test system without switching has a direct connection from each instrument to the unit under test.

Switching in Test Rack Only

The second switching architecture uses only commercial off-the-shelf (COTS) switches to route signals between measurement instruments and DUTs. Switching in the test rack provides a way to use existing switch products and offers the easiest path to expansion. It is important to choose a COTS switching platform that can offer a broad range of functionality and easy expansion options. Not doing so can result in higher expenditures because of test system redesign over the life of a tester.

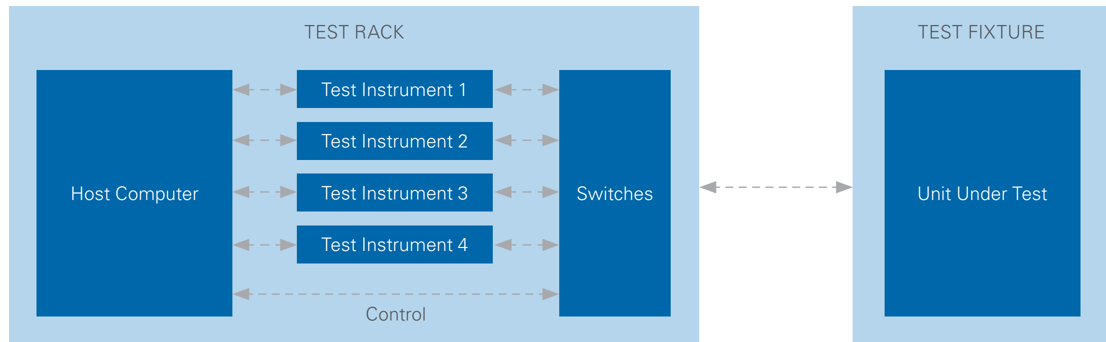


Figure 3. Some test systems integrate switching within the test rack for ease of expansion.

The PXI platform, for instance, offers more than 600 different types of switch modules that can route signals as high as 600 V, 40 A, and 40 GHz. NI alone makes 100 different PXI switch modules that you can configure in more than 200 different switch topologies.

Advantages of Switching in Your Test Rack

By using a COTS switching solution, you can save considerably on development time, including printed circuit board (PCB) design and driver development. Additionally, COTS switching improves test system scalability because you can now add more switching by purchasing additional modules from a switch vendor rather than redesign your entire test fixture.

Furthermore, each switch vendor provides solutions with their own unique advantages. NI switches, for example, have an onboard EEPROM that keeps track of the number of instances each relay on the module is activated and other features to monitor relay health, such as functional and resistive relay tests. With these features, you can predict when a specific relay will reach the end of its mechanical lifetime, and thus conduct predictive maintenance. These features are especially useful when maintaining high-channel-count switch systems that can be extremely difficult to debug manually, or on a manufacturing production floor where unexpected downtime can cause significant and costly delays.

You can also use NI switch modules to increase the throughput of your test application by downloading a list of switch connections to memory onboard the switch modules and cycling through the list using bidirectional triggering between the switch and any instrumentation, without interruption from the host processor.

Disadvantages of Switching in Your Test Rack

Using switches can often slow down your test process because it requires you to take measurements from your test points on any given DUT sequentially rather than in parallel, as discussed previously. Placing all of the switching within the test rack also increases the total amount of cabling. In addition to using cables between your switches and measurement instruments, you also need cables between DUTs and the switch. This can cause error in sensitive measurements, such as leakage current or low-resistance measurements.

Switching in Test Fixture Only

The third switching architecture uses switches in the test fixture only. In this case, signals from the measurement instrument are switched to various test points on the DUT using individual relays placed on a PCB near the fixture or in the fixture itself. If you are using this architecture, you need a relay driver in your test station to control the individual relays from your test program. A good example of a COTS relay driver is the PXI-2567, which is a 64-channel relay driver module that allows you to use the NI-SWITCH driver to control the external relays using a standard API, removing the need for custom programming. Alternatively, you can design an external circuit to drive your relays, but this requires additional design work.

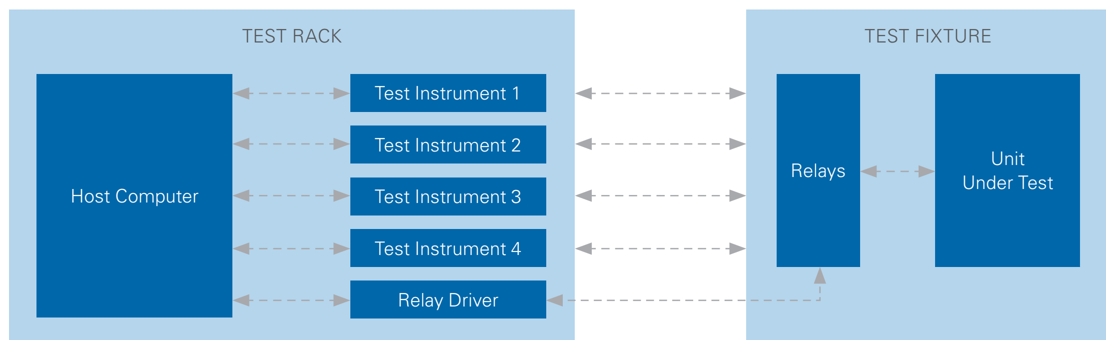


Figure 4. Test systems with switches in the test fixture require a relay driver.

Advantages of Switching in Your Test Fixture

As mentioned before, switching helps reduce test cost, regardless of location. Additionally, building switches into your test fixture eliminates the need for cables between your DUTs and the switches themselves. Reduction in cabling also helps decrease measurement error.

Disadvantages of Switching in Your Test Fixture

As discussed previously, using switches can often slow down your test process. Additionally, building custom switching into the test fixture requires PCB design experience, so this may not be an option for everyone. Switching within your test fixture also poses problems with the ability to scale your test system to accommodate more test points.

Advantages of Not Switching

Cables and switches can often degrade the integrity of your signal. By not using switches, you can provide your signal with a more direct path to the measurement instrument and thus improve your measurement accuracy. In addition to improved measurement accuracy, you can achieve faster test speeds. By having a dedicated instrument for each test point, you can make parallel measurements rather than sequential measurements, and thus increase test throughput.

Disadvantages of Not Switching

Having a dedicated instrument for each test point can prove to be extremely costly. Another disadvantage is expandability. You can easily run out of space in your test rack if you build a test system without switching. This can cause you to completely redesign your test system, which can result in additional costs for hardware changes, software updates, and revalidation. For instance, suppose you have a test system that tests 20 resistance temperature detector (RTD) sensors in parallel using 20 PXIe-4081 digital multimeters (DMMs). Now assume your system needs to expand to test 40 RTD sensors. To do so, you need to add 20 more DMMs, which require 20 more PXI slots. Alternatively, you can use a single PXIe-4081 7½-digit DMM along with a switch to test all 40 RTD sensors sequentially, which requires as little as two PXI slots.

When to Build a Test System Without Switching

Building a test system without any switching is usually recommended if you are making either extremely sensitive measurements that would get distorted with the addition of cables and switches or if you need to keep test time to a minimum. For instance, some semiconductor test applications have a single parametric measurement unit or source measure unit dedicated to every pin on a chip, because semiconductors are a high-volume business and test costs often make up a significant portion of the total manufacturing cost of a chip. You can significantly reduce test costs by minimizing test time through parallel measurements with dedicated instrument channels. Additionally, in the semiconductor industry, testers are often built for specific chipsets or chipset families, so they are not usually expanded over their lifetimes.

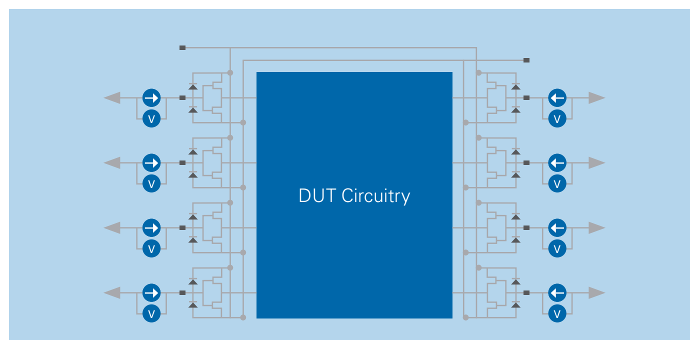


Figure 2. Some semiconductor applications use dedicated instruments to test each pin on a given chip in parallel.

Another disadvantage of this option is the cost associated with designing a custom board for specific safety and compliance standards. If you are testing a high-voltage device, you likely need to build a switching fixture that adheres to various regulations such as UL, CE, and VDE. It can be challenging to design a PCB filled with relays that complies with the creepage and clearance requirements of those various standards. In such cases, using COTS switches can help reduce costs. Many COTS vendors certify their modules to comply with a wide range of safety standards. For instance, all NI switch modules that have a voltage rating greater than 60 V_{DC} or 30 V_{AC} and 42.2 V_{pk} are considered high-voltage devices and therefore built to adhere to the following safety standards.

Figure 5. NI switch modules meet a wide range of safety and compliance standards.

Switching in Test Fixture and Test Rack

The last switching architecture includes switches in the test station as well as the test fixture. Using this paradigm, you can take advantage of the benefits of both COTS switching solutions and simultaneously minimize errors in specific, sensitive measurements by placing switches closer to the DUTs and in the test fixture. By using the PXI-2567 relay driver along with other PXI-based switches, you can program your whole switch system, including both test rack COTS switches and custom relays in the test fixture, using a standard, well-supported driver API.

Advantages of Switching in Your Test Fixture and Test Rack

By placing COTS switches in your test rack and relays in your test fixture, you can build a switching system that scales easily and adds minimum errors into your critical or low-level measurements. Using this architecture, you can place those switches being used to route sensitive signals in the test fixture and all of the remaining switches in the test rack. In addition to scalability, using COTS switches helps you take advantage of vendor-specific features, such as the relay count tracking and hardware triggering features in NI PXI switch modules.

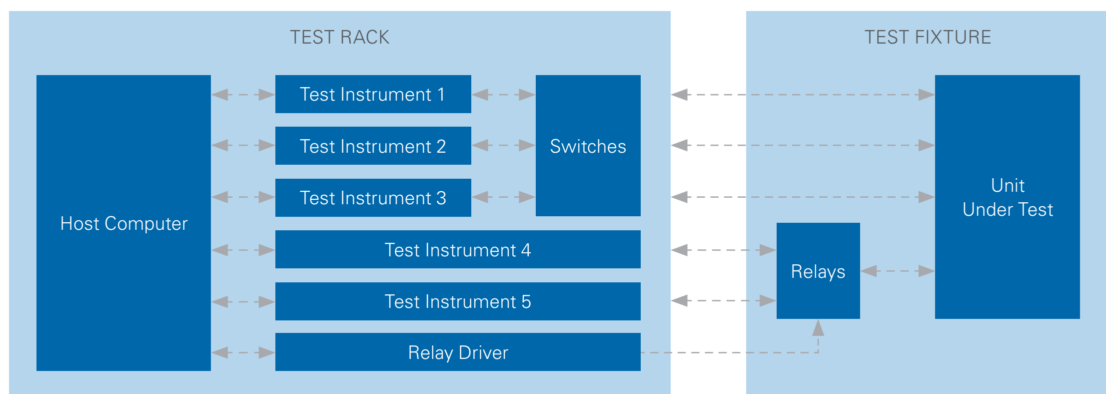


Figure 6. Test systems with switches in the test rack and test fixture provide great flexibility, but require additional design work.

Disadvantages of Switching in Your Test Fixture and Test Rack
Using switches can often slow down your test process because it requires you to take measurements from your test points sequentially rather than in parallel. Building custom switching into your test fixture can also be time-consuming and require a considerable amount of PCB design expertise, especially for high-voltage or high-frequency signals.

How to Select the Best Switch for Your Application

In addition to switch location, you want to compare the various switch topologies and relay types to ensure your switching subsystem meets your signal requirements and test goals. For automated test applications, the term switch is often used to describe a COTS device that uses relays to switch signals between multiple DUTs and instruments. Switches organize relays in various ways to create different switch topologies, such as general-purpose relays, multiplexers, and matrices. Different relay types have various trade-offs, including size, signal rating, and life expectancy. This section describes common switching topologies, popular relay types, key switching specifications, and general tips and tricks for switching in an automated test system.

Common Switch Topologies

After you have decided that switching is ideal for your application, the next step is to select the best switching topology, or way of organizing the relays to build a larger switch network. Most switch vendors categorize their switches into three main categories: general-purpose relays, multiplexers, and matrices. Some switches, such as PXIe-2524, are capable of multiple topologies, which gives you the ability to configure the topology in software. You can choose among five different topologies to meet changing requirements. When considering topologies, it is important to think about the total number of connections required, the maximum number of simultaneous connections, and the need to scale for future changes to the test system.

General-Purpose Relays
A general-purpose switch consists of multiple independent relays meant to be used independent of each other. A general-purpose relay is a great option when you simply want to make/break a connection within a circuit or switch between two possible inputs or outputs. Individual relays are often classified by their number of poles and number of throws. The pole of a relay is the terminal common to every path, and each position that a pole can connect to is called a throw.

A single-pole single-throw (SPST) relay is similar to a standard light switch with on and off states. An SPST relay comes in two forms: Form A and Form B. Form A SPST relays are normally open until the relay is activated, which causes the relay contacts to touch, completing the circuit. Alternatively, a Form B SPST is normally closed until the relay is activated, which causes the relay contacts to break their connections, opening the circuit.

Figure 7. SPST relays come in two forms: normally open (Form A) and normally closed (Form B).

A single-pole double-throw (SPDT) relay has a single pole, or common connection, that can alternate between one normally open contact and one normally closed contact. Every SPDT is categorized as either a Form C or Form D relay. When a Form C SPDT activates, the normally closed signal path is opened before the relay connects to the normally open contact. This SPDT relay operation is described as "break before make," or BBM. Alternatively, actuating a Form D relay connects the normally open signal path before the normally closed signal path is opened. This SPDT relay operation is called "make before break," or MBB.

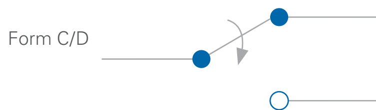


Figure 8. SPDT relays share one common pole between two possible throws, or connections.

TASK	OPEN	DURING OPERATION	OPERATION COMPLETE
Form C	N.C. COM N.O.	N.C. COM N.O.	N.C. COM N.O.
Form D	N.C. COM N.O.	N.C. COM N.O.	N.C. COM N.O.

Figure 9. SPDT relays also come in two forms: Form C and Form D.

A double-pole single-throw (DPST) relay is when two Form A SPST relays are actuated simultaneously, usually with the same coil and packaged together. A DPST is ideal when two signal paths need to be opened or closed simultaneously. You can build a DPST from two independently controlled Form A SPST relays, but there might be some time difference between actuating the two relays.

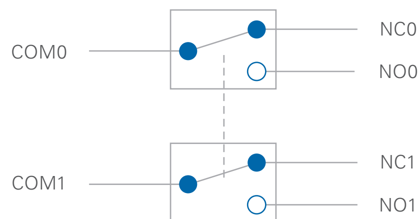


Figure 10. DPST relays offer simultaneous control of two Form A SPST relays.

Multiplexers
A multiplexer, or mux, is a way of organizing relays that gives you the ability to connect one input to multiple outputs, or one output to multiple inputs. Multiplexers provide an efficient way to connect multiple DUTs to a single instrument. However, this switching architecture requires more upfront knowledge of which DUT connections need access to your various instruments.

Multiplexers are sometimes built using multiple Form A SPST relays with the ends connected together. This method of building a multiplexer is simple and efficient, but its drawback is that the unused signal paths can cause AC signal reflections that degrade the bandwidth rating of the switch.

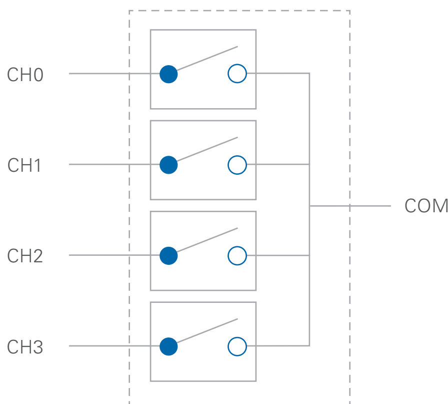


Figure 11. 4 x 1 Multiplexer Built from Multiple Form A SPST Relays Tied Together

Alternatively, multiplexers are sometimes built using cascaded levels of Form C SPDT relays to ensure the signal integrity of AC signals. This type of multiplexer often requires more PCB space, but it reduces any stubs or extra unterminated signal paths that might degrade the bandwidth of the switch.

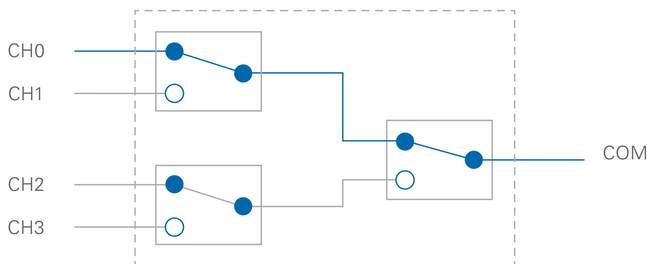


Figure 12. 4 x 1 Multiplexer Built From Cascaded Levels of Form C SPDT Relays

Matrices
A matrix is the most flexible switching configuration. Unlike a multiplexer, a matrix can connect multiple signal paths at the same time. A matrix has columns and rows with a relay at each intersection, which gives you the ability to connect column-to-column, column-to-row, and row-to-row signal paths. With the flexibility of matrices, you can connect all switch channels to each other through various signal paths that do not need to be predetermined. It is recommended that you plan your switch routes during the hardware planning phase, but matrices give you the flexibility to make changes to the switch routes as test requirements change.

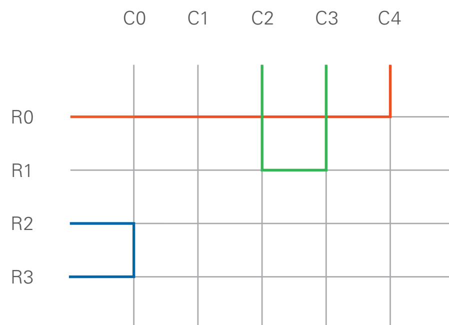


Figure 13. Matrices allow for maximum flexibility when routing signals.

Matrix size is often described as M rows by N columns (M x N) configurations. Some common configurations are 4 x 64, 8 x 32, and 16 x 16. However, in most cases, there is nothing special about rows or columns. A switch matrix can be transposed if it is easier for you to think in terms of more rows than columns, such as a 64 x 4 matrix instead of a 4 x 64 matrix.

Other Topologies
General-purpose, matrix, and multiplexer switches make up the vast majority of switches, but there are other specialized switching topologies such as a sparse matrix or a fault insertion unit (FIU).

A sparse matrix is a hybrid combination, somewhere between a matrix and a multiplexer, generally used for RF applications. By connecting the COMs of two multiplexers, you can create a pseudo-matrix with numerous rows and columns, but only one possible signal path can be connected at any given time. Multiplexers typically offer more channel density than a matrix does, because a matrix requires at least one relay per row-column intersection. Therefore, a sparse matrix typically offers more channel density in a given space, but is limited by a single signal path between the rows and columns. Sparse matrices are also useful for AC applications where signal bandwidth might be compromised by the stubs created by the unterminated rows and columns of a traditional matrix.

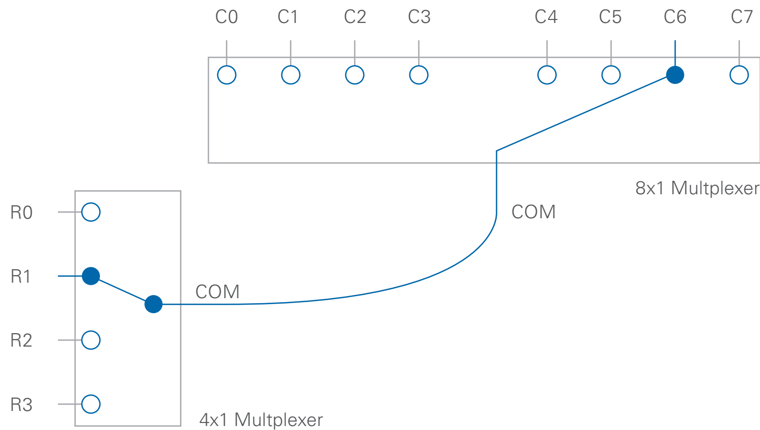


Figure 14. A sparse matrix is created by connecting the COMs of two or more multiplexers, and is commonly used for switching RF signals.

Another specialized switching architecture is the FIU, which is commonly used in hardware-in-the-loop (HIL) test systems. Hardware fault insertion, also known as fault injection, is a critical consideration in test systems that are responsible for the reliability of embedded control units, where it is imperative to have both a known and acceptable response to fault conditions. To accomplish this, FIUs are inserted between the I/O interfaces of a test system and the ECU so the test system can switch between normal operation and fault conditions, such as a short to power, short to ground, pin-to-pin shorts, or open circuit. For more information on FIUs, read the Using Fault Insertion Units (FIUs) for Electronic Testing white paper.

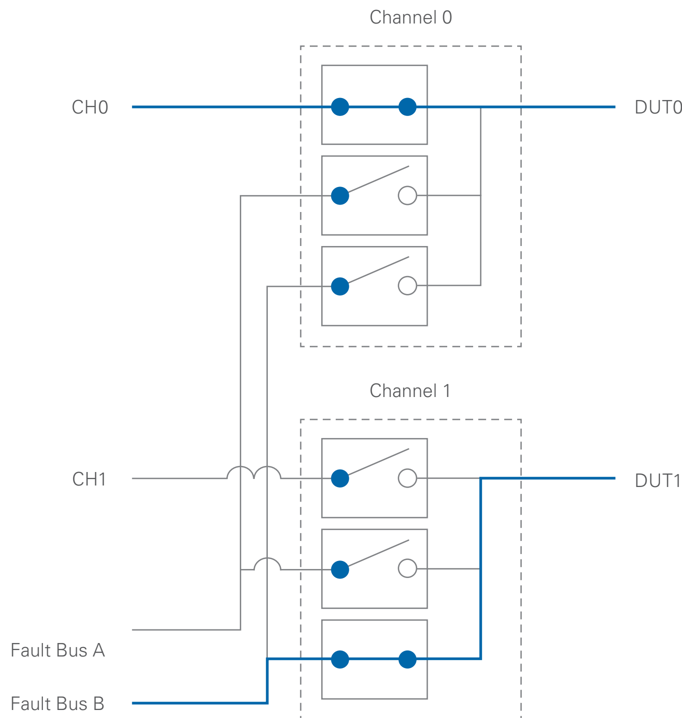


Figure 15. FIUs allow for automated fault condition testing, commonly used to test the reliability of embdded systems, such as automotive ECUs.

Relay Types

A relay is a remotely controlled device that makes or breaks a connection in an electric circuit. There are various types of relays, but four of the more popular relay types are electromechanical relays, reed relays, solid state relays, and field effect transistor (FET) relays. Each relay type has trade-offs that can impact the performance, cost, life expectancy, and density of a switch system, which is why it is important to select the best relay type to fit the needs of your application.

Note that the specs for an individual relay and a finished switch product differ in most situations. Relay specs, such as bandwidth, power rating, and contact resistance, refer only to the individual relay and do not include the PCB routes that connect the relays into a switch topology or the connector that provides the user with an interface to the switch topology. For example, a single relay may be rated for 0.05 Ω contact resistance and 300 V, but the finished switch product may have a larger path resistance (for example, 1 Ω), including multiple relays and PCB traces, and may not have the PCB creepage and clearance necessary to safely spec the switch product at 300 V.

Electromechanical Relays

An electromechanical relay (EMR), or armature relay, uses current flowing through an inductor coil to induce a magnetic field that moves the armature to the open or closed position, which completes the circuit by causing two contacts to touch. There are various types of EMRs, such as latching and nonlatching, that have small differences in operation. A nonlatching EMR uses a single coil and returns to its default position after the current stops flowing. Alternatively, a latching EMR remains in the position that it was switched to, even when the current stops flowing. Some latching EMRs use one coil and reverse the flow of current to reverse the direction of the magnetic field to push or pull the armature into the desired position. Other latching EMRs use a coil on either side of the armature to push the armature open or closed.

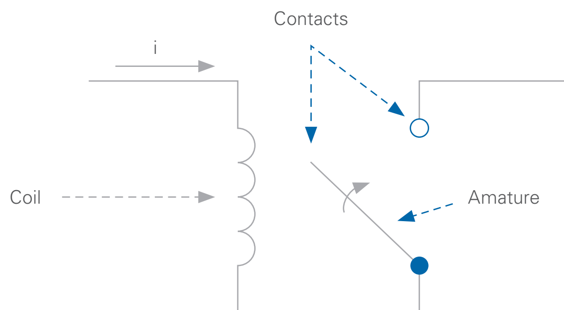


Figure 16. A single-coil electromechanical relay uses a magnetic field to open and close a mechanical switch.

EMRs support a wide range of signal characteristics, from low voltage/current to high voltage/current and DC to GHz frequencies. Also, EMRs have low contact resistance, typically much less than 1 Ω, and can handle unexpected surge currents and high power, up to 300 W. For these reasons, you can almost always find an EMR that fits the signal characteristics a test system requires. However, EMRs take up a lot of PCB space, are slow compared to other options (150 cycles/s), and have shorter life cycles because of their moving parts (up to 10ˆ6 cycles).

Because of these trade-offs, EMRs are a great choice when you need a durable relay rated for high power, high current, or high bandwidth, but you are not as concerned with relay speed and are willing to replace the relay as it degrades over time.

Reed Relays

Reed relays also use current flowing through an inductor to create a magnetic field used to connect physical contacts. However, reed relays can have much smaller and lighter contacts than EMRs. Reed relays use a coil wrapped around two overlapping ferromagnetic blades (called reeds) hermetically sealed within a glass or ceramic capsule filled with an inert gas. When the coil is energized, the two reeds are drawn together causing their contacts to complete a signal path through the relay. The spring force of the reeds causes the contacts to separate when the current ceases to flow through the coil.

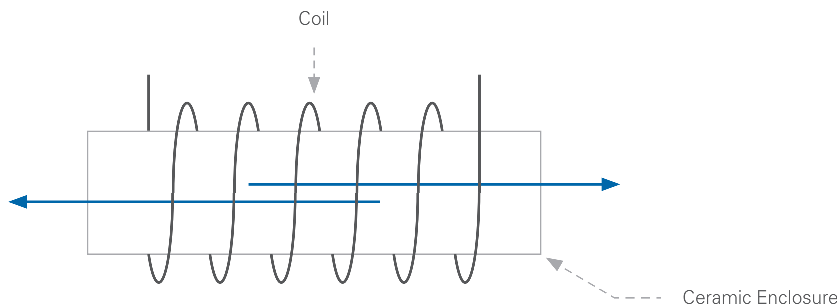


Figure 17. The spring force of the reeds causes the contacts to separate when the current ceases to flow through the coil.

Because reed relays can be smaller, you can fit more within a smaller footprint and they can switch faster than EMRs, up to 2,000 cycles/s. Also, their limited moving mechanical parts and isolated environment provide longer mechanical lifetimes, up to 10^9 cycles.

However, because of their smaller contact size, reed relays cannot handle as much power and are more susceptible to damage from self-heating or arcing, which can melt small sections of the reeds. If the two reeds are still connected when the molten section solidifies, the contacts may weld together. In this situation, the relay remains shut, or breaks one of the reeds if the spring force is enough to pull the two reeds apart. To protect against damage, monitor signals for large inrush currents that might be caused by hot-switching a capacitive load and use inline protection resistors to reduce the level and duration of the current spike. For more information on protecting reed relays, read the Reed Relay Protection white paper. The small size and high speed of reed relays make them a great choice for many applications. Reed relays are more often found on matrix and multiplexer modules rather than general-purpose switch modules. A good place to start is the PXI-2530B, which is a COTS switch that you can configure as 13 unique matrix or multiplexer topologies by swapping various front-mount terminal blocks.

Solid State Relays

Solid state relays (SSRs) are electronic relays that consist of a sensor that responds to an input, a solid-state electronic switching device that switches power to the load circuitry, and a coupling mechanism to enable the control signal to activate without mechanical parts. They are often constructed using a photosensitive metal-oxide semiconductor, field-effect transistor (MOSFET) device with an LED to actuate the device.

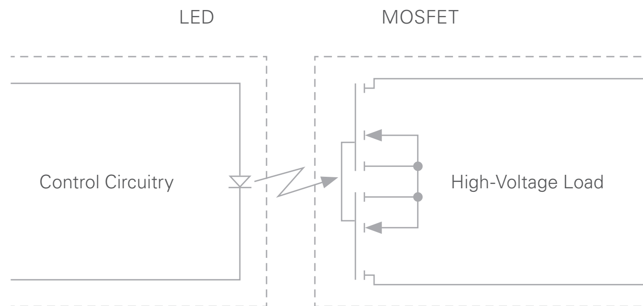


Figure 18. SSRs use photo-sensitive MOSFETs with an LED to actuate the device.

SSRs are slightly faster than EMRs, up to 300 cycles/s, because their switching time is dependent on the time required to power the LED on and off. Because there are no mechanical parts, SSRs are less susceptible to physical vibrations that could damage the relay, which provides an unlimited mechanical lifetime.

However, SSRs have their downsides. First, they are not as robust as EMRs and are easily damaged if used with signal levels outside of their rating. Second, they are expensive and generate more heat than alternatives. Finally, SSRs can have large path resistances, anywhere from less than 1 Ω to 100 Ω or more, because the connection is made through a transistor instead of a physical metal connection. Most modern SSRs have improved path resistance to make this less impactful.

Unlimited mechanical lifetime of SSRs make them an excellent choice when you have small-to-moderate signal levels and you need a relay that can last through many relay cycles. An example COTS SSR switch is the PXI-2533, which is a 4 x 64 matrix rated for 55 W of switching power and offers unlimited mechanical lifetime.

FET Relays

Similar to SSRs, FET relays are not mechanical devices and use transistors to route signals. Unlike SSRs, the control circuitry drives the gates of the transistors directly instead of driving an LED.

Directly driving the transistor gate allows for much faster switching speeds than any other type of relay mentioned, up to 60,000 cycles/s. Also, the lack of mechanical parts make FET relays much smaller and less susceptible to shock and vibration issues than electromechanical or reed relays, which affords FET relays an unlimited operational lifetime. However, FET relays have a much higher path resistance than any other relay option, typically in the 8 Ω to 15 Ω range, and they lack physical isolation and thus may be used with only low-level signals.

FET relays are an excellent choice for low-level signals and applications that require fast relay operation or unlimited mechanical life. An example of a COTS FET switch is the PXI-2535, which is a 4 x 136 matrix that can perform relay operations in less than 16 µs.

Below ○ Average ◒ Above ●

Capability	Armature	Reed	FET	SSR
High Power	●	◒	○	◒
High Speed	○	◒	●	◒
Small Package Size	◒	●	●	●
Low Path Resistance	●	◒	○	◒
Low Voltage Offset	◒	○	◒	●
Extended Lifetime	○	◒	●	●

Table 2. Comparison of Relay Options

Switch Expansion

If you build your own switching topology, then you can create a matrix or multiplexer to meet the exact dimensions of your application. However, many customers use COTS switches to reduce development effort and most COTS switches have fixed dimensions. Therefore, it is important to know how to combine multiple matrices or multiplexers to create a larger matrix or multiplexer.

Multiplexer Expansion

The easiest way to expand the channel count of a multiplexer is to directly tie the COMs of multiple multiplexers together. With this approach, there is some risk of shorting input channels together and possibly damaging your hardware. Therefore, you need to ensure that only one of the channels is connected to a COM at any given time. Some switching software, such as Switch Executive, gives you the ability to define software exclusions that prevent multiple input paths from being connected to a COM at any given time. Another downside to this approach is that the unused and unterminated routes result in stubs, which adds capacitance and degrades high-frequency performance.

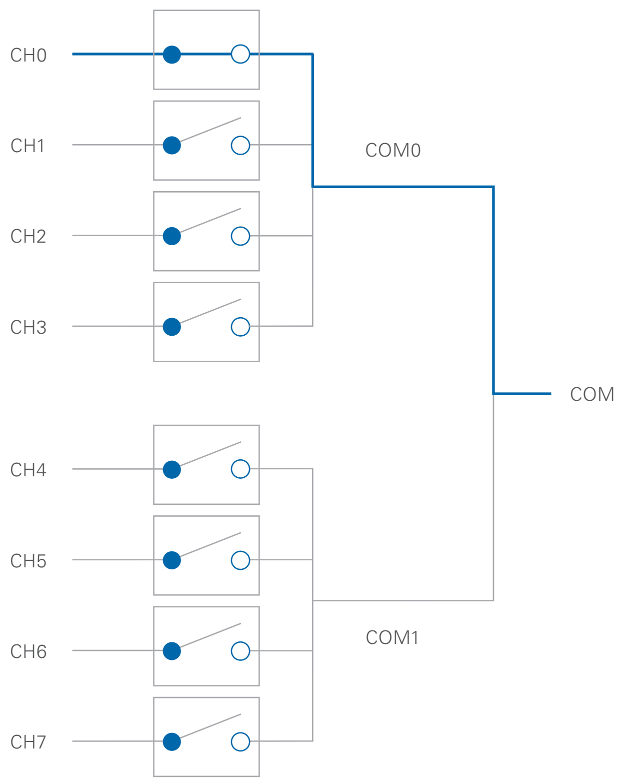


Figure 19. An 8 x 1 multiplexer is created by tying together the COMs of two 4 x 1 multiplexers.

Another approach is to connect the COMs of multiple multiplexers through an additional multiplexer, which inherently allows only one channel path to a COM but requires more multiplexers. However, this approach still results in PCB trace stubs that can degrade bandwidth performance.

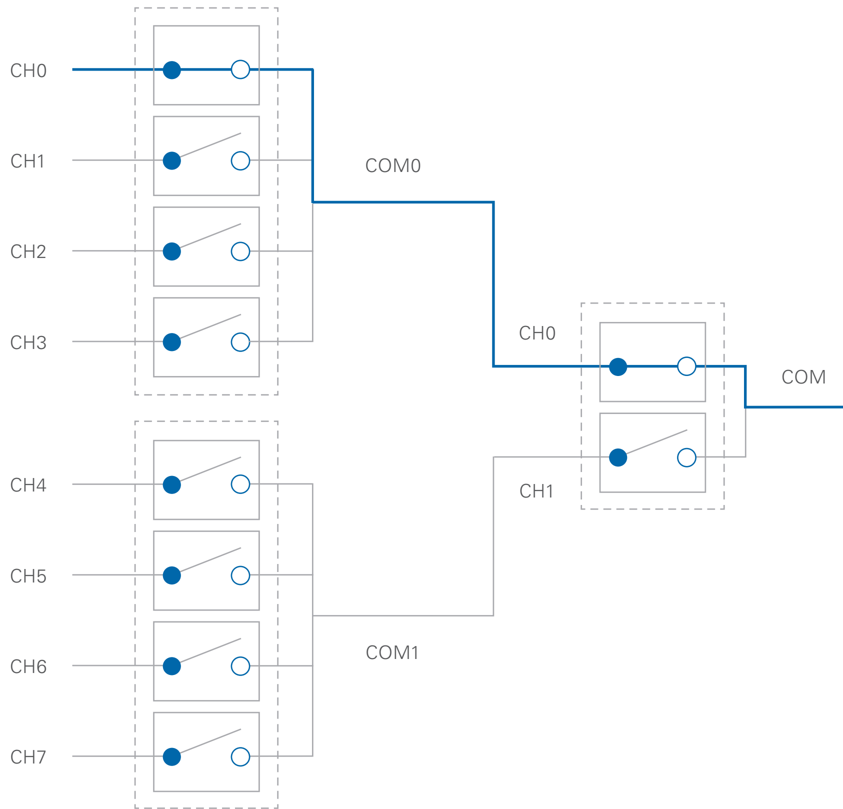


Figure 20. An 8 x 1 multiplexer is created by switching the individual COMs of two 4 x 1 multiplexers through an additional multiplexer.

For high-frequency applications, you should use Form C SPDT relays to create a large multiplexer. This option ensures there are no stubs along the active signal path, which helps increase the bandwidth of the switch.

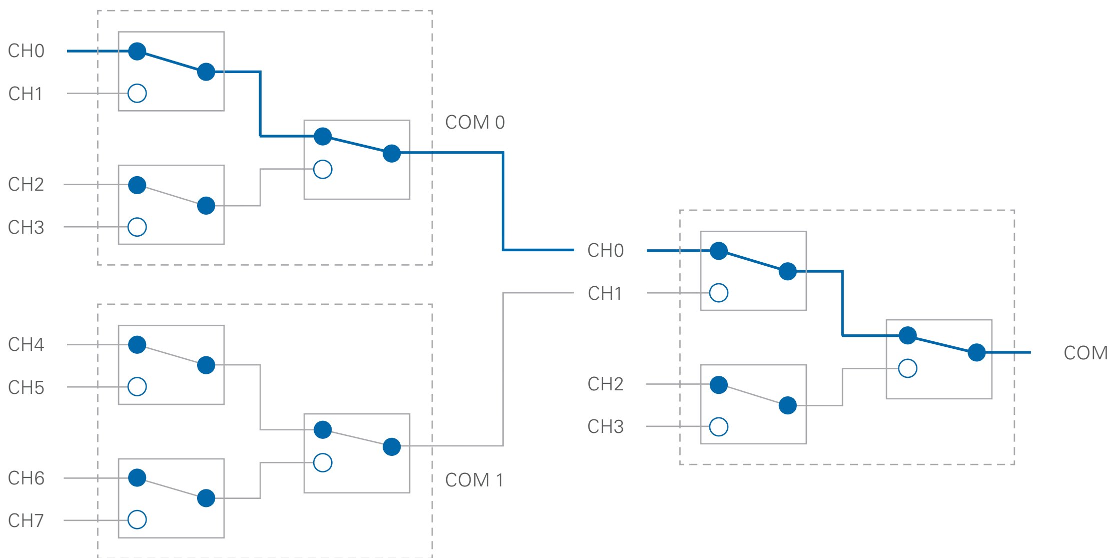


Figure 21. An 8 x 1 multiplexer is created by cascading three 4 x 1 multiplexers with Form C SPDT relays.

Matrix Expansion
Switch matrices can also serve as building blocks for creating larger configurations that are well beyond the size of a single COTS matrix switch. There are two ways to expand matrices. Column expansion is the process of connecting each row between two or more matrix modules, effectively doubling the number of columns within the expanded matrix. Alternatively, row expansion is the process of connecting each column of two or more matrix modules, doubling the number of rows within the expanded matrix.

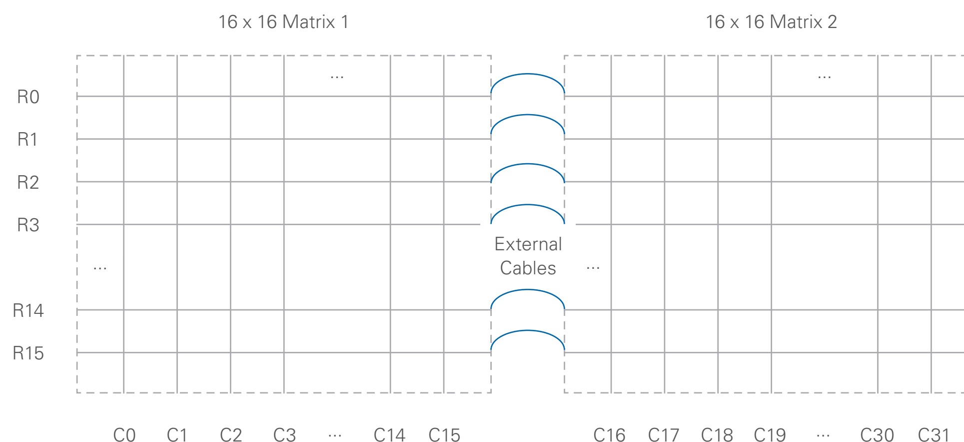


Figure 22. 16 x 32 Matrix Created by Column Expansion

For easy matrix expansion, some COTS matrix switches, such as the PXIe-2532B, offer specialized cables to combine matrices by easily connecting rows of multiple switch modules. However, all matrices are expandable, even if there are no prebuilt accessories to do so. To expand a matrix manually, you can use external wires to connect the rows or columns of individual matrices. For more information on matrix expansion, including examples and frequently asked questions, read the Matrix Expansion Guide for PXI Switch Modules.

Key Switching Specifications
In addition to relay type and switch topology, it is important to ensure that your switching subsystem maintains the signal integrity of the connected signals. Most switches fall into two categories based on signal types: low-frequency/DC and RF.

Low-Frequency/DC Switching Specifications
Switches typically advertise voltage and current ratings, but you should also pay attention to the maximum switching power specification, which refers to the upper limit of power that the contacts can switch. For example, a 150 V, 2 A switch may be limited to 60 W switching power and should not be used with 150 V at 2 A (300 W). Therefore, it is important to consider the maximum power of your signals in addition to your maximum voltage and current levels.

Signal frequency is also a tricky topic when dealing with switches. Many times, a signal is described with its fundamental frequency, which is fine for a simple sine wave. However, if you plan to switch square-shaped signals, or signals with sharp edges, then it is important to remember that a square wave has harmonic frequencies well above the fundamental frequency, which help shape the sharp edges. If you plan on switching a square wave, choose a switch rated for seven to 10 times the fundamental frequency of your signal. For example, if you were to route a 10 MHz square wave through a switch rated for 10 MHz, the output would look closer to a sine wave than a square wave.

For more information on switch bandwidth, read the Selecting Switch Bandwidth white paper.

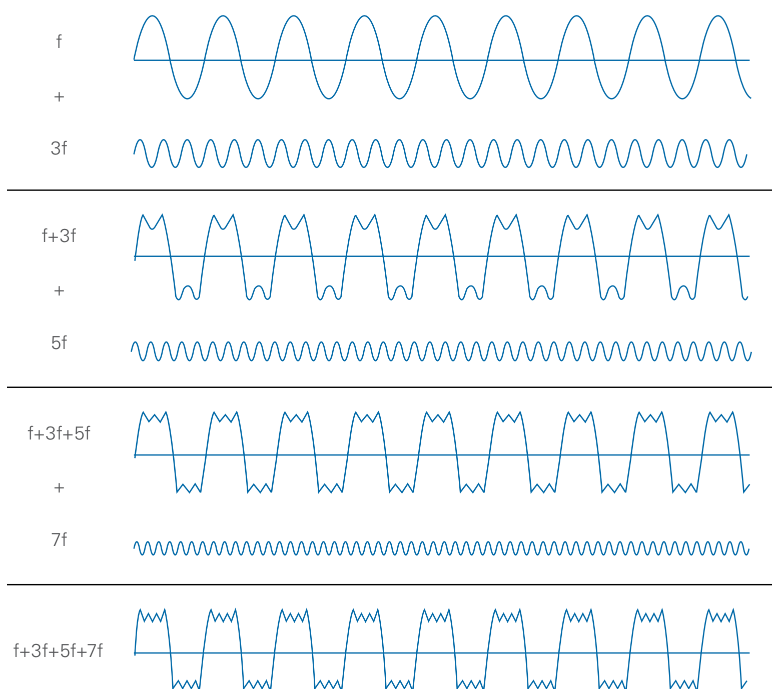


Figure 23. Square-shaped signals have harmonic frequencies well above the fundamental frequency.

Switch path resistance, thermal EMF, and offset voltage can affect low-level signal measurements, such as DMM resistance measurements. Therefore, you should select a switch that minimizes the effect on your measurements and design a measurement technique to compensate for these sources of error. For more information on how to reduce errors when switching low-level signals, see one of the following white papers:

Part I: How to Reduce Errors When Switching Low-Voltage Signals

Part II: How to Reduce Errors When Switching Low-Current Signals

Part III: How to Reduce Errors When Switching Low-Resistance Signals

RF Switching Specifications

A switch that is rated for more than 10 MHz or 20 MHz is often called an RF switch. RF switches typically have lower channel density to preserve signal integrity, so RF switches should be reserved for signal paths that require the increased bandwidth. However, topology and bandwidth do not provide you with enough information to select an RF switch.

All RF switches have a rated characteristic impedance, which is a transmission line parameter that determines how propagating signals are transmitted or reflected in the signal path. Component manufacturers specifically design their equipment to have a characteristic impedance of either 50 Ω or 75 Ω, because all components in an RF system have to be impedance matched to minimize signal losses and reflections. 50 Ω RF systems make up the bulk of the RF market and include most communications systems. 75 Ω RF systems are smaller in number and are prevalent mainly in video RF systems. It is crucial you ensure parts such as cables and connectors in addition to other instruments that may reside in the test system are all impedance matched.

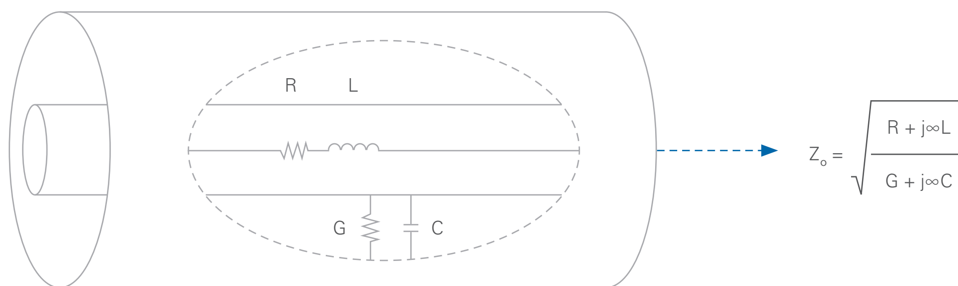


Figure 24. Characteristic Impedance of a Transmission Line

In addition to bandwidth and characteristic impedance, there are other RF switching specs that directly affect your signal integrity, such as insertion loss, voltage standing wave ratio (VSWR), isolation, crosstalk, and RF power. Insertion loss is a measure of the power loss and signal attenuation that occurs as a result of passing the signal through the switch. VSWR is the ratio of reflected-to-transmitted waves, specifically the ratio of maximum (when reflected wave is in phase) to minimum (when reflected wave is out of phase) voltages in the "standing wave" pattern. Isolation is the magnitude of a signal that is coupled across an open circuit and crosstalk is the magnitude of a signal that is coupled between circuits, such as separate multiplexer banks.

An interesting thing about RF switches is that all of these specifications vary depending on the signal frequency. Therefore, when choosing an RF relay or switch, you should compare specs at the specific frequency of your signals. Otherwise, it is easy to misinterpret the performance of an RF switch.

For more information on RF switch selection, read the Understanding Key RF Switch Specifications white paper.

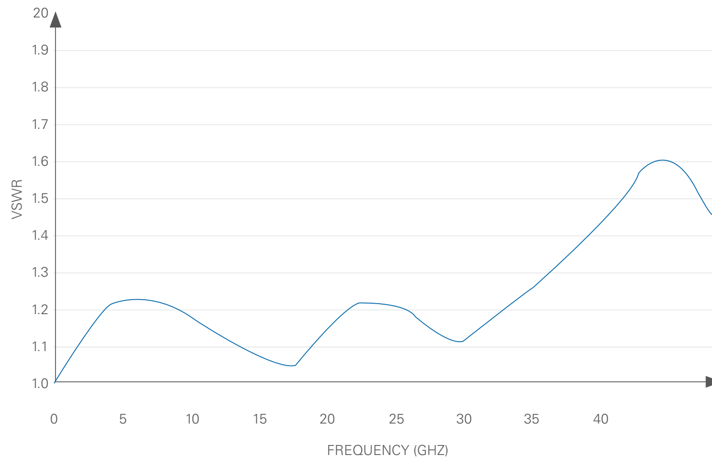


Figure 25. Many RF switch specifications vary with signal frequency.

Switching Tips and Tricks

When planning the switching portion of an automated test system, a few general tips can help you build an efficient switching system that preserves signal integrity.

Total Test Points Versus Simultaneous Connections

When using a matrix, consider the maximum number of possible connections and the maximum number of simultaneous connections. If you simply focus on the total number of possible connections, then you often end up with entire rows dedicated to each I/O pin of each instrument. However, this approach can lead to unnecessarily large matrices. For example, if you have 22 instrument pins and 106 DUT test points, then you might suggest a 22 x 106 matrix (2,332 relays) with the 22 I/O pins connected to the rows and the 106 DUT test points connected to the columns.

However, if you only need to connect at most four instrument pins at any given time, then the 22 x 106 matrix is unnecessarily large and wasteful. Instead, you could consider placing the instruments on 22 additional columns and use the rows for routing between columns. In this case, you would reduce the matrix size to 4 x 128 (512 relays), nearly 20 percent of the original size. This can save you space and money without affecting the test time or quality.

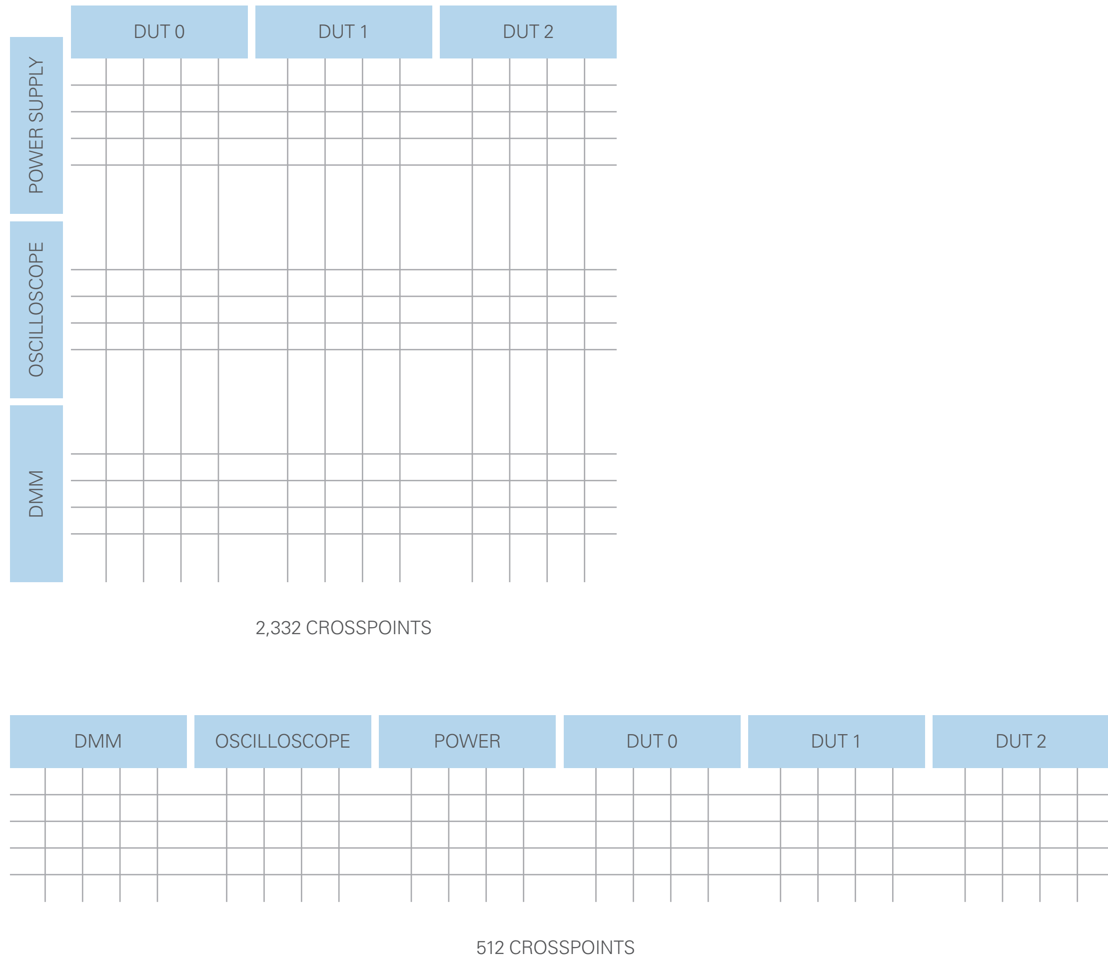


Figure 26. Place instruments on columns and use rows for routing to conserve matrix space during sequential test execution, but keep instruments in rows for faster parallel test requirements.

N-Wire Switching

Many matrix or multiplexer switch modules can switch two or four signal paths within a given topology instead of the standard 1-wire switching mode. You can use 1-wire switching to route various signals to an instrument that might reference a single signal or ground when performing measurements.

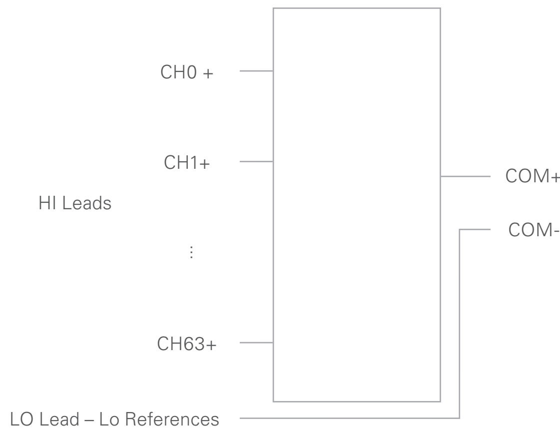


Figure 27. Single-ended multiplexers are great for measurements that reference a shared signal or ground.

Sometimes more than one signal needs to be switched at the same time. A 2-wire, or differential, switch provides two signal paths that you can control with one command. This provides an easy way to switch differential signals, which offers a great common-mode noise rejection. A 4-wire switch is typically reserved for 4-wire resistance measurements, which use two leads for excitation and another two leads to measure the voltage drop across the DUT.

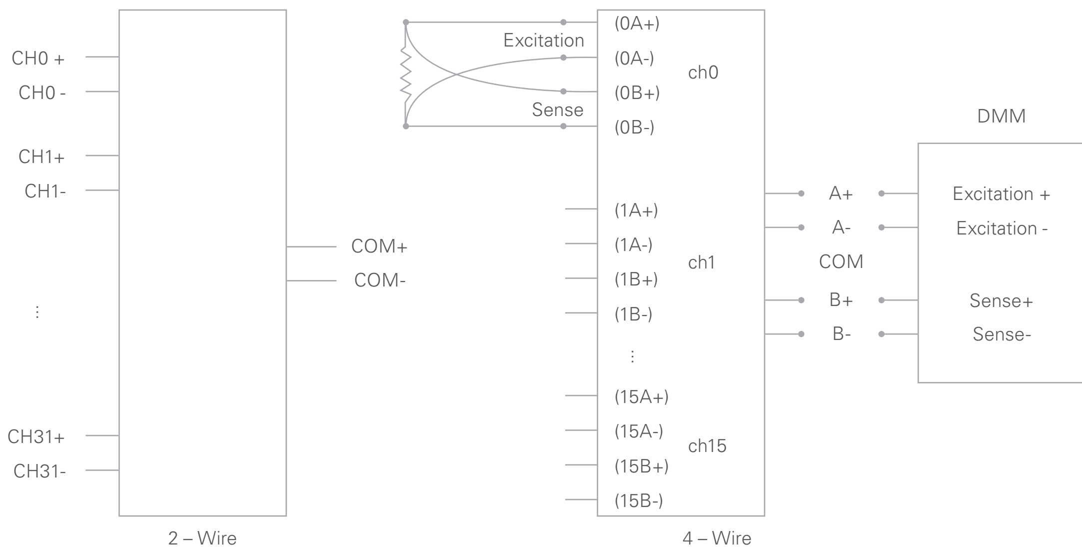


Figure 28. Switch multiple signal paths at the same time using 2-wire or 4-wire switching.

Switching Power

Many times, a test requirement plan includes maximum voltage and current levels, but instantaneous power is often overlooked. A switch or relay may be rated for 100 V and 2 A, but that doesn't necessarily mean it can handle 200 W. Many switches have maximum power ratings completely separate from their voltage and current ratings. For example, a common reed relay might be rated for 100 V and 500 mA, but it may have a maximum power rating of 10 W. Therefore, you should consider your maximum instantaneous power levels when selecting your switches.

Separate High-Level Signals From General or Low-Level Signals

Switches rated for high-power or high-frequency signals generally have lower density than switches for general-purpose signals. Therefore, you should isolate your high-power or high-frequency signals from your main switching system to preserve the channel density of the main switching system. If you try to build a single switch for all of your signals that is spec'd to handle your high-level signals, then it will likely end up large and expensive.

Compare RF Specs Based on Signal Frequency

When comparing RF switches, you should evaluate specifications based on signal frequency. Many RF specs, such as isolation, VSWR, insertion loss, and RF carry power, vary depending on the signal frequency. For an accurate comparison, look in the detailed switch specs to find the specs at the frequency of interest. Additionally, some switch vendors publish guaranteed and typical specs for each category, while others publish only typical specs that will appear to be much better than guaranteed specs.

Consider Hardware-Triggered Switches for Maximum Switching Speed

In many automated test scenarios, time is money. Many switches are controlled individually using software commands, with bus latency and software overhead added to each switching operation. Some switches offer hardware timing and triggering, which gives you the ability to load a list of switch connections to memory onboard the switch and use hardware triggers to advance through the list of connections. After each switching operation is complete, the switch can send out triggers to your instrumentation, starting the next measurement.

This operation is called switch handshaking and can eliminate the software overhead and bus latency associated with traditional software-triggered switches. Switch handshaking is especially important for faster relay types, such as FETs or SSRs, where the software overhead and bus latency make up a larger portion of each switching operation. An application using switch handshaking with reed relays might realize a 10X improvement in total switching time, while an FET switch might see a 100X improvement or more. The faster a relay, the more that switch handshaking can improve throughput.

Next Steps

NI Switch Products

Whether you are performing high-accuracy, low-speed measurements on a dozen test points or high-channel, high-frequency characterizations of integrated circuits, NI delivers a flexible, modular switching solution based on PXI to help you maximize equipment reuse, test throughput, and system scalability.

Learn more about NI PXI switch products

Switch Executive

Switch Executive is an intelligent switch management and routing application that accelerates development and simplifies maintenance of complex switch systems. The point-and-click graphical configuration, automatic routing capabilities, and intuitive channel aliases make it easy to design and document your test system.

Learn more about Switch Executive

NI Switch Health Center

To simplify relay maintenance and increase reliability in high-channel-count systems, the NI Switch Health Center verifies the condition of each relay by sending a test signal through every route in a switch. The health center alerts users if it determines a relay has failed, is stuck open, or is stuck closed, and reports changes in resistance to determine whether a relay is nearing the end of its usable life.

Learn more about the NI Switch Health Center

©2016 National Instruments. All rights reserved. National Instruments, NI, and ni.com are trademarks of National Instruments. Other product and company names listed are trademarks or trade names of their respective companies.

NATIONAL INSTRUMENTS

FUNDAMENTALS OF BUILDING A TEST SYSTEM

Test Executive Software

CONTENTS

Introduction

Background

Features of a Test Executive

Conclusion

Next Steps

Introduction

Most test systems are designed fundamentally around two concepts: efficiency and cost. Whether working in the consumer electronics industry or in semiconductor production, test engineers are concerned about individual test time and total throughput of a test system, and how these affect resources. When applications grow large enough to constitute multiple tests, a variety of instruments, and several units under test (UUTs), they inevitably require the oversight of test executive software to continue to address their cost and efficiency concerns.

Test executives are typically implemented as in-house solutions, or purchased as a commercial off-the-shelf (COTS) products. In the prototypical build versus buy argument, a test architect must determine whether it makes more sense to write a custom test executive or to invest and integrate an existing solution. Before deciding whether to build or buy a test executive, it is necessary to understand the purpose and core functionalities of this kind of software. This guide summarizes key functions of a test executive and explores practical scenarios to apply this knowledge.

Background

A test executive can automate and streamline large test systems. Sitting at the top of the software stack, it consolidates common functions, such as test execution, result collection, and report generation, up from the individual test level. The features of this solution are not unique to a particular UUT, so a variety of applications can use the test executive as a framework. This means that developers writing test code in G in LabVIEW software, C, .NET or other languages can focus on the intricacies of testing a particular device, while common functions across all UUTs are maintained at the top-level test executive. Overall, the test executive defines such common functions in a manner that proves efficient from a development, cost, and maintenance perspective.

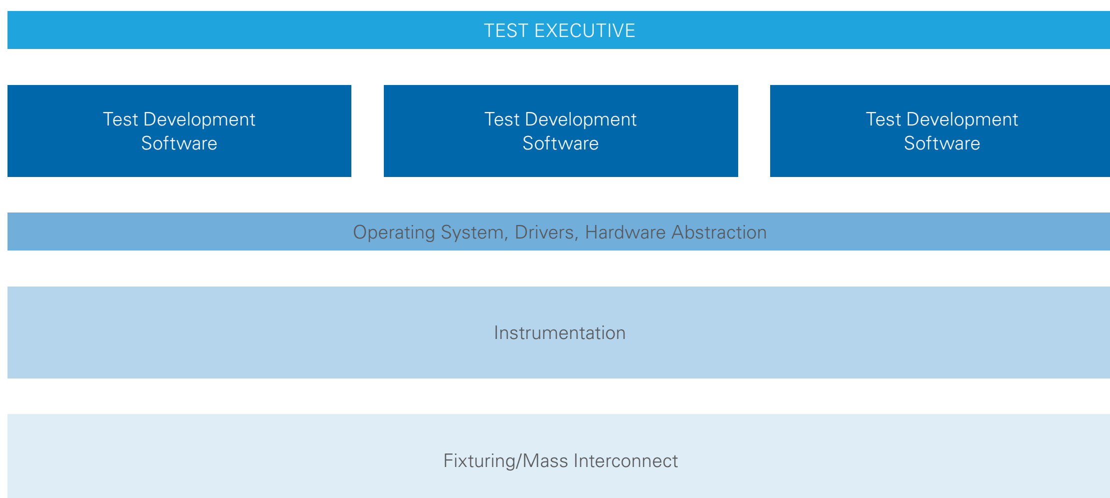


Figure 1. Test executives allow the separation of individual test development from the architectural needs of the entire test system by accomplishing tasks common across all tests at a higher level of abstraction.

Features of a Test Executive

Depending on the size of a company, the scale of a particular tester, and the variety of devices under test, complexity of a test executive can range from simple to advanced. This guide outlines common features that this software might contain. Some features are crucial to all implementations of a test executive, while others represent additional functionality that may not be strictly necessary. Each feature outlines an estimated amount of development time to complete. These estimates are based on experience with hundreds of automated test customers, as cited in Test Executive Software—Build or Buy? A Financial Comparison Using NI TestStand.

Test Sequence Development Environment

A test executive provides a development environment in which to architect test sequences. This feature is both fundamental— providing the development interface for the whole execution— as well as complex. Sequence architecture encompasses the ability to implement branching or looping logic, a means to import test limits, and the specification and organization of individual test code. Interfacing with test code requires flexibility across a variety of built formats, such as DLLs, VIs, and scripts, as well as integration across different development environments. Test executives may also use test code that originates from a source code control provider.

Implementing a test sequence development environment in a custom-built test executive can take around 100 person-days to complete, whereas a commercial solution provides this environment outright. This feature requires the most development time for an in-house solution because of the range of functions that a development environment provides. However, it is fundamental to the sequence architecture experience and cannot be omitted.

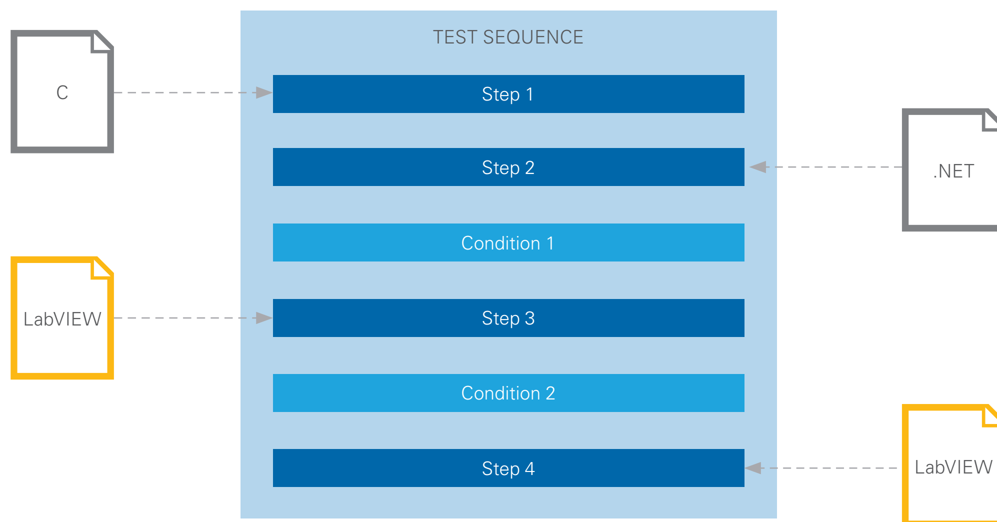


Figure 2. A productive sequence development environment gives test engineers the ability to develop and debug complex sequences that call into existing test code.

Custom Operator Interface

The operator interface is the display through which the operator interacts with the test system. It typically allows for the selection of key input parameters, such as UUT identifier, test sequence to execute, or report path. It also contains a Run or Start button to control execution. Many large test systems today require a professional GUI differentiated by application or company and written in the programming language of developer choice. In addition to customization, this highly functional interface includes the ability to load, display, and run test sequences complete with interactive user prompts, execution progress indicators, visualization of test data, and localization.

Implementing a custom operator interface can take a range of eight to 32 person-days' worth of development time. A COTS solution can reduce this estimate because of existing libraries and UI controls. Developing a custom operator interface can be a nontrivial time investment, regardless of whether the test executive solution is built or bought. Test engineers who do not feel this component is crucial to their system may instruct operators to work through the development environment instead.

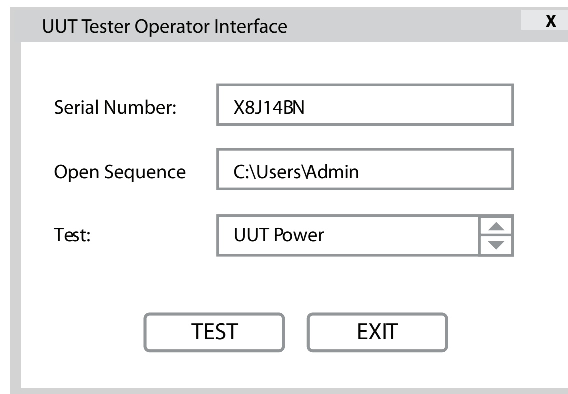


Figure 3. A customer operator interface uniquely identifies the UUT, company, application, test, and role of the operator for a given test sequence.

Sequence Execution Engine

A core provision of the test executive is a sequencing engine. The sequence execution engine is responsible for all the actions required to evaluate a UUT. This includes calling individual test code, creating a flow for execution between tests, and managing data between tests. The sequencing engine is what executes a given test sequence, whether in the development environment, through a custom operator interface, or on a deployed tester.

Implementing a sequence execution engine requires a minimum of 15 person-days to develop in-house. However, it is a must-have feature of all test executives.

Results Reporting

Given the abstracted role of the test executive, this piece of software is responsible for consolidating individual test data, storing temporarily into memory, and publishing comprehensive test results. Reports can come in a variety of formats, including XML, text, HTML, and ATML. Data may also be pushed to a database following execution. The test executive makes this variety in formats possible through extensible reporting options. Results reporting is a necessary component of many test systems.

Developing result collection and a report generator from scratch can take around 15 person-days, depending on the specific report required. Given a built-in report generator in a COTS solution, results reporting can be customized to meet the needs of an application in a person-day or less.

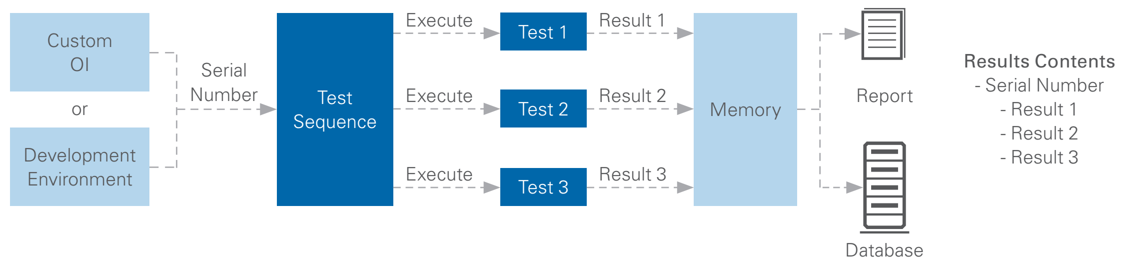


Figure 4. Part of a test executive's role in a test system is to consolidate results across an execution and publish to a report or database.

User Management

It may be necessary to separate roles and responsibilities at the test executive level. User management tools effectively compartmentalize the responsibilities between the overarching test architect, the individual test developer who writes and debugs test code, and the operator or production manager who runs the test. Functions available to a given user may even be password protected to prevent misuse of the test sequence.

Implementing a user management system in a custom test executive takes about five person-days' worth of development time. Although not necessary for the use of a test executive, user management tools do not require a significant amount of developer effort to implement and can simplify the enforcement of test executive responsibilities.

Privilege	Architect	Developer	Operator
Edit	√	—	—
Save	√	—	—
Deploy	√	√	—
Loop	√	√	—
Run	√	√	√
Exit	√	√	√

User	Level
Mark	Operator
Larry	Operator
Julie	Developer
Scott	Developer
Lauren	Architect

Table 1. Similar to Windows file permissions, a user manager separates the roles and responsibilities associated with a test executive.

Parallel Testing Capabilities

Parallel test involves testing multiple devices at the same time, while still maintaining proper code-module performance, result collection, and UUT tracking. Parallel test approaches range from pipelined execution, where test order is maintained, but the test executive can test across multiple sockets concurrently, all the way to dynamically optimized, batch, or other complex execution styles.

Implementing parallel test is typically the most time-intensive for a test executive developer, and can take 100 person-days to develop from scratch. Although parallel test may require a large amount of time to develop, the ability to scale up an execution to mitigate throughput needs in a large test system is often crucial. Many organizations do not consider parallel test when first implementing a test executive, and learn later that it is a function they ultimately need and cannot settle on.

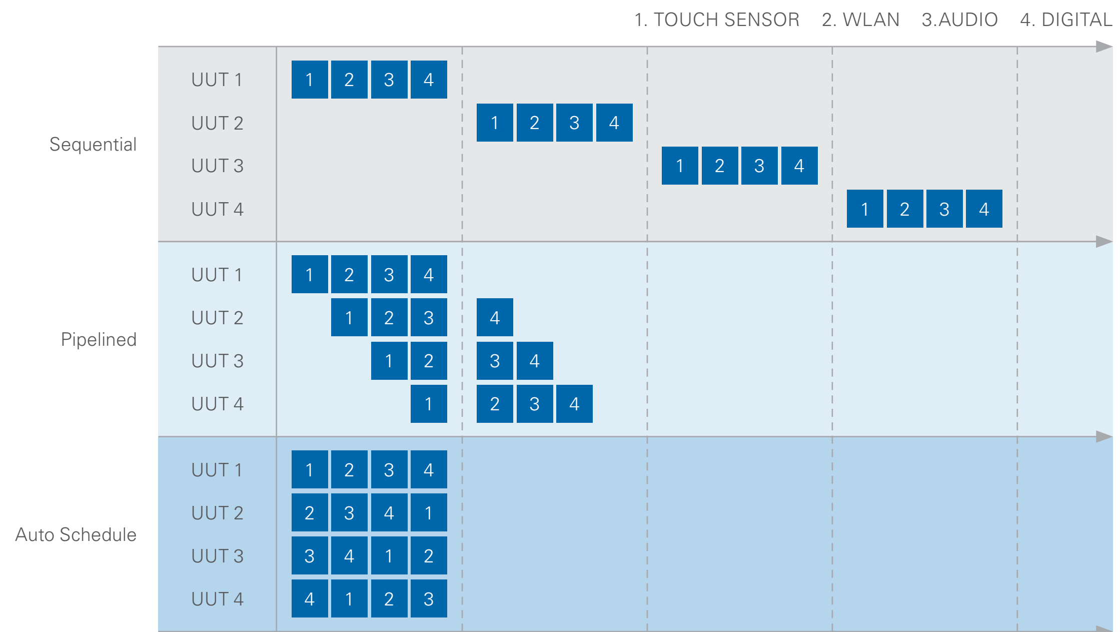


Figure 5. Parallel test capabilities allow for dramatic increases in system throughput without a re-architecture of the test executive.

Unit/Device Tracking and Serial Number Scanning

When testing across multiple UUTs, it can be necessary to uniquely identify and track each device tested. This information can be stored alongside test results for specific analysis at the unit or batch level, or to pinpoint the source of error when things go wrong. Device tracking can range from manual entry by an operator on a keyboard, to a fully automated scanner that loads UUT information after reading a barcode.

Developing this type of functionality can take five person-days from the ground up, or about one person-day to customize when provided by a COTS solution. UUT tracking is not required for every test system. However, it is useful where high-volume, high-throughput testing is needed, such as the semiconductor or consumer electronics industries.

Test Deployment Tool

Most large test systems are not architected in isolation; they represent solutions for multiple test sites or for an entire production floor. A test executive plays a key role in system deployment by providing a mechanism or utility to package the entire software stack into a built, distributable unit. A test system can be distributed in a variety of ways—an architect may be looking to deploy an image of the test system or a fully functional installer containing all necessary dependencies and run times. More information on this topic is covered in the white paper from the Fundamentals of Building a Test System series, Software Deployment.

Deployment is a nontrivial task, and it can take a team of developers as many as 20 person-days to implement from scratch. With an out-of-the-box deployment utility from a commercial test executive, it still may take three person-days' worth of time to successfully deploy. Given the applicability of this feature to multiple test sites, it is often necessary to have in a test executive solution.

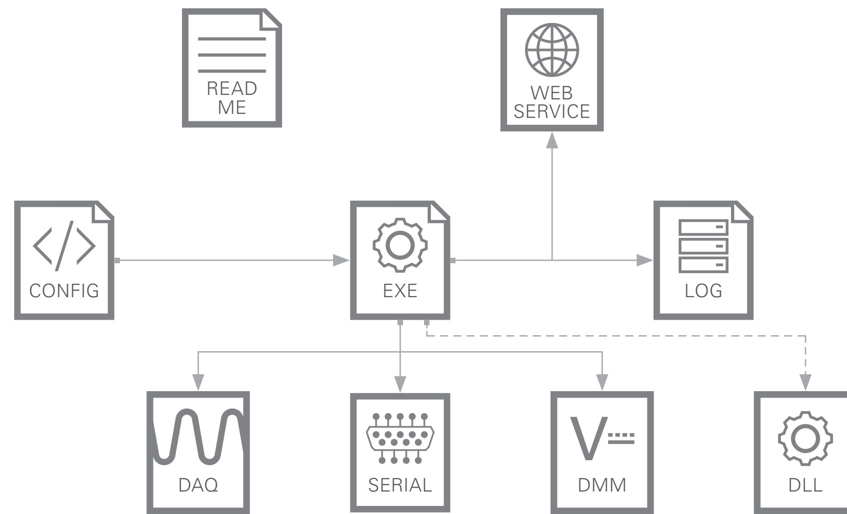


Figure 6. Deployment involves packaging all necessary components of a test system through a deployment tool or build server, before distributing to the wanted test stations.

Maintenance

Just as with any other component in a large test system, test executives must be properly supported to ensure their performance over time. This encompasses expanding to include new tests, maintaining compatibility across software or OS upgrades, and fixing any bugs that are detected. Maintenance of a test executive solution even extends to the realm of documentation. This is a crucial resource that operators, developers, and architects rely on when working with a test executive.

Although it is difficult to predict the needs of a given tester, 15 percent of the initial time spent to develop a custom test executive is spent annually in maintenance. Total development time includes the estimated 20 days required to produce adequate documentation. The granularity of support for a test executive can vary, which changes these cost estimates dramatically. However, it is inadvisable to implement any test executive solution with the minimum in maintenance efforts.

Practical Scenario 1

Jonathan is a test engineer in the design lab at a small company that provides low-cost consumer electronics. A remote controls each device, and Jonathan specializes in writing test code to validate the transmitter-receiver communication between the remote and the prototype, before the device is sent to production. With his company's recent expansion, Jonathan has less time per device to perform the requisite testing. He needs to automate the execution of his existing validation code, so that he can spend more time writing code for new devices. Therefore, he decides to employ a test executive to sequence through his test code.

The table below shows the needs that Jonathan identifies in a test executive.

FEATURE	IMPLEMENTATION
Test Sequence Development Environment	Jonathan needs a development environment in which to architect his sequences. The test code he has been working with is already fairly modular, so he should only have to call and loop over test code within this environment.
Custom Operator Interface	Jonathan wants to be able to execute a set of test code with minimal interaction. He wants to have to specify only a few relevant parameters to identify the device, wanted tests, and report path. However, given that he is the end-use operator, it is not crucial to have an interface separate from the development environment.
Sequence Execution Engine	This is the core need for Jonathan's test executive solution. Each test consists of several individual LabVIEW VIs that must be executed sequentially.
Results Reporting	The existing test code currently prompts the user to generate a new Technical Data Management Streaming (TDMS) file for a given prototype. Each subsequent VI deposits minimal test results into this same file. The test executive needs only to automate the creation of this TDMS file, and then execute the remainder of the test code to generate results according to convention.
User Management	User management is not a priority, because Jonathan plans to architect, develop, and operate this test executive.
Parallel Testing Capabilities	Jonathan executes his tests on only one prototype at a time, so UUT volume is not a concern.
Unit/Device Tracking and Serial Number Scanning	Given that testing is done on design prototypes, there are no assigned serial numbers to track. Instead, Jonathan tracks each UUT by a unique name that the operator enters at run time.
Test Deployment Tool	Jonathan does not intend to deploy this code to additional testers. His test bench is unique to the design lab, and separate from the manufacturing facilities.
Maintenance	This project belongs exclusively to Jonathan. He will implement and maintain whatever test executive is selected. He does not plan to document his work, as he will be the sole person to work on and use this test executive.

Table 2. Jonathan's needs in a test executive solution center on simple automation of existing validation code.

Jonathan decides to build a test executive in-house. He does not have complex sequencing or reporting needs, and does not have plans to deploy this system to other users or test stations. If he purchased a commercial solution, he would not see the return on investment as the majority of features would not be used. Instead, relying solely on his software knowledge and previously purchased application software, he can develop a sequencer to meet his needs in as little as 10 person-days.

Jonathan builds his test executive in LabVIEW software. He architects a solution with a simple interface that gives operators the ability to call a predetermined set of test steps and select the path of the TDMS log. Jonathan can occasionally make small changes to the sequencer as he introduces additional tests for a new prototype. Overall, the design lab sees an increase in productivity thanks to the implementation of this sequencer.

In this particular case, a custom-built test executive proved to be the best option for Jonathan and his criteria. Often, an in-house solution is the first step taken when scaling up to sequencing or full automation, and may be more appropriate overall for a test bench application when compared to the needs of a production setting.

What If…

- After a few months, a new test engineer is hired into the design labs and begins to assist with the testing process. How will this engineer learn how to operate the sequencing tool, or effectively troubleshoot any errors or bugs that appear?
- Jonathan transfers to another department, or leaves the company. How is the knowledge required to update or fix the sequencer maintained?
- It becomes necessary for the sequencer to perform a functional evaluation of an entire prototype. How would it incorporate additional test code that different engineers write in other languages, with different programming paradigms and reporting techniques?
- The test executive is ported to a production setting to ensure consistency in testing. Can these solutions scale up to such needs?

Practical Scenario 2

Dave's company is designing a new functional tester to be implemented at the end of a manufacturing line. Currently, UUT testing is performed by manually executing across a series of existing, disaggregate pieces of code. This process significantly limits throughput of the line, and Dave wants to employ a test executive in automating this process. The company does not standardize on a test executive, and each group typically chooses its own from within a small pool of commercial solutions and innumerable custom-built solutions.

The table below shows the set of requirements that Dave outlines for the tester.

FEATURE	IMPLEMENTATION
Test Sequence Development Environment	A productive development environment that supports key features of a test executive is a must. The environment must enable the sequencing of LabVIEW, .NET, and Python code.
Custom Operator Interface	Dave ultimately wants an operator interface that is customized to the company. He also wants to remove most functionality beyond a Run button.
Sequence Execution Engine	This is an obvious need for this system to address throughput needs.
Results Reporting	Currently, each test individually logs data to an SQL database. There is a need for consolidated result collection by the test executive, with aggregate results communicated to the database and identified by a serial number.
User Management	The majority of interaction with a tester occurs at the production level by the operator. Dave prefers a user management tool or customizable interface that removes development privileges from the operator's view.
Parallel Testing Capabilities	As long as tester throughput matches production throughput, Dave does not need to test multiple UUTs at once.
Unit/Device Tracking and Serial Number Scanning	A serial number identifies each component and assembled UUT. A barcode scanner is used to track such information. The test executive must be able to propagate such information across the different tests it executes.
Test Deployment Tool	Dave needs to deploy the final product to 10 additional testers.
Maintenance	The test engineering department will maintain the test executive, either in full capacity for an in-house solution or where needed for a COTS option.

Table 3. Dave's evaluation of test executive software is driven by underlying throughput requirements on functional testers.

To make his decision, Dave also weighs the financial considerations of the tester. He estimates that a new tester will consist of a large, high-performance PXI chassis and embedded controller pair. Because of the nature of tests required to evaluate the UUT, the chassis will contain several modules that range from DAQ cards and PXI instruments, such as digitizers and arbitrary waveform generators, to RF test equipment. The cost of each tester will sit at around $100,000 USD regardless of the test executive solution.

When evaluating the software stack, Dave notes that purchasing a COTS solution adds to the project cost. A development license of the test executive costs a few thousand dollars, with the added cost of $500 USD per additional tester for a license to deploy.

Dave believes he can save on test executive cost by building a custom solution in Python. The language is open source and the development environment is free—both are benefits he believes will more than offset the additional development time required to build a test executive in-house.

The test engineering team is proficient in Python, which delivers core functionality—a sequential sequencing engine, database connectivity, and code reuse of their existing tests—in the required timeframe. The test executive is successfully deployed to the manufacturing lines. The test engineers are occasionally called in to fix bugs in one or more of the testers.

What if…

- Production demands on the manufacturing lines increase, such that the existing test executive cannot meet throughput needs. It is necessary to scale up to parallel test.
    - How much additional development time would it require to attempt to implement this functionality? How does this affect the cost comparison of a custom versus COTS solution?
    - Assume throughput needs of the tester cannot be met because of known multiprocessing limitations in the Python language. Dave's team is faced with purchasing additional hardware to reuse the current solution, or pursuing another test executive altogether. How does this further affect the cost comparison of a custom versus COTS solution?
- The test engineering team cannot always service or upgrade the test executive because of other priorities.
    - How is production affected when such needs arise and the team cannot help? How does this downtime factor into system maintenance costs?
    - How is the time that the test engineering team spends maintaining the tester quantified? How does this factor into system maintenance costs?

Practical Scenario 3

Karen works at a company that designs and produces small medical devices. Each product has its own fully automated production line. Although each group enlists a test executive for top-level system management, the company has not standardized on a solution. Recently, a new test manager has come aboard and expressed interest in test executive standardization. Karen is tasked with the responsibility of selecting the commercial solution, existing in-house product, or new development effort to act as the de facto test executive.

Karen compiles the following list of requirements across the assorted groups responsible for each product.

FEATURE	IMPLEMENTATION
Test Sequence Development Environment	The test developers require a flexible development environment that, specifically, can interface with their LabVIEW and VB.NET code. Tortoise SVN is used for source code control, and integration with this tool is required.
Custom Operator Interface	The test manager wants to customize operator interfaces according to the product being built or tested. Operators have reported they want a progress indicator to update test status when overseeing a tester.
Sequence Execution Engine	A definite requirement for all testers.
Results Reporting	All production systems must conform to a company-wide, HTML reporting standard.
User Management	The test engineering team consists of a few system architects and a larger number of test developers. The test manager wants to separate responsibilities between these two roles.
Parallel Testing Capabilities	When performing functional testing on an assembled unit, production lines evaluate one UUT at a time. However, board-level testing should be optimized to execute as quickly as possible. To meet the needs of all testers, parallel test is needed.
Unit/Device Tracking and Serial Number Scanning	UUT information is tracked by operator input for each product and board in the company.
Test Deployment Tool	The company has a dedicated team of test engineers that writes test code. This team must be able to deploy from the development environment in their lab to the wanted production setting. Currently, this is accomplished manually.
Maintenance	The test manager requires a formal maintenance plan as part of the standardization effort. Part of this plan needs to accommodate an OS migration that the company is facing later this year when their current selection goes end-of-life.

Table 4. Karen's interest in a test executive solution stems from standardization needs across a variety of testers.

Given this criteria, Karen eliminates all of the existing in-house solutions. Most of them were architected as part of a focused effort to get a single tester off the ground. There is little consistency in architecture that would lend for extensibility into other production lines, specifically in terms of sequencing needs, operator interface customizations, and effective deployment practices. Additionally, it has already proven difficult to track down the test engineer responsible for a given test executive when a problem occurs in the software, or a modification is made to the device.

Instead, Karen proposes a commercial solution to her manager. The test executive is made by a well-known vendor whose other hardware and software tools are already used in the testers. Out-of-the-box features of this test management software can meet the range in sequencing paradigms that testers require, and employ the specific reporting format needed. The test executive includes a set of tools designed to meet some of the other testers needs, including a user management tool and deployment utility. Given that a commercial vendor maintains it, Karen's manager should not have to worry about incompatibility across OS migrations later.

Karen's company is ultimately successful with their decision. Overall, the test executive provides a flexible framework that scales across the different production lines. Standardization across a purchased test executive comes with additional benefits that the company can use. The vendor provides training to facilitate the test engineer's acclimation to the new software. Part of their purchase of the test executive includes a maintenance contract, wherein the vendor agrees to provide routine patches and upgrades. The company also has access to technical support resources that can assist in troubleshooting their test sequences.

The commercial solution remains the standard at Karen's company. When test engineers need to be replaced, because of promotions, retirement, or natural attrition, the test manager can hire an individual with experience in the test executive. The company successfully migrates from an obsolete OS up two complete versions while maintaining their selection in test executive. As new products are developed, the extensible architecture can continuously meet production needs.

Conclusion

Regardless of company size, industry, or individual test criteria, it is necessary to implement a test executive for top-level system management. This implies introducing a degree of abstraction that separates common functions of a system from the specific functionality of test code. A complete evaluation of test executive needs is necessary before architecting the ultimate solution. Many test engineers grapple with the decision to build or buy their test executive. Selection of one path over another involves careful consideration of each solution's benefits from a cost, functionality, and maintenance perspective.

Next Steps

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 results 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

©2016 National Instruments. All rights reserved. LabVIEW, National Instruments, NI, NI TestStand, and ni.com are trademarks of National Instruments. Other product and company names listed are trademarks or trade names of their respective companies.

NATIONAL INSTRUMENTS

FUNDAMENTALS OF BUILDING A TEST SYSTEM

Hardware and Measurement Abstraction Layers

Grant Gothing, ATE R&D Manager, Bloomy Controls

CONTENTS

Introduction

Background

Approaches

Practical Scenario 1

Practical Scenario 2

Next Steps

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.

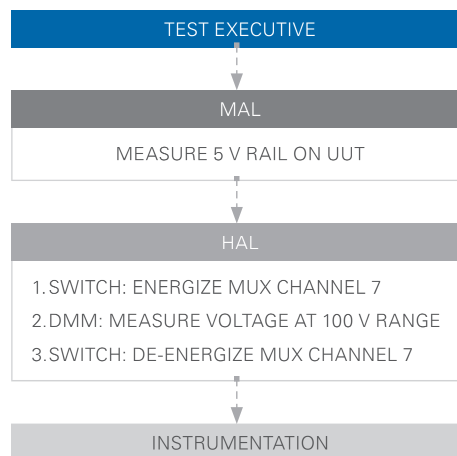


Figure 1. High-Level Overview of an Abstraction Framework

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.

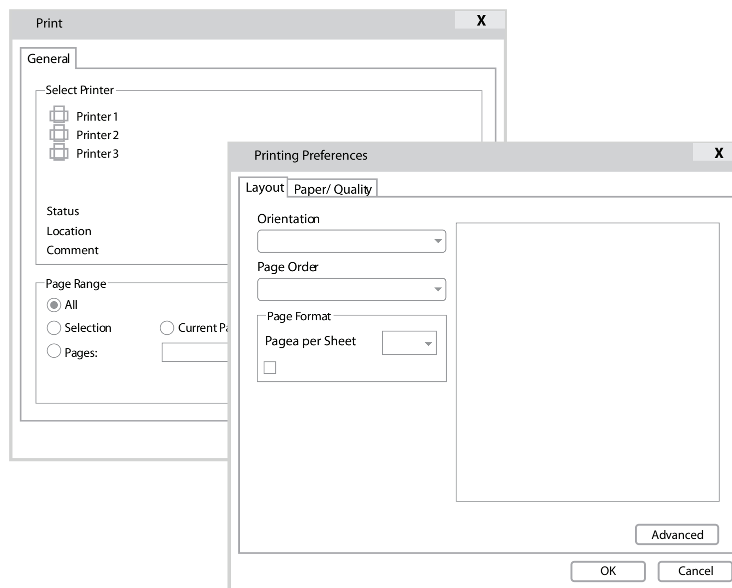


Figure 2. Printer dialogs are an excellent everyday use of a HAL/MAL.

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).

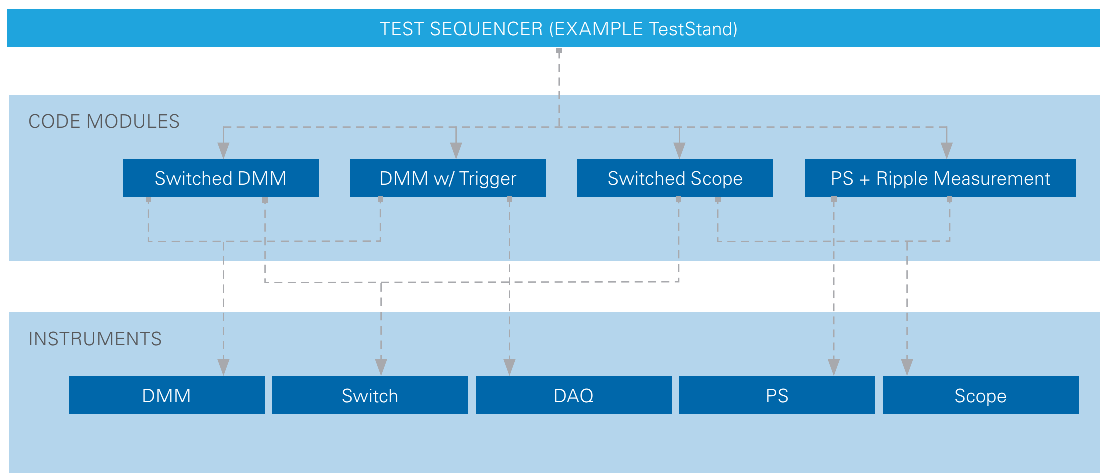


Figure 3. The Anatomy of Nonabstracted Automated Test Software

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.

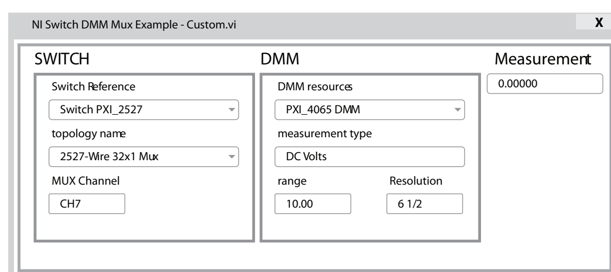


Figure 4. Front Panel of a Typical Multiplexed DMM Measurement Application in LabVIEW

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.

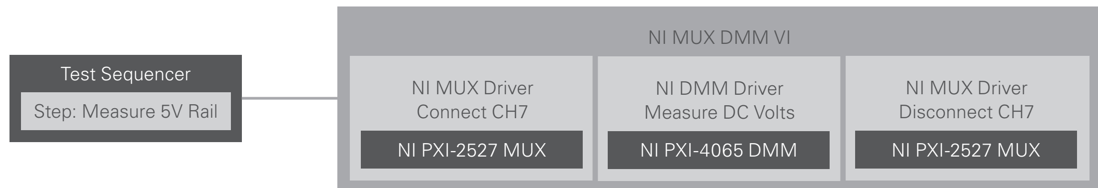


Figure 5. Nested Command Calls to Perform a Multiplexed DMM Measurement

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

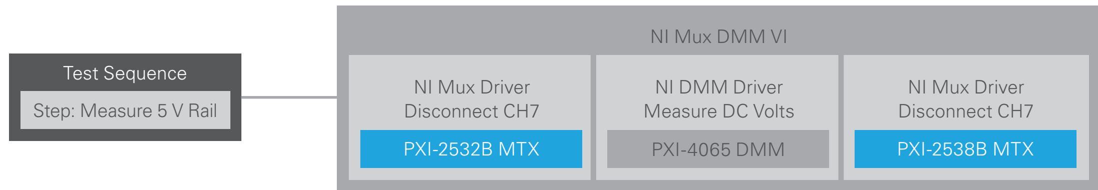
2. **Driver**—NI Mux driver changes to NI Matrix (rows/columns instead of channels)

3. **Code Module**—NI Mux DMM VI must be changed to an NI Matrix DMM VI

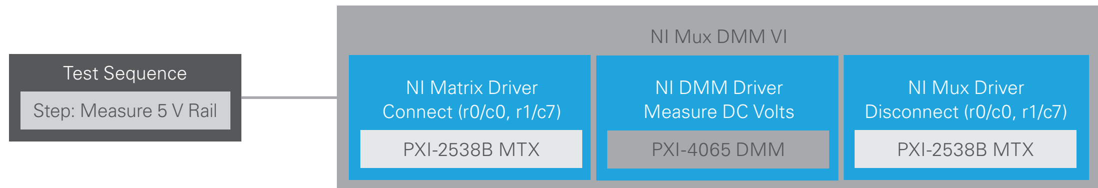
4. **Function Call**—The test executive call to the code module must be updated

5. **Sequence**—Test sequence must be updated for every call to that code module

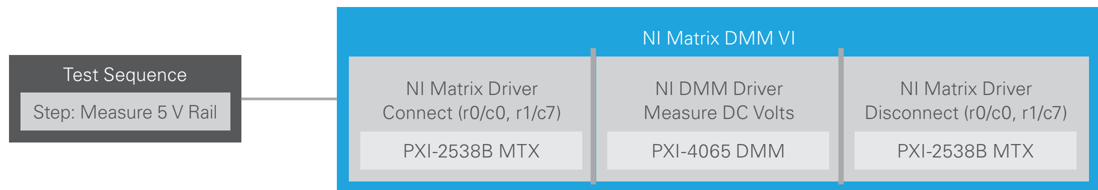
STEP 1: INSTRUMENT CHANGE



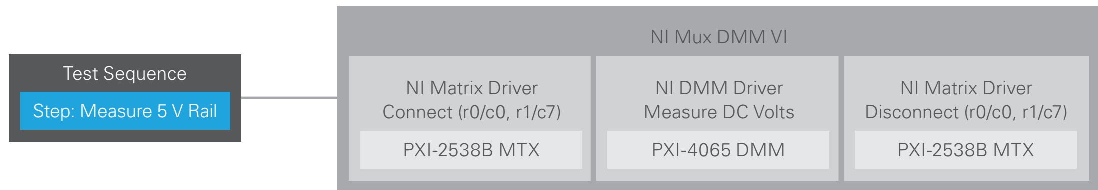
STEP 2: DRIVER CHANGE



STEP 3: CODE MODULE CHANGE



STEP 4: FUNCTION CALL CHANGE



STEP 5: TEST SEQUENCE CHANGE

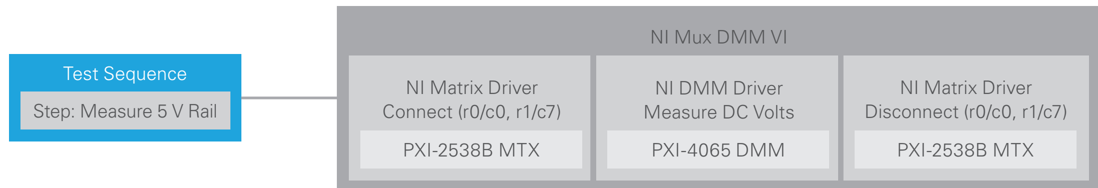


Figure 6. Nonabstracted Changes Required by Chain of Dependencies

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.

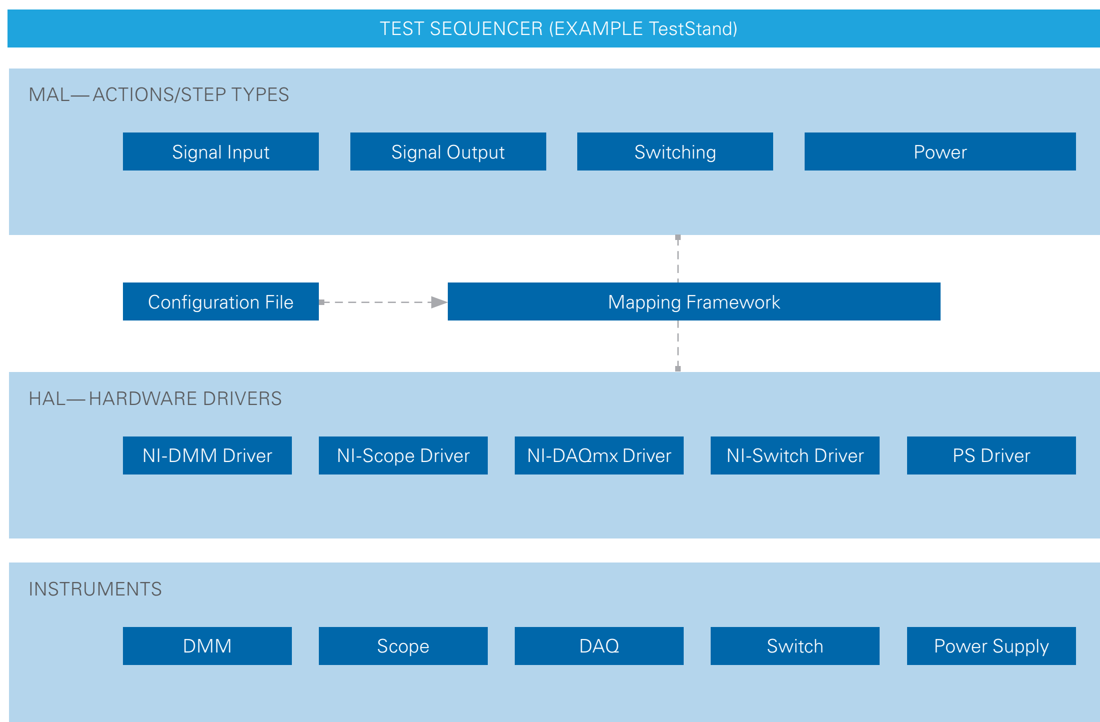


Figure 7. Anatomy of Abstracted Automated Test Software

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.

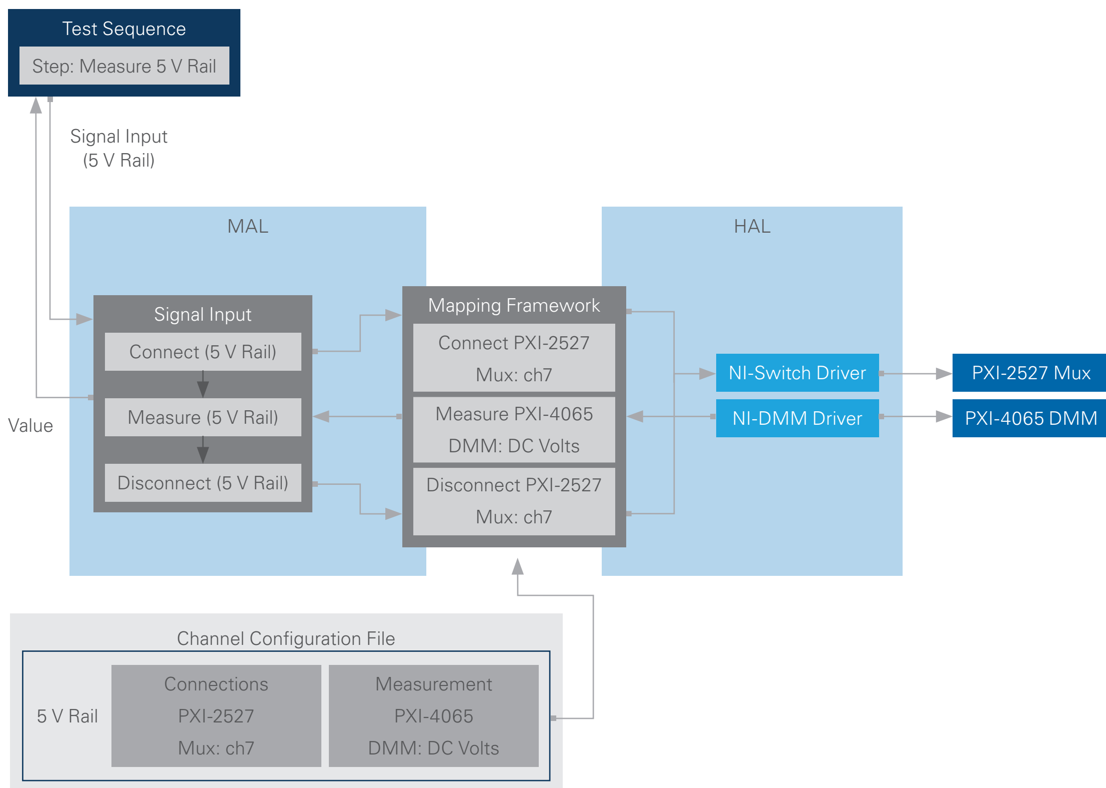


Figure 8. Function Calls for a DMM Measurement With an Abstraction Framework

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.

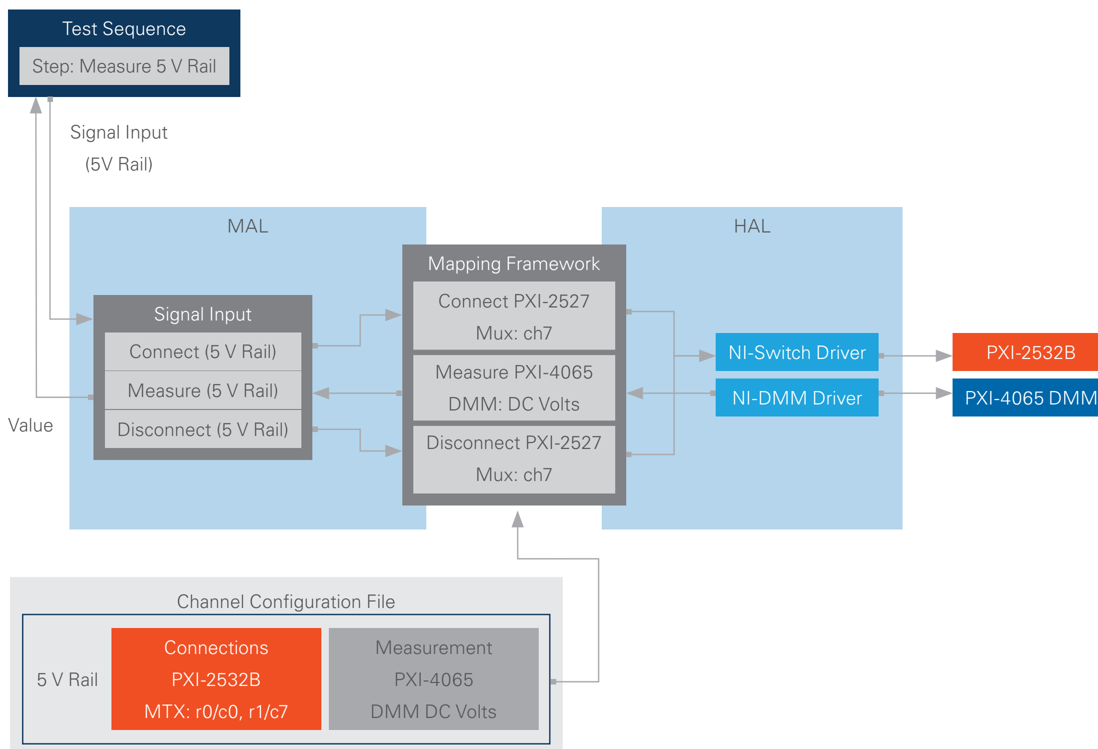


Figure 9. Abstraction makes it easy to update hardware with minimal software updates—just updates to the configuration file.

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.

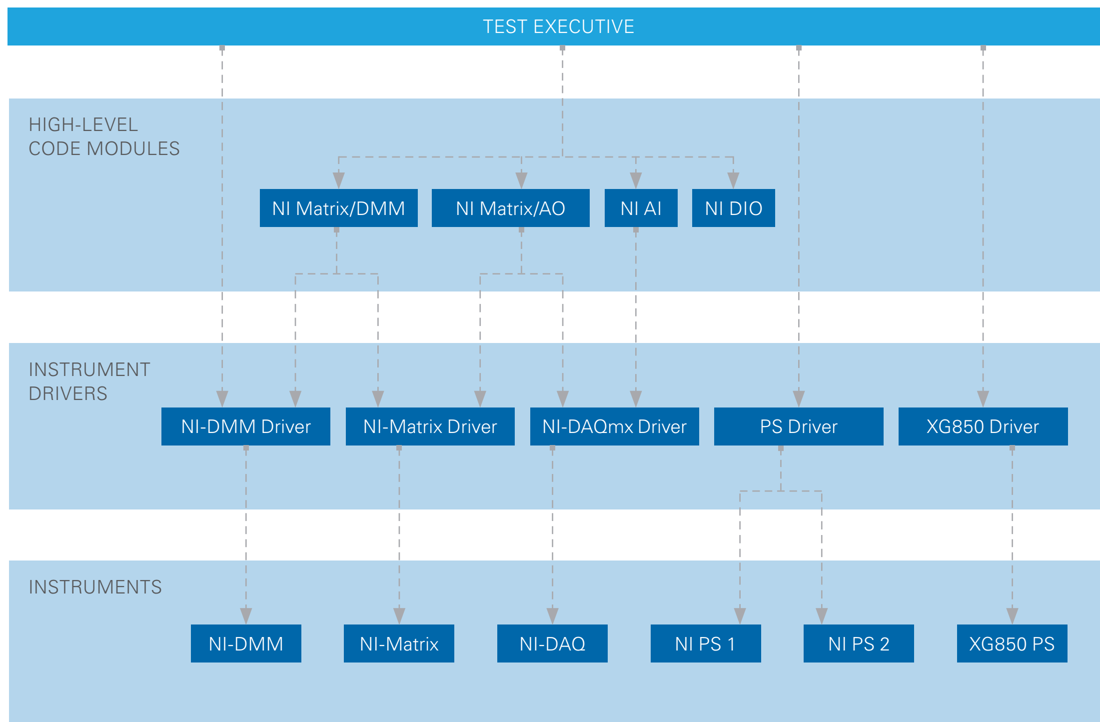


Figure 10. Overview of an Instrument-Specific Driver Method for Automated Test Software Without Abstraction

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.

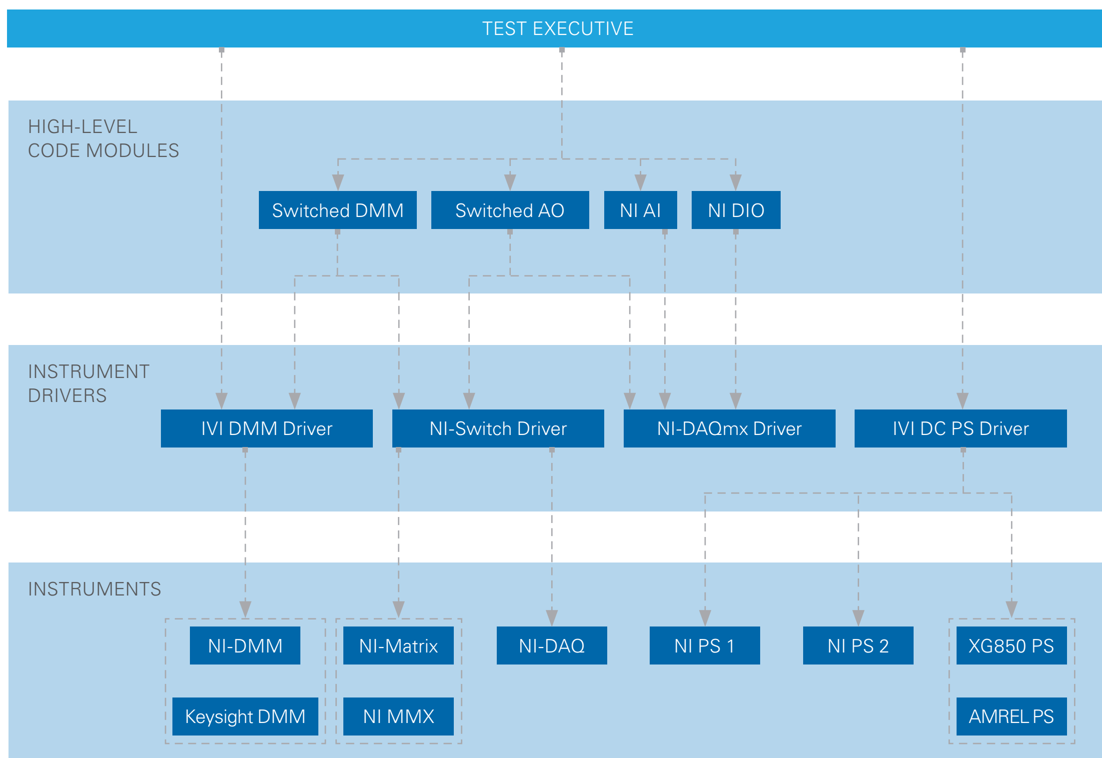


Figure 11. Overview of Automated Test Software With Out-of-the-Box Abstraction

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.

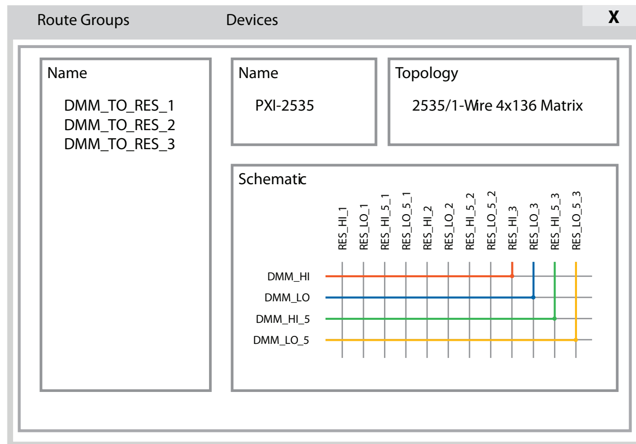


Figure 12. Switch Executive MAL Configuration Interface

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.

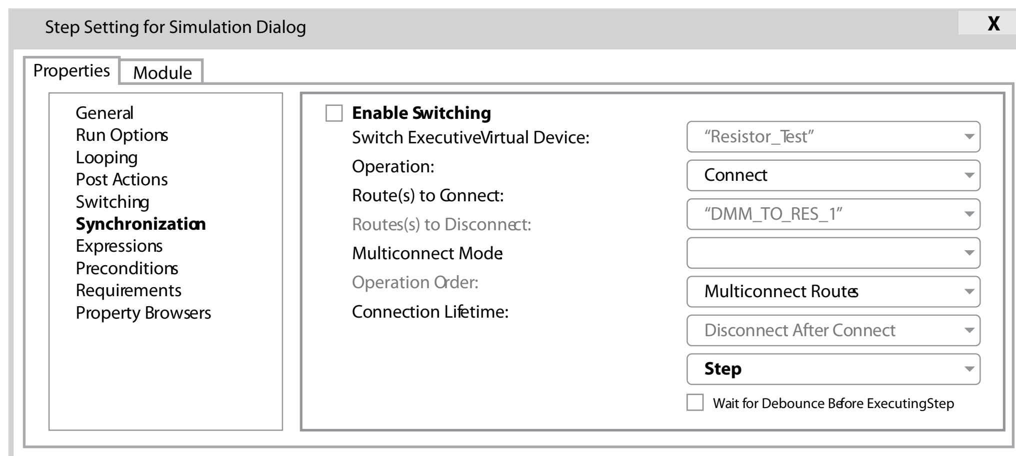


Figure 13. Switch Executive MAL Test Setup

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.

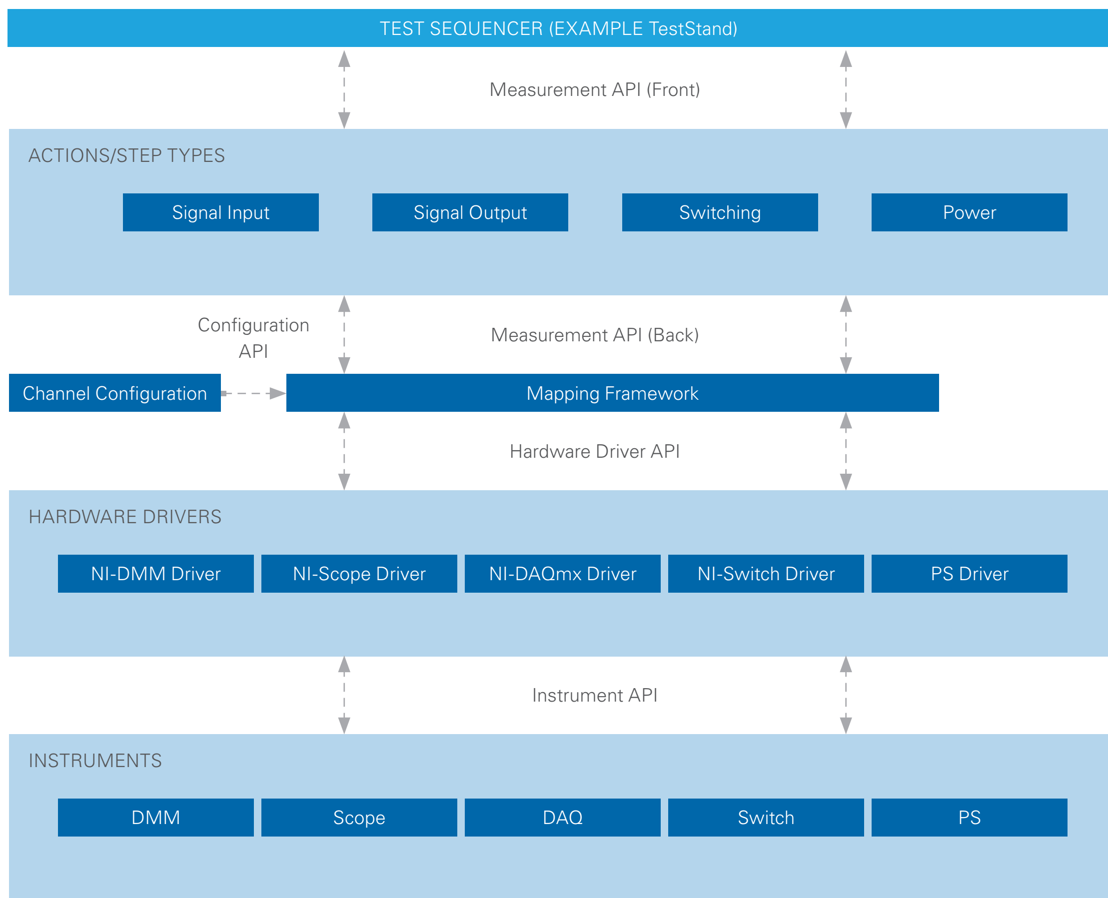


Figure 14. Overview of Automated Test Software With an Integrated MAL and HAL

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.

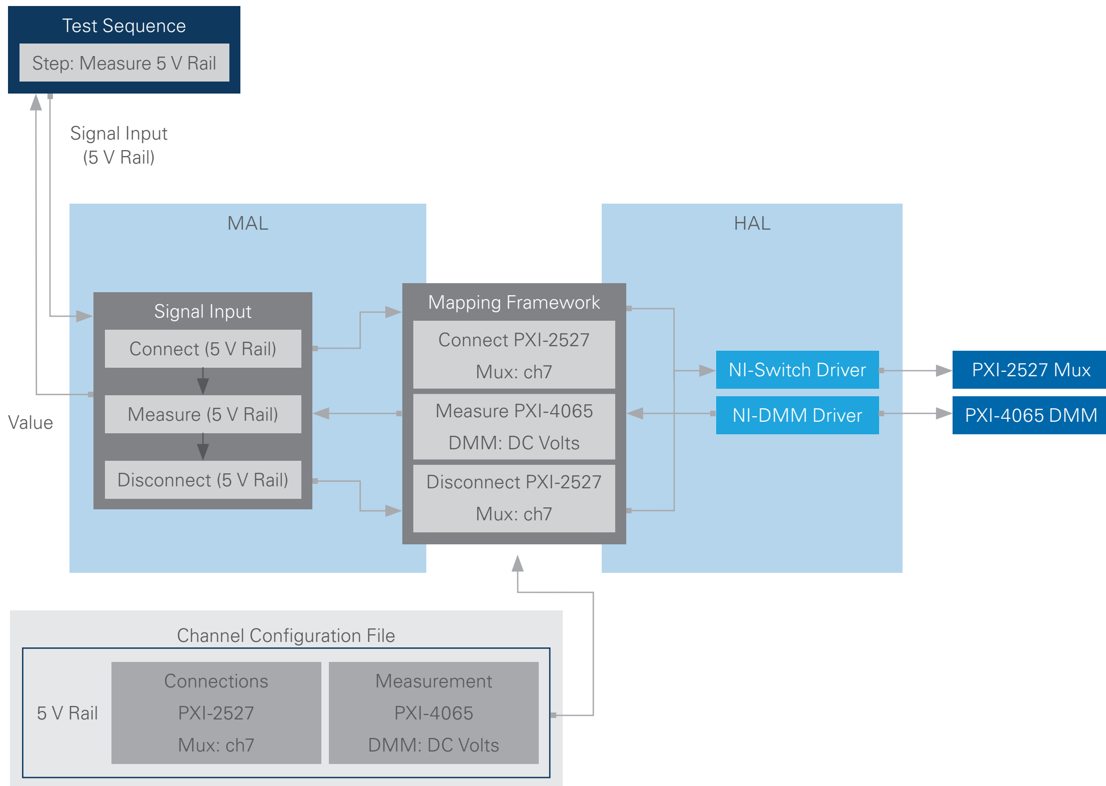


Figure 15. Multiplexed DMM Measurement With an Abstraction Framework

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.

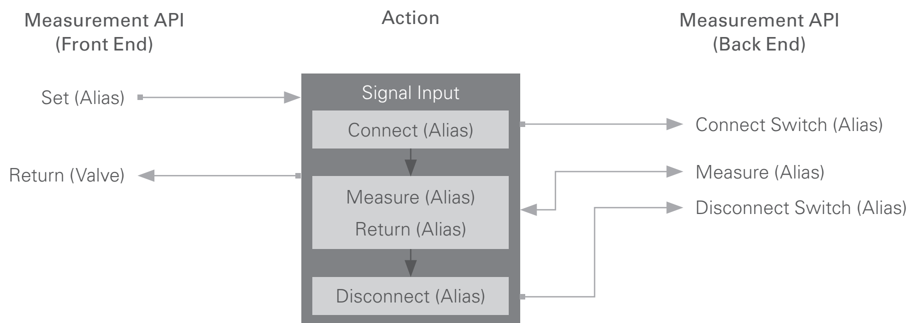


Figure 16. Example of MAL Action APIs for a Signal Input

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.

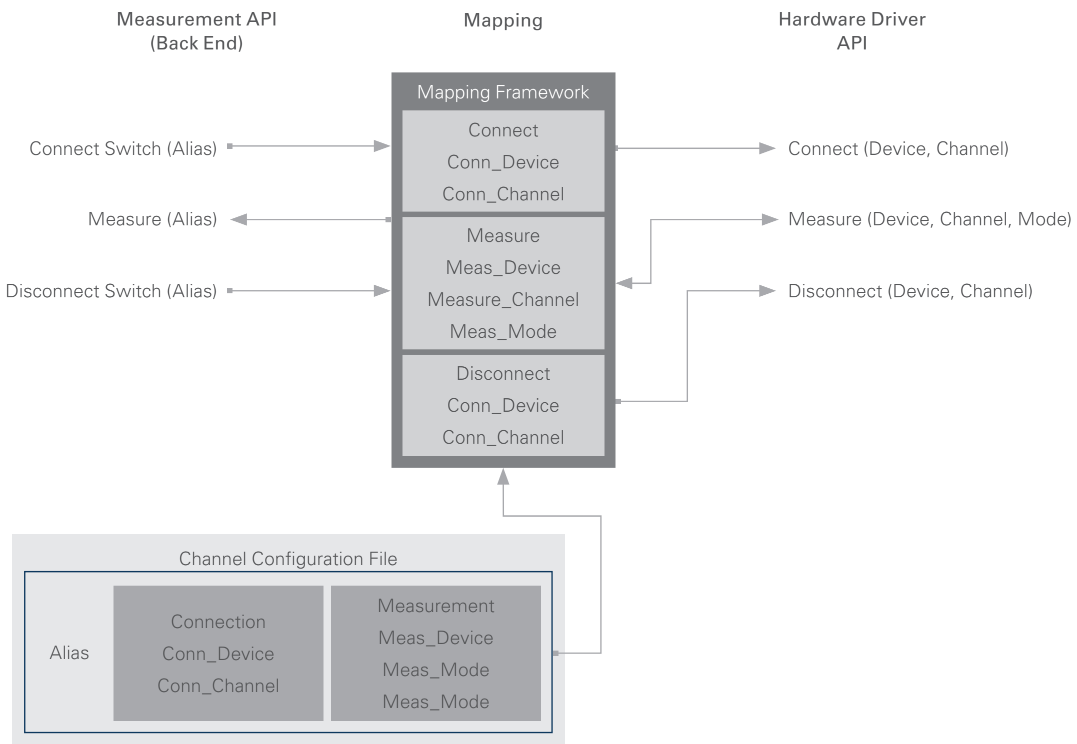


Figure 17. Example of Mapping Framework APIs for a Signal Input

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.

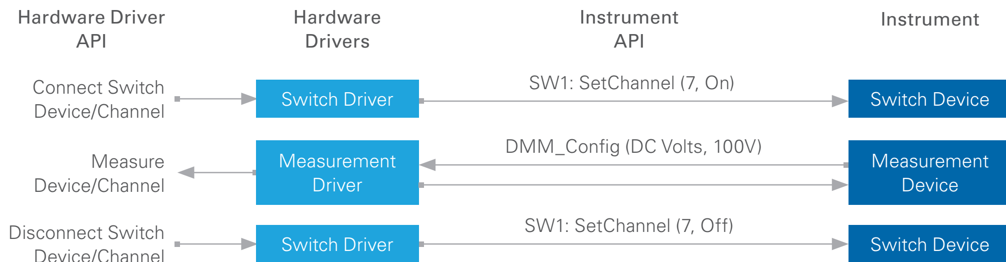


Figure 18. Example of Hardware Driver APIs for a Signal Input

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:

- 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?

No

Use instrument specific drivers.

Yes

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

©2016 National Instruments. All rights reserved. LabVIEW, National Instruments, NI, NI TestStand, NI VeriStand, and ni.com are trademarks of National Instruments. Other product and company names listed are trademarks or trade names of their respective companies.

NATIONAL INSTRUMENTS

FUNDAMENTALS OF BUILDING A TEST SYSTEM

Rack Layout and Thermal Profiling

CONTENTS

Introduction

Importance of Thermals in Rack Designs

Design Approach

Modeling and Validation

Applying Design Criteria to Product

Next Steps

Introduction

Every minute of every day, new projects are placed on the desks of test engineers with the expectation they develop measurement systems that not only meet specification requirements and release deadlines but also offer high quality and reliability. In an ideal world, engineers have the time and resources to perform in-depth research, modeling, and simulation to produce perfect systems. Unfortunately, real-world project schedules do not typically permit the time and resources to develop perfect systems. In a System Integration Study performed by Control Engineering in August of 2014, only 67 percent of system projects were completed on time and within budget. In a world of tight release schedules and demanding project timelines, it is important to consider the aspects of a measurement system that can impact the quality of measurements, which, in turn, can increase risk to schedule, cost, and performance. These aspects range from the instrumentation selected to the quality of the connections and cables to the implementation of the measurement methodology. An overlooked area, however, is the impact thermals can have on measurement quality and measurement system reliability.

This paper equips you with the knowledge to learn more about your design to avoid risk. Learn how thermals impact measurement quality, see basic design approaches, and explore thermal modeling tools for designing a rack measurement system.

Importance of Thermals in Rack Designs

In a generic measurement system, thermals can develop in many ways; however, in a rack-mount measurement system, heat generation within the rack and heat exchange to the environment around the rack are the primary sources of thermal changes that can impact a measurement. You should be concerned with thermals for several reasons:

- Good design practices—Being aware of the implications of thermals and designing your system to account for them is a good design practice. By knowing how thermals may impact your system, you won't allow thermals to become a major contributing variable in your measurements. Keep in mind that operating instruments outside of their specified temperature ranges may have an impact on the quality and life expectancy of that instrument, which is also a reason to maintain good thermal designs for your equipment.
- System uncertainty—Thermals will always exist and are difficult to eliminate completely. Therefore, by better understanding what they are, you are better positioned to account for them in your system uncertainty and can more accurately account for them in your measurement derivations and measurement results.
- System stability—Stability is important for a good measurement system. If variability is observed, often it is difficult to determine the root cause and/or how to address it. Thermal changes in a system can lead to false results in testing because of this variability. Minimize this risk by controlling the thermals in your system.
- Product quality—Products require certain thermal environments to ensure optimal performance, specifically during adjustments. Minimizing the impact of system thermals on product performance can improve the overall product quality.

Thermals and Instrumentation

Thermals should receive significant consideration with respect to instrumentation. Instruments specify certain temperature adherence requirements to meet specifications. Most instruments experience temperature drift, and measurement results will vary if the temperature is unstable or is beyond the adherence requirements. To truly understand and trust the measurement results from a test solution, you should understand and know this impact.

For example, take a look at some related industries such as telecom and IT that have developed best practices for recommended and allowable temperature ranges, which most device manufacturers follow. Some devices still have their own specifications, so the design objective here is to meet those individual device specifications as well as the industry best practices.The primary concerns in these industries include long-term reliability, system uptime, and a lower total cost of ownership (TCO), which are very relevant to automated test.Likewise, automated test engineers should also consider the potential impacts that relate to rack systems and thermals.

The impacts of thermal mismanagement in these industries all tie to a higher cost of operation. For example, if the cooling system fails, the rising temperatures put stress on the rest of the system, resulting in reduced equipment lifetime. If the temperature is too high, IT systems can experience computation errors at the CPU level, resulting in application errors. Redundant cooling systems can be implemented, but increase the TCO. Most importantly, downtime because of auto-shutdown results in loss of service and any downtime translates to loss of money.

National and International Standards

In regard to national standards, Network Equipment-Building System (NEBS) and the American Society for Heating, Refrigerating, and Air-Conditioning Engineers (ASHRAE), have established guidelines and best practices for telecom and IT equipment, respectively.

Whereas ASHRAE is an organization that focuses on best practices across a breadth of areas, NEBS is a more focused effort, specifically for telecom equipment. ASHRAE may reference NEBS for some of its guidelines related to enclosures and rack-mount equipment, but ASHRAE best practices appear to be more comprehensive for all aspects of the enclosure designs.

Although these national standards are not directly applicable to test and measurement certification, they follow many of the same principles and guidelines for rack design and performance that are relevant for automated test.

International standards that the International Electrotechnical Commission (IEC) creates relate to the thermal aspects of enclosures. Manufacturers of enclosures mostly refer to these for either designing or testing the enclosures or for providing usage guidance to customers.

- IEC 61587-1 specifies environmental, testing, and safety requirements for empty enclosures (that is, cabinets, racks, subracks, and chassis) in indoor conditions.
- IEC 62194-1 provides methods for evaluating the thermal performance of empty enclosures under indoor and outdoor conditions.

The design objectives for the telecom and IT industries are similar to the test and measurement industry, but the primary focus areas and challenges are somewhat different. Design objectives in test and measurement focus more on meeting individual device specifications, because there is no universal standard for test and measurement equipment racks, though best practices from various companies in the industry exist. The primary focus is to ensure that each instrument, as well as the device under test (DUT), maintains alignment to its specifications. This comes even before long-term reliability or uptime, because those types of considerations become more relevant at much higher temperatures, while loss of required accuracy can occur at relatively lower temperatures.

In terms of challenges, automated test systems have some added constraints. One is that test racks are usually used in environments occupied by moving humans or in uncontrolled production facilities all around the world. This does add randomness in the overall thermal profile of the room/location, unlike unoccupied server rooms with stationary objects.

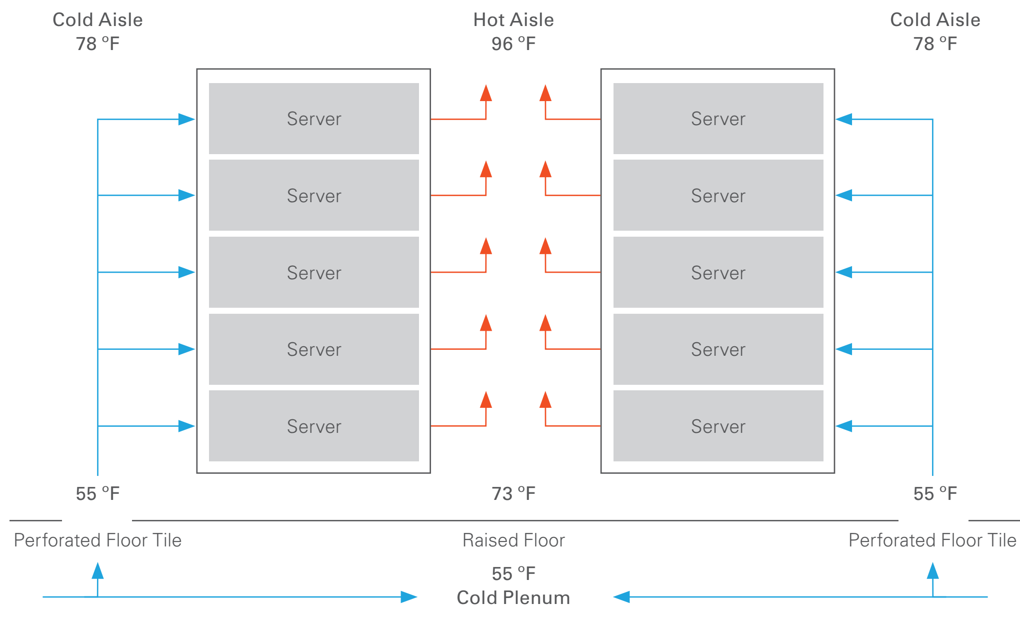


Figure 1. Server rooms/facilities have controlled environments, unlike test systems, which are placed into uncontrolled, chaotic environments.

Impact of Thermals on the DUT, Test System, or Test Results

The first and most fundamental impact is on the accuracy of your instruments and DUTs. If you can't ensure the correct ambient temperature for any of your instruments, its accuracy would have to be derated.

For example, your calibration could be invalid if the ambient temperature changed enough between adjustment and verification to change the instrument or DUT accuracy. This might also result in false failures or false passes in manufacturing. The best way to manage this risk is to account for thermal offsets in your measurement uncertainty calculations.

Even if the temperature is within the specified range of operation, you might notice some difference in data from different test stations. The same is true for data collected in development rather than data collected in a production environment, with changes in ambient temperatures around the test station being the primary cause for variations.

Most instruments follow the ambient temperature closely, albeit with an offset. This means that even a slight change in ambient temperature can translate into a change in the instrument's temperature, which creates a rising potential for variation in data across test stations.

As a difference in temperature may result in data variation across development and production, it can also happen across verification and validation and test development. Usually, verification and validation is performed on a benchtop setup in an office setting, whereas test development is done using a rack-mounted test station in a controlled environment. This results in a totally different environment for the instruments, even if the instruments and subsequent test system are the same. Some instruments even have alternate measurement specifications for certain temperature ranges, so it's important you use the applicable specifications in your design and measurement calculations.

In addition, none of the previously discussed environments perfectly model the production manufacturing environment, so being conservative in your design to address the potential of these environments is recommended if no known environmental information can be applied. For example, the figure below compares environments by looking at room ambient temperature data over a period of a few hours at the front-top of a test station in an office cube area, as well as in a controlled environment.

The controlled environment is a small room with dedicated air conditioning, so you see the thermostat turning on and off more clearly with sharper temperature changes; this temperature is maintained around 23 °C to 25 °C. The office cube area is slightly warmer, though more stable.

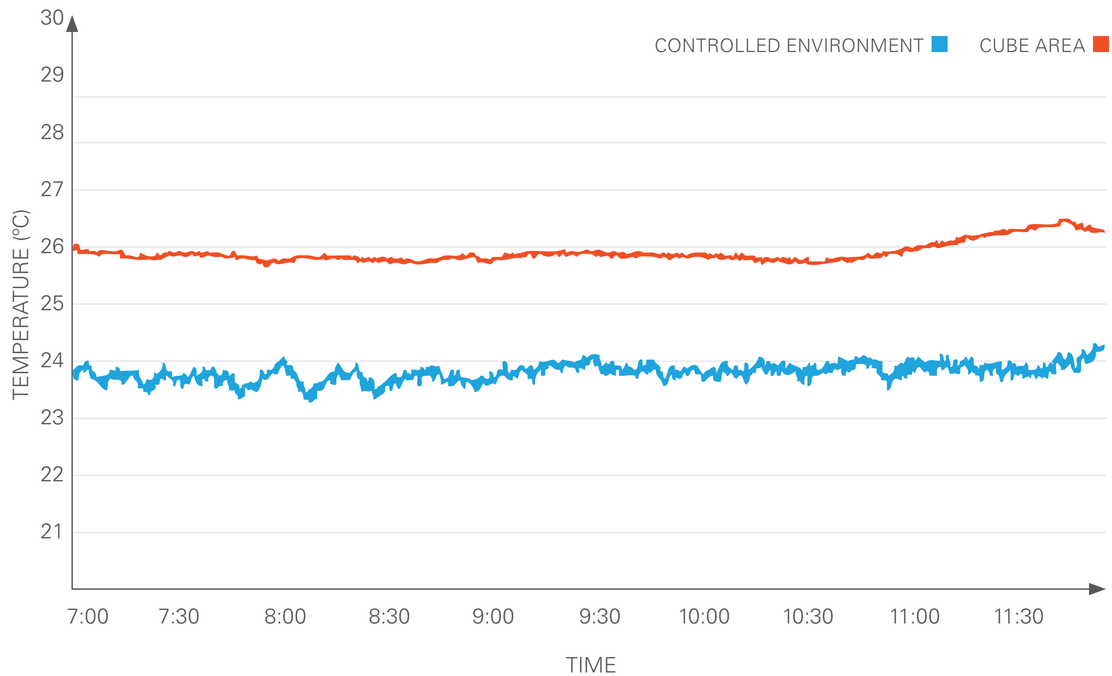


Figure 2. Room Ambient Temperature

The slight increase in both temperatures is the start of a workday (far right of chart); when people arrive, the temperature increases from body heat and doors opening. Note that the temperature for office cube areas can vary with time of year, location, floor, and other factors. In contrast, the controlled environment temperature is fairly constant throughout the year because of a dedicated air conditioner. In light of all these facts, you should always keep track of the ambient temperature while conducting verification and validation or developing tests and collecting data. This helps in data analysis if differences are seen across verification and validation data and test data.

Thermal Profile of Rack-Mounted Systems

When thinking about the heat distribution in any system, there is a temptation to oversimplify it. The most common simplification involves the perspective of "cold at the bottom, hot at the top." You might also think that the temperature gradient is uniform across the rack, but in most cases, these simplifications are not exactly true.

Figure 3. The common assumption of an even distribution of cold at bottom to hot at top will deliver bad results.

In a real system, a number of variables contribute to the thermal profile, and as such, the thermal distribution varies. If an infrared thermometer gun or thermocouples were applied appropriately to a system, you would see characteristics like local heat zones and nonuniform temperature gradients in the horizontal or vertical axis of the rack system. This is because you are not dealing with just hot and cold air in isolation of everything else; the thermal profile depends on the rack layout, fan speeds of individual devices, location of inlet and outlet vents, power dissipation of each device in the system, and the airflow forced by the combination of all fans in the system.

This is important because it means that the top of the rack may not necessarily be the point needing the most attention, which is directly counterintuitive to one of the most common oversimplifications regarding thermals. A thorough evaluation is required to understand the unique thermal profile of each system and address areas of concern.

The following example helps explain how thermals can behave in a typical rack-mounted test system.

Figure 4. Localized heat zones create a variety of temerpatures throughout the test rack.

The first things noticed are the localized heat zones, which depend on the system's rack layout and device specifications as well as usage. In this example test station, the power supply has the warmest air around it, followed by the PXI chassis. Other parts of the test station appear to be colder than these localized heat zones, so depending on each system, there might be a need to address these localized heat zones differently. Another thing to notice is that the bottom heat zone is nonuniformly distributed in the x-axis.

What Causes Thermal Nonuniformity?

Usually, this nonuniformity is because of the device airflow patterns. For example, the power supply is composed of individual power supply units and a power supply mainframe. As you can see, the mainframe has its airflow from the left to the right. That means most of the warm air is on the right-hand side of the instrument.

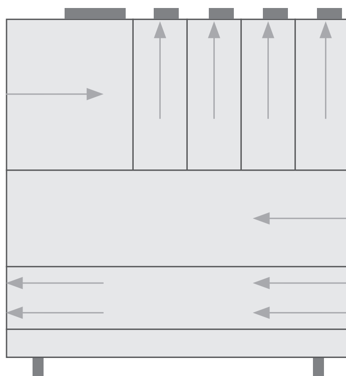


Figure 5. Top View of a Modular Power Supply With Airflow Direction Indicated

The exhausts at the rear end are from individual modules; not all of them might be exercised at the same time, so the rear might not usually be as warm as this side. Moreover, the thermal profile of a rack system changes with usage, so the characterization is more involved than simply looking at the temperatures of a given case.

How Should the Thermal Profile of a System Look?

There are a few aspects of the rack-mounted system that you need to understand to determine how the thermal profile should look. To begin, you need to understand the unique needs of each system:

- What are the required system-level specifications?
- What environment will the system be operating in?
- What instrumentation will be used and what are the temperature requirements for those instruments?
- Is keeping the temperature within a range enough or does your application require the temperature to be stable as well?

For example, if your DUTs are PXI modules and you need to power the DUT PXI chassis on and off while switching DUTs, the thermal profile of your rack would change repeatedly. These repeated changes require awareness of any instability in the rack's thermal distribution.

Lastly, not all points inside the rack are necessarily required to maintain the same temperature. It is typical to have some areas warmer than others, as long as the inlets of all instruments are drawing air that is in the specified temperature range.

Design Approach

This next section highlights best practices for developing a rack-mount system from design through rollout.

Before starting a rack design, you should understand several key elements about the instruments you are using that will have an overall impact on the design:

- **Evaluate the Air Inlet and Outlet for the Device**
  First, investigate your instruments and understand where the air inlets and outlets are on the module. The ability to provide the temperature requirements for the device is heavily dependent on the temperature at the air inlets and where you exhaust the heat generated from the instruments. Having an understanding of this will help you to successfully map out a rack design.
- **Understand the Specifications and Temperature Requirements**
  Often, devices specify certain storage, operating, and calibration temperatures. Which ones do you care about and how do you interpret each of them? Understand the way in which each instrument specifies temperature and what the impact is on the instrument performance or specifications—specifically, those specifications that impact warranted performance. For example, the 3458A states that you must maintain 23 ˚C ± 5 ˚C of ambient temperature to warranty the specifications of the device. Further, it states that you must autocalibrate the device if you have a relative change of ± 1 ˚C from the last autocalibration. The first specification is absolute to ambient, and the second is relative to the last calibration. Understand the differences and how that might impact your solution.

- **Understand the Definition of Ambient for the Device**
  Most traditional box instruments define ambient temperature as the temperature of air in the environment that surrounds the instrument. Typically, for PXI products and chassis, ambient temperature is defined as the temperature of the air immediately outside the fan inlet vents of the chassis. Because chassis like PXI-1045 require a clearance of about 1.75 inches for correct flow of air, you can safely assume that measuring the temperature within this clearance close to the inlet fans would give you the ambient temperature you care about.

  A common mistake is to assume that the ambient temperature for your instrument is the same as the temperature in the room where you are using the instrument. In general, this may hold true if you are using your instrument on the desktop with no influencing heat sources in close proximity; however, in a rack-mount design, you must consider the localized air temperature inside of the rack as the ambient temperature for your instrument. Instruments inside a rack design are more susceptible to thermal issues.

  Depending on your application and use of the instrument, be sure to accurately understand what your ambient temperature for each device may be.

Before selecting your rack or getting into any other specifics of the rack design, understand the expected thermal load that your rack may experience. This is easily done by performing a power budget of all of the electronics planned for the system. An understanding of power consumption provides insight into the thermal load.

Power Budget for All Electronics

Consider all instruments and peripherals in the design. This can include measurement devices, PCs, monitors, battery backups, or anything that may be a heat-generating source within your rack. For these devices, reference the product specifications to determine the power consumption of each. In general, product specifications list the worst-case power consumption (under full use or full load), which often isn't representative of the general or average performance of the device over time. Often, 60 percent rated max power consumption is used as a general guide for design. Having said that, in the future you may use your rack design for other purposes, which may result in an increase in thermal load, so consider adding some guard-banding to your calculations as you see fit.

Ideally, if you can measure the actual power consumption of your devices ahead of time, this gives the best outlook of the overall power consumption; however, this may not be feasible while planning your system. As a best practice, come back after the system is designed and make these measurements for documentation purposes.

Temperature Requirements and Airflow Profile

Based on instrument temperature requirements and airflow profile, map out the general locations wanted for instruments. Heat rises, so, often, the rack is cooler toward the bottom and warmer toward the top. Plan to place your most sensitive instruments toward the bottom of your rack design. You can use techniques to establish an acceptable thermal environment for your instruments in other rack locations, but often that comes at a cost.

Usability may be a constraint that drives the placement of some of your instruments, but evaluate and understand the impact of that placement. Maybe it requires additional consideration of how to handle airflow or how to provide cooler air to a device intake that is somewhat unconventional. Keep that in mind as you continue with your design. Also, account for any clearance constraints specified by the instruments. Often, instruments specify a certain distance that must be maintained around the device or in proximity to its inlets and outlets. Be sure that these specifications are met.

Rack Size Based on Layout

The layout should give you an idea of the size of the rack required to house the instruments. Remember to factor external constraints, such as floor space and room height, into your rack selection.

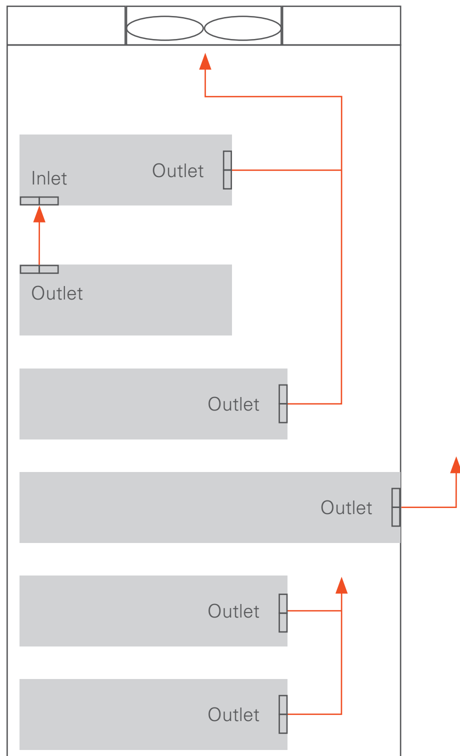


Figure 6. Example of Instrumentation Blocking Exhaust in a Rack System

The exhaust air from all instruments should have an unobstructed pathway to exit the rack. In this example layout, you can see a whole instrument blocking the exhaust from instruments below it. Another bad practice is to have the hot and cold air short circuited. You can fix these by laying out the rack more cleverly, but start with a good understanding of the instruments' inlets and outlets and expected airflow.

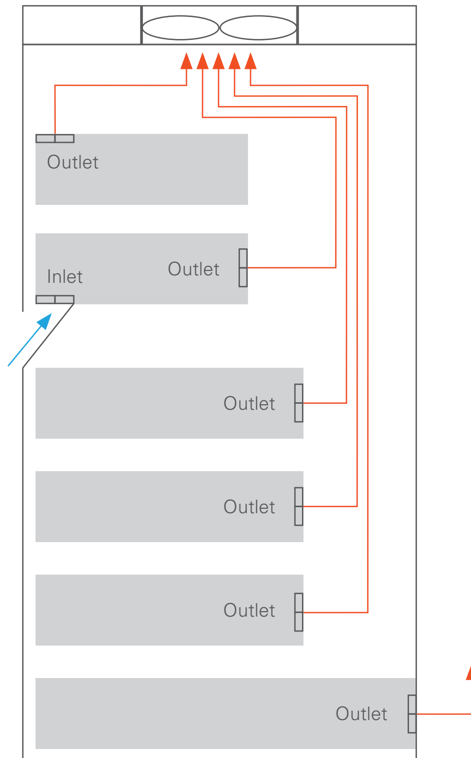


Figure 7. Example of Proper Exhaust Layout in a Rack System

Rearrange the instruments to provide a continuous airflow path from all instrument outlets. In addition, use mechanical separations to ensure that all instruments are getting inlet air from outside air when possible.

For situations where the air inlet of an instrument is located on the inside of the rack, because of how the instrument is installed, the exhaust from one instrument may be recycled into the inlet of another instrument. You could have multiple instrument exhausts feeding into each other's inlets. This could raise the ambient temperature inside the rack significantly, and would increase as you go up the rack.

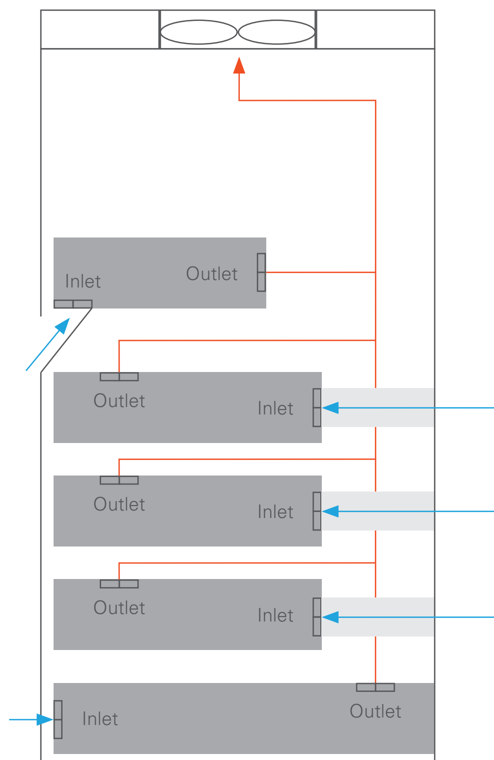


Figure 8. Example of Custom Inlet Venting in Rack Design

In these situations, the ideal approach is to provide an isolated path from outside the rack to the inlet of the instruments. This ensures that the ambient temperature for an instrument is understood and controlled.

Remember, it's not enough to look at only the inlet temperatures. Be sure to provide adequate clearance for every instrument per the instrument's specifications so that it has proper insulation and airflow around it. It may be tempting to ignore these constraints because often it results in quite a bit of unused or wasted space. To get the specified performance from of an instrument, however, these clearance specifications must be followed.

From the image, it may appear that you are creating localized heat pockets between the instruments. Keep in mind that the red heat arrows are shown just to illustrate that this heat would be blocked from entering the air inlet. Design your isolations paths to the inlets of your devices to allow air to flow around and up the rack. Most rack assemblies provide adequate spacing to the sides of all instruments to ensure that the appropriate "chimney effect" can be established. Any heat that your instruments exhaust should also be allowed to flow around your inlet isolation barriers. There are many ways to ensure that the heat is properly extracted from the rack without impacting your instruments. These are just a few examples.

Heat Transfer and Airflow

As you saw earlier, most instruments' outlet temperatures follow the ambient temperature trend. The difference or offset comes from self-heating and air heating:

- Self-heating—Any component on an electronic device will heat up above the ambient temperature because of warm air coming from other components as well as self-heating. Self-heating is not in your control.
- Air heating—A cleverly laid out rack system can minimize air heating and a correctly designed rack cooling system or room cooling system can take care of ambient heating. So, control this offset while designing your system.

In this case, the two chassis have slightly different self-heating and air heating because of different PXI cards installed and different locations in the rack.

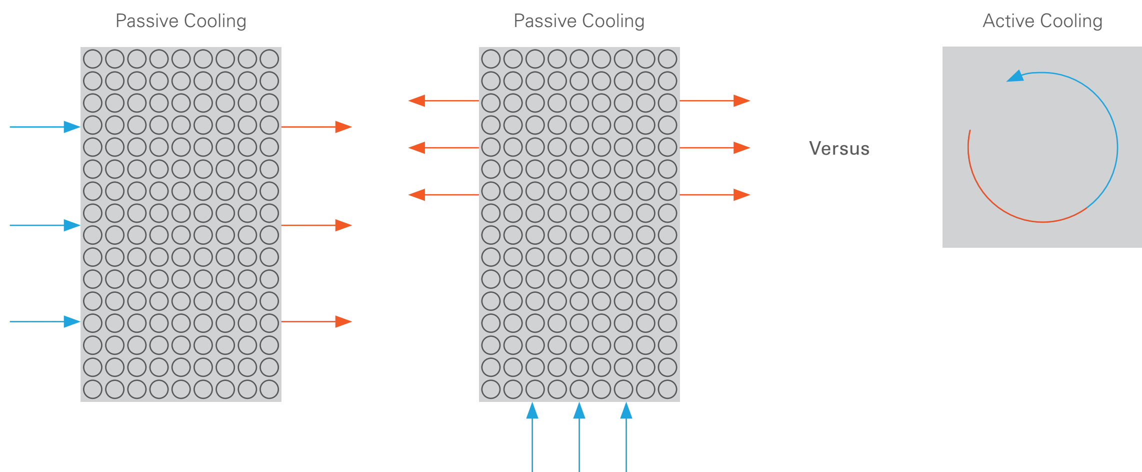


Figure 9. Passive cooling relies on the fans of the internally mounted instrumentation, whereas active cooling uses auxilary fans and blowers mounted in the rack.

Passive Cooling

Passive cabinets are designed to maximize the ability of the internally mounted equipment to cool itself through its own fans. In this method, the equipment produces airflows, and the surfaces and ventilation in the rack exchange heat.

Active Cooling

Whereas passive cooling simply relies on the equipment fans and heat transfer, active cabinets use additional, strategically placed fans and\or blowers to supplement airflow, thus increasing heat dissipation.

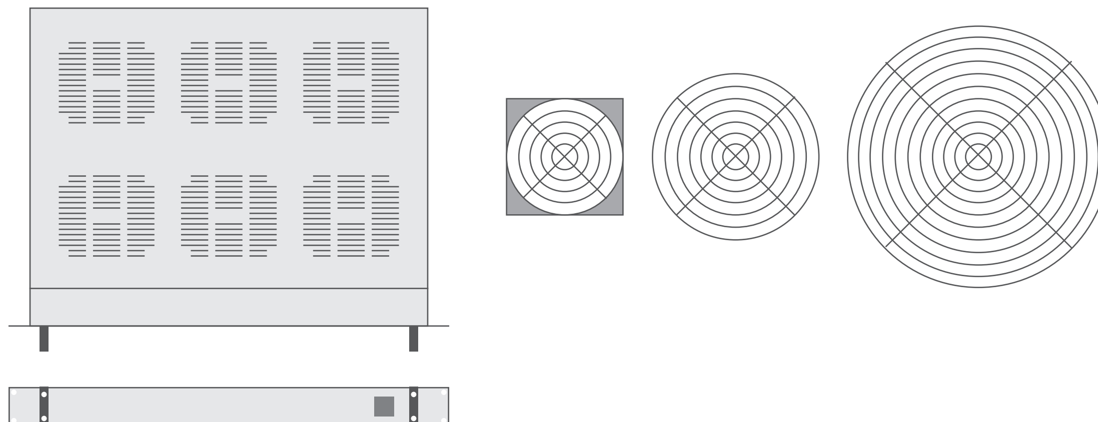


Figure 10. Active cooling options range from internal trays to individual side or top-mounted fans.

More often than not, forced-air cooling is needed for rack-mounted test systems. Having one, properly sized exhaust fan at the top of the rack is typical; however, the vents and airflow need to be planned according to the exhaust fan location. Putting a fan tray in the middle of the rack is recommended to help with airflow if your air path has bends, obstructions, or maybe a concentration of high-power instruments in one area. You may also consider localized heat removal if your rack has hotspots that you need cooled. You can use individual fans or fan trays for this purpose.

Fan Capacity

To get a good idea about how large your fan should be, you need to calculate the cubic feet per minute (CFM) of air the fan should be able to move with consideration given to the total power wattage of all devices in the rack and the temperature difference between the air inside and outside the rack.

Higher CFM comes with some trade-offs like cost, vibration, and acoustic noise. Also, however high your fan's CFM is, you can't cool your rack below the room ambient anyway. So the idea of this equation is to arrive at a good balancing power between all these parameters. Air resistance also plays a part in the ability of a fan to cool. Air resistance increases based on the cross-sectional area of objects that are in the path of the airflow, whether it be the area of opening for intake area or area of devices in the path of the airflow. Therefore, margin should be given for the CFM rating of the fan to ensure it can overcome air resistance and still provide the necessary air movement.

When evaluating the total wattage, avoid using the rated power of the devices; this is usually the maximum possible power that a device can dissipate, but rarely does a device use this much power consistently. A good rule is to use something between 50 to 60 percent of the rated power. Or better yet, use the PDU of your rack system to get the actual power being delivered and use that value.

Delta T (ΔT_c) equates to the amount of heat you want carried away from the rack. This comes from looking at the requirements of your devices or by looking at simulation results. It might vary, depending on the location inside the rack—usually higher at instrument outlets or at the top of the rack, so be sure to understand what the appropriate delta T is for your system, and how to confirm you are achieving that delta T in practice.

Modeling and Validation

If a system is costly, includes long lead times for components, is part of a critical or strategic application, or contains many unknowns, modeling should be part of the design process.

Modeling the Rack Design

Apart from theoretical calculations, modeling and simulation of your rack system can greatly speed up your design optimization and provide you with solid feedback on what changes to make.

Enter as many design specifications as possible into the software to get the most accurate modeling. If you cannot locate a necessary specification, evaluate similar components/ instrumentation and ask senior members of your team if they have experience with these devices to gather estimates. The fewer unknowns you have, the better your modeling will be.

When you have exhausted research on your design, either through evaluating the instrument temperature requirements, performing calculations to optimize airflow and temperature, or simulating the design, you have done as much as possible short of performing real-world qualification testing. At this point, complete the fabrication of your design and validate the performance.

To characterize the performance, use temperature sensors or thermal imaging cameras. Focus on the critical areas within the rack that are of importance, such as the air intake of the rack and the air intakes of the instruments. Collect temperature data across your rack design while powering the rack and exercising the instruments in a general fashion as they may normally be used for the products to be testing. This gives you the most realistic view of how temperature will behave.

You may also want to exercise certain worst-case loading conditions, such as when the thermal load may be at its highest or lowest, to ensure that your design can still accommodate these conditions. Consider the time of day of testing, the duration of testing, and test conditions (how many operators are present, what normal interactions someone might have with the station, and so on) as factors that can impact your results. Analyze the results carefully to look for any previously unidentified anomalies or issues that you may need to address.

Validation Methods

First, use standard graphs, equations, and simulations early in the design to gain a general comfort level with the approach. Second, perform system characterization using temperature sensors or thermal imaging to validate the design and iterate on it until wanted requirements are met. Last, perform gage R&R studies to validate stability and performance to ensure station-to-station performance and production test to validation to verification agreement.

Maintainability Through System Monitoring
Consider implementing a health and monitoring system that gives you the ability to evaluate the system in real time to ensure that the expected performance is still being met. This ensures you can at least make informed decisions during testing by having feedback on the system performance, but may also lead to better understanding of measurement data and better prediction of system maintenance.

Areas of focus include:

- Stand-alone system to monitor system performance
- System watchdog for reporting maintenance issues
- Feedback for tests to validate test conditions
- Historical logging for evaluation and trend analysis

Applying Design Criteria to Product

Accounting for Thermals in Specs/Limits
When collecting data using your rack design, do not overlook the effects of thermals when considering specification validation of setting manufacturing test limits. If there are differences between expected results and actual, thermals may be the cause of these differences.

Spec Validation
Know the assumptions used during your spec derivations. If the ambient conditions while collecting real data differ from your derivations, be sure to account for them. Ensure that you are experiencing the performance you expect under the provided temperature conditions.

Limit Calculations
Similar to spec validation, account for temperature differences and variation in your limit derivations. If the product specifications are stated as a certain range and you are testing in an environment that provides a different temperature range, account for the difference by accounting for the temperature coefficient of your device appropriately. For example, NI switch modules are typically specified at 0 to 55 °C, however, the general temperature environment that they are tested under would be the standard manufacturing test floor, which carries on a maintained 24 °C ± 4 °C. Subtract the equivalent, worst-case temperature coefficient from the specifications and uncertainty of your measurement when establishing your test limits.

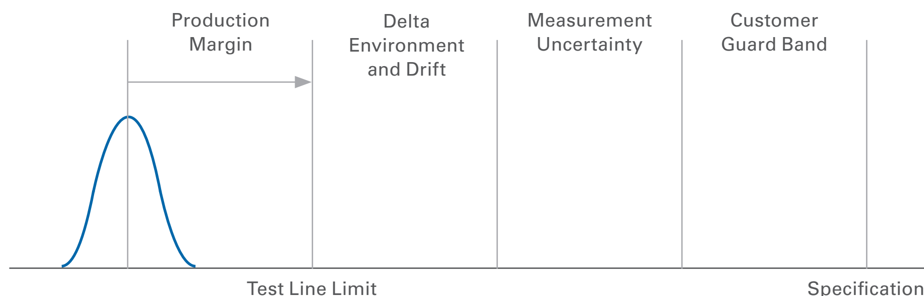


Figure 11. Example General-Purpose Specifications Model

Independent System Monitoring

Monitoring Temperatures in Real Time

Monitoring temperatures in real time gives you the ability to make dynamic decisions during your testing. You may determine that you are violating a certain operating requirement, thus you may halt your testing; or you may determine that you need to perform a self-calibration or institute a delay before continuing testing to account for stability. Lastly, just having the data and logging it historically may provide insight in the future if questions arise about performance or correlation to measurement results.

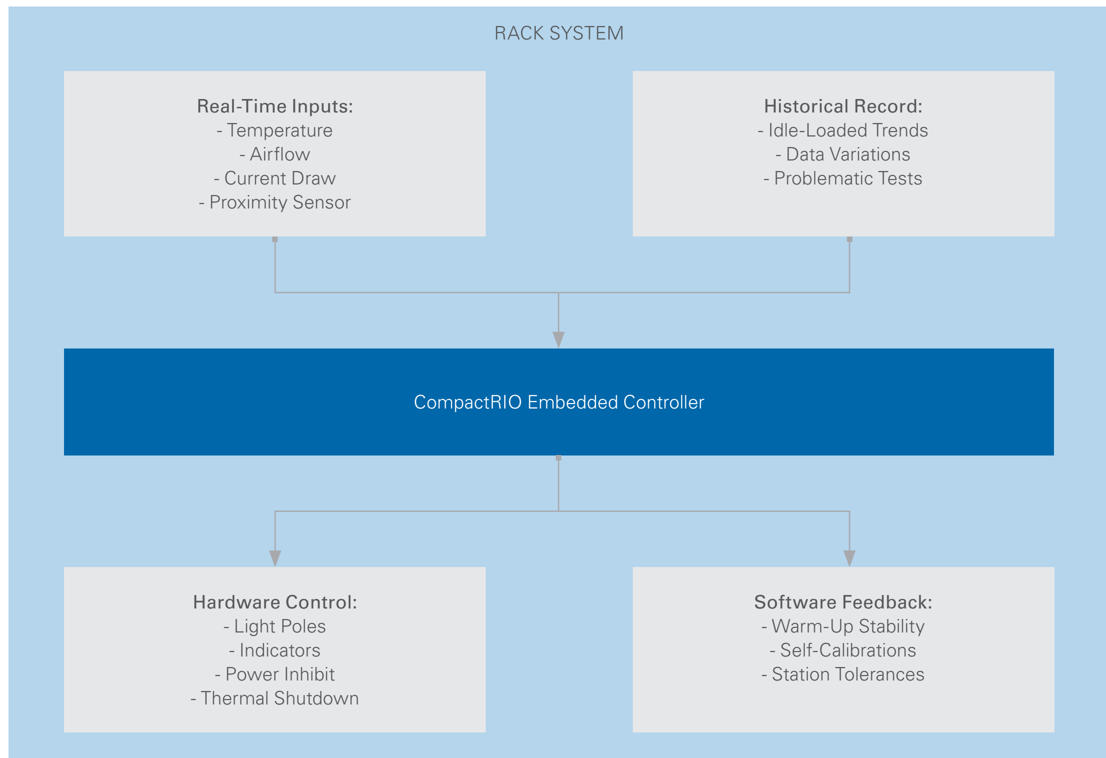


Figure 12. Example Independent Monitoring System Using CompactRIO

For example, you could use an embedded, independent system to monitor and control certain aspects of a test station design. You can monitor temperature and make it available in your test execution to make decisions, as well as monitor and manage resources in the station to support parallel test. In general, you can use an independent monitoring system for several tasks and it can be beneficial during not only the design and validation of your rack system but also the deployment and long-term use.

Although many options exist, you can use an approach like this to:

- Monitor
  - Ambient temperatures throughout the rack
  - Airflow, current draw, and internal temperatures of your instruments
  - Instrument health for maintenance concerns
  - Doors of the racks using proximity sensors to detect if the system has been accessed
  - Temperature conditions to implement thermal shutdown mechanisms to safeguard the system
- Gain data to make real-time decisions in your test application
- Log this historical data for future analysis
- Provide feedback to the user of the station through light poles, indicators, or displays on the status of the station and to report any out-of-tolerance conditions

A health and monitoring system can help you to evaluate the system in real time, make informed decisions during testing, and better understand measurement data and predict system maintenance.

Next Steps

NI Alliance Partner Network

The Alliance Partner Network is a program of more than 950 independent, third-party companies worldwide that provide engineers with complete solutions and high-quality products based on graphical system design. From products and systems to integration, consulting, and training services, NI Alliance Partners are uniquely equipped and skilled to help solve some of the toughest engineering challenges.

Find an Alliance Partner

NI PXI Chassis Cooling

NI chassis are designed and validated to meet or exceed the cooling requirements for the most power-demanding PXI modules. Chassis designed by NI go beyond PXI and PXI Express requirements by providing 30 W and 38.25 W of power and cooling in every peripheral slot for PXI and PXI Express chassis, respectively. This extra power and cooling makes advanced capabilities of high-performance modules, such as digitizers, high-speed digital I/O, and RF modules, possible in applications that may require continuous acquisition or high-speed testing.

Learn more about the NI PXI Chassis Design Advantages

Build Your PXI-Based Test System Today

NI is the creator and leading provider of PXI, the modular instrumentation standard with more than 1,500 products from more than 70 vendors. Select the appropriate chassis, controller, and modules for your application, and let the advisor recommend the necessary components and accessories to complete your system.

Configure your PXI system

©2016 National Instruments. All rights reserved. CompactRIO, LabVIEW, National Instruments, NI, and ni.com are trademarks of National Instruments. Other product and company names listed are trademarks or trade names of their respective companies. A National Instruments Alliance Partner is a business entity independent from National Instruments and has no agency, partnership, or joint-venture relationship with National Instruments.

NATIONAL INSTRUMENTS

FUNDAMENTALS OF BUILDING A TEST SYSTEM

Mass Interconnect and Fixturing

CONTENTS

Introduction

Overview of a Mass Interconnect System

How to Choose a Mass Interconnect System

Overview of a Test Fixture

Fixture Considerations

Next Steps

Introduction

Building a test system without a plan for how you will connect your instrumentation to your device under test (DUT) is similar to trying to drive your car without wheels. Your car may have best-in-class horsepower and Italian leather seats, but you aren't going to reach your destination without wheels. Mass interconnects and test fixtures are where the rubber meets the road for automated test systems. After determining your instrumentation, the number of switches you need, and the location that your switches will reside in the test system, the next step is to choose a suitable mass interconnect system and design an appropriate fixture that seamlessly mates your DUTs to the rest of the system.

Overview of a Mass Interconnect System

A mass interconnect system is a mechanical interconnect designed to easily facilitate the connection of a large number of signals either coming from or going to a DUT or DUT fixture. Rather than connect each signal one by one, a mass interconnect system connects and disconnects all signals at once. For automated test systems, a mass interconnect system usually entails some interchangeable mechanical enclosure through which all signals are routed from instruments, typically in a test rack, to the DUT, making it easy to quickly change DUTs or to protect the cable connections on the front of the instruments from repeated connect and disconnect cycles.

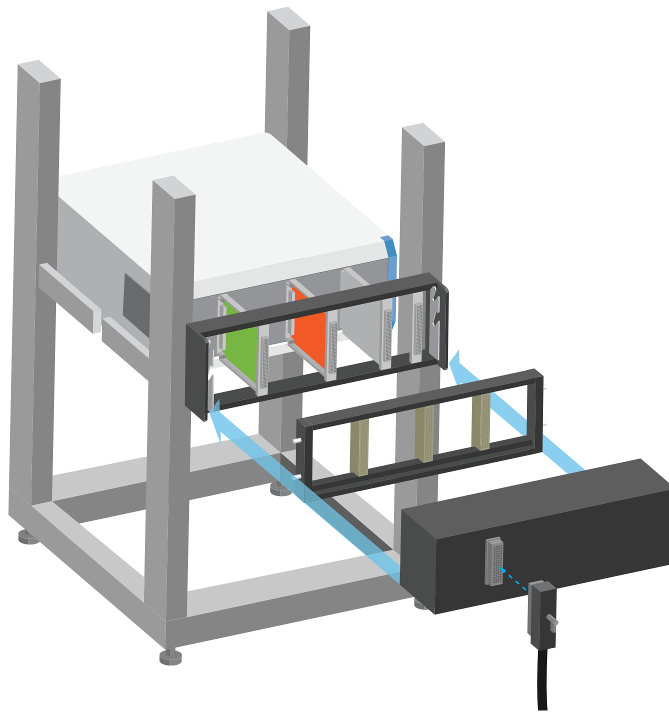


Figure 1. A mass interconnect system simultaneously connects a large number of signals between your test system and DUT, providing an easy way to reuse common test equipment with multiple test fixtures for different DUTs.

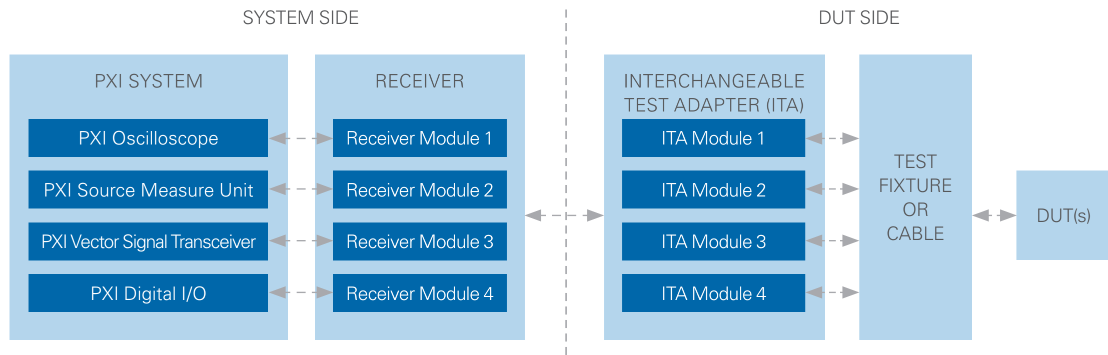


Figure 2. A mass interconnect system can be simplified into two parts. The system-side components, often referred to as the receiver, connect instrumentation to the mass interconnect, acting as the "socket" for the ITA. Alternatively, the DUT-side components, often referred to as the ITA, connect the DUT to the mass interconnect, acting as the "plug" for the receiver. The receiver and ITA mate together with a single mechanical action, providing an easy way to reuse a common set of test hardware with various DUTs.

System-Side Components

The system-side components comprise everything between the instrumentation and the mass interconnect. System-side components are part of the common components that stay with the test system, even if you change the interchangeable test adapter (ITA) and fixture to interface with various DUTs. Below is a brief explanation of each of the system-side components.

Receiver

The receiver is the core component of the test system side of the mass interconnect. It is the mechanism that provides the ability to connect multiple instruments simultaneously to the DUT. The receiver system includes the frame, mounting hardware, receiver modules, and the connections from receiver modules to instrumentation.

Mounting Hardware

Mounting hardware holds the receiver on the front of the rack or PXI chassis. It is typically mounted on the front side of a 19-inch rack, providing easy access for the test operator. Sometimes, the mounting hardware has a hinge, or is mounted on slides, for easy access to the instruments or cables behind the receiver.

Receiver Modules

Receiver modules are mounted into the receiver so that all appropriate connections can be made from instrumentation to the receiver's main connector, a type of standard connector that is connected to the DUT side of the mass interconnect system, along with all other receiver modules, in one mechanical action. The connections are routed from the instruments and other auxiliary equipment to the appropriate contacts within the receiver modules. These are typically specified according to density, bandwidth, current, or other special-purpose requirement of the signals being passed through them.

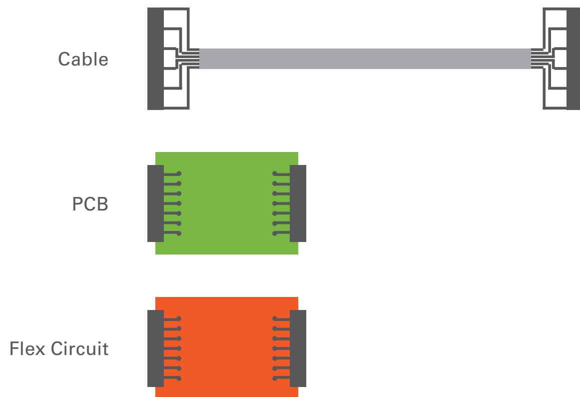


Figure 3. Cable assemblies (top) offer flexibility with mounting and receiver module location but typically have longer signal paths. PCBs (middle) and flex circuits (bottom) minimize signal length, preserving signal quality but offering less flexiblity than cable assemblies.

To connect instruments and other auxiliary equipment to the receiver modules, two methods can be used independently or in combination:

- **Cable Assemblies—**With cable assemblies, standard or custom-made cables are connected from the instruments directly to contacts in the receiver. Cable assemblies offer more flexibility with mounting and receiver module location, but typically have longer signal paths (24 in. or more) between the receiver module and instrument, which can affect performance if not managed correctly.
- **Interface Adapters—**Interface adapters typically connect all I/O from the instrument connectors (for example, DIN, D-SUB, SCSI, and so on) to the receiver modules. The adapters are always directly in line with the instrument and use printed circuit boards (PCBs), flex circuits, or cables to provide the most efficient connection method for the particular instrument. Interface adapters offer the advantage of minimizing the signal length (usually 6 in.) and variable signal performance between the receiver module and the instrument, but are rigid and therefore require more upfront planning and precise, inflexible mounting locations.

Below ○ Average ◒ Above ●

	Cables	Mass Interconnect With Cables	Mass Interconnect With PCBs or Flex Circuits
Frequent Changeover Between DUTs	○	●	●
Optimized for Design and Characterization	●	○	○
Optimized for Verification and Validation (V&V)	◒	◒	◒
Optimized for Test Production	○	●	●
Signal Quality	◒	◒	●
Continuity of Performance (System to System)	◒	◒	●
Ease of System Maintenance and Upgradability	○	●	●
System Reconfiguration (that is, Scalability)	○	●	●
Ease of Duplication (for example, Global Deployments)	○	◒	●
Instrument to Module Pin Efficiency	○	●	◒
Repairability in the Field	●	●	○
Instrument Card Rev. Control Tolerance	●	●	○

Table 1. Cables are useful for designing and characterizing a device, but a mass interconnect is ideal for a production test environment.

DUT-Side Components

The DUT-side components comprise everything between the mass interconnect and the fixture or DUT. DUT-side components are assembled as one unit, often referred to as the ITA, and can be easily interchanged to test different DUTs with a common set of instrumentation. Below is a brief explanation of each of the DUT-side components.

ITA

The interchangeable test adapter (ITA) is the core component of the DUT side of the mass interconnect, encompassing the enclosure, or mechanical frame, that contains the ITA modules and contacts that mate with the receiver and transfers the system inputs and outputs to the DUT. If the receiver is the socket, the ITA is the plug. Many test systems are designed to test many different DUTs by interchanging different ITAs while using the same test system and receiver.

ITA Modules

The ITA modules are mounted inside the ITA in much the same way that the receiver modules are mounted inside the receiver. They provide the main interconnect with the various signals routed through the receiver and expose those connections through cables, PCBs, or other connections inside the ITA enclosure. The ITA modules and contacts are chosen to match the receiver modules and contacts previously specified. The appropriate signals are then routed within the enclosure to the fixture or DUT connectors.

Enclosure

The enclosure is the mechanical housing around the ITA and corresponding ITA cables/modules. It is common to integrate the ITA enclosure and test fixture into a single frame or physical platform on which to set the DUT, as in consumer electronic or semiconductor testing, or have a single cable connected between the ITA and the DUT. Although you can choose from standard enclosures, in practice almost every enclosure is customized in some manner to suit the requirements of the DUT.

Test Fixture

Each DUT is different and requires a unique method of connection to achieve the most efficient testing. For example, some DUTs benefit from a single cable between the ITA and DUT, whereas others benefit most from an integrated test fixture, such as a bed of nails fixture, providing a direct connection method without cables.

How to Choose Your Mass Interconnect System

Upfront planning and design of your Mass Interconnect approach ensures that your test system can perform to its full potential, unlocking the spectrum of capabilities of your chosen instruments, stimulation and data collection requirements. However, performance is only one aspect of the decision-making process and should be combined with a cost comparison of different approaches. Associated overall cost computations should also include design verification procedures, life cycle management of custom components, documentation, and overall maintenance costs spread across the anticipated life of the automated test equipment and fixturing. Working with a provider of Mass Interconnect solutions upfront can save you a lot of time and money.

The following general steps can help you to determine the mass interconnect components designed into your system, but you should also seek advice from a mass interconnect specialist and an NI Alliance Partner, such as MAC Panel or Virginia Panel Corporation (VPC), who can walk you through this process. Mass interconnect providers also offer configured solutions, predesigned for specific PXI assets and common instrumentation, reducing the time and effort required to configure and document the interface. Simply provide a list of the instruments and other assets required in your system and they can provide a corresponding configured part number list for the mass interconnect (interface) and mating (fixture) components.

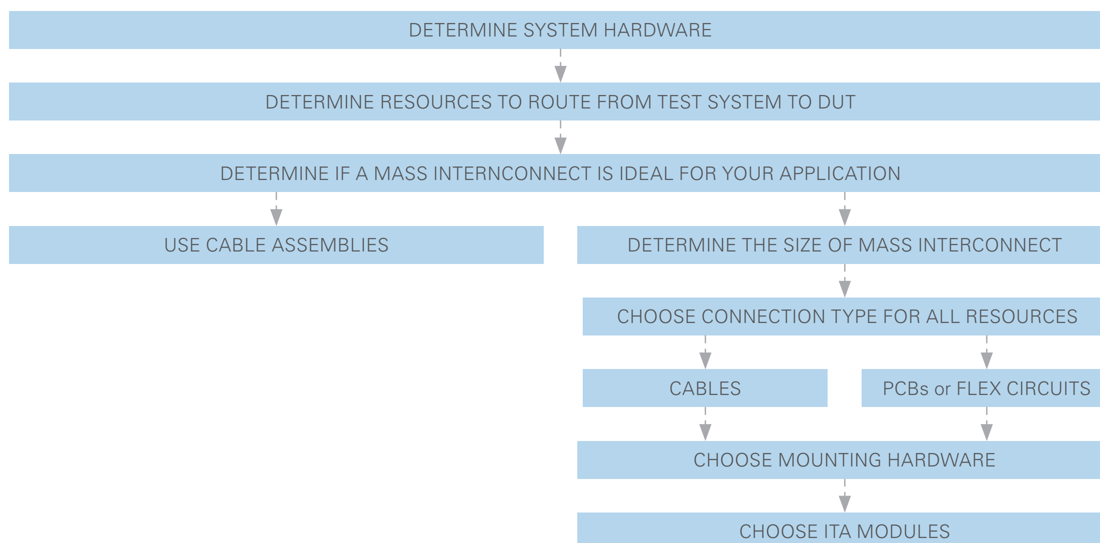


Figure 4. Considerations for Mass Interconnect Systems

1. Determine your system hardware.

First, determine the system hardware needed to test your DUT. This includes instruments, switching, any auxiliary items that you need to route to the DUT (for example, the power supply), and the size and style of test rack.

2. Determine which resources from the test system need routing to the DUT.
Depending on the complexity of the test system and the device being tested, you will most likely route all required instrument and auxiliary component connections through the mass interconnect system. Some test components do not need to be routed through the mass interconnect, such as PXI embedded controllers or RAID storage systems.

When choosing the resources that you plan to route through the mass interconnect, consider how your test requirements might change in the future so you can build a flexible test system that can meet future requirements. For example, if you have extra resources, such as extra instrument or power supply channels, that you do not need for your current generation of DUTs but think you might need for future DUT generations, then you can save time and money by routing those resources in anticipation of future changes.

3. Decide the best style of interface for the application—cables or a mass interconnect.
This choice depends on a number of factors, including the complexity of the system, technical performance requirements, flexibility, and total cost of ownership. Use Table 1 as a guide for your decision, but seek advice from a mass interconnect specialist to ensure you make the correct choice for your particular test system. Assuming that you decide to use a mass interconnect, the following steps can help you determine the size of receiver and ITA and select receiver modules, ITA modules, and mounting hardware.

4. Determine the size of the receiver and ITA.
You can choose from many different receiver sizes and styles, depending on the answers to the previous questions. Consider reserving spare slots in the receiver for future expansion. Similar to planning instrumentation for a PXI system, a general guideline is that a minimum of 20 percent of the slots should be unpopulated in an initial design. Note that the ITA size will always match the receiver size.

5. Choose receiver modules and a connection method (cables or interface adapters) to accommodate routing of all necessary resources.
Your choice of receiver modules and connections depends on your test goals and the chosen instruments and any other auxiliary requirements. Again, you can use Table 1 as a guide for your decision, but seek advice from a mass interconnect specialist to ensure you make the correct choice for your particular test system. You can choose from a wide range of contact styles and predesigned interface adapters, patch cords, or predesigned cables to meet the requirements of any signal type, including:

- Low-frequency AC signals
- Power
- RF signals
- Microwave signals
- Thermocouple
- Fiber optics
- Pneumatic
- High-speed signals and data transmission

6. Choose mounting hardware.
After selecting your receiver modules and connection method, the next step is to choose the mounting hardware, which depends on several factors. Most systems using a mass interconnect will mount to a 19 in. rack assembly. With a cabled system, it is often advantageous to mount the receiver on a hinged frame, or on slides, to allow access to the instruments and chassis. If using interface adapters, the receiver is mounted to the PXI chassis using standard mounting flanges and the receiver and chassis are then mounted onto a slide shelf within the rack.

7. Choose mating ITA modules and contacts.
Configure the ITA modules and contacts to match your choices from step five. It may not be necessary to fully populate contacts in the ITA modules to completely match the contacts in the receiver modules. In most cases, all the resources from a particular instrument will be passed to the receiver module. However, not all of these resources are required on the ITA side to test a particular DUT; therefore, not all ITA modules need to be fully populated with contacts.

Overview of a Test Fixture

A test fixture is a device that provides repeatable connectivity between your test system and your DUT, and it is often custom designed to meet the needs of a specific DUT. Using a mass interconnect at the fixture back side, you can interchange various test fixtures with a common rack of instruments; you can reuse the test equipment by easily interchanging the test fixtures for different DUTs, which is ideal for a universal tester or high-mix tester.

When designing a test fixture, know in advance what types of tests you plan to perform, because the needs for device design, DUT characterization, verification and validation (V&V), and production testing are quite different and require different features from a test fixture.

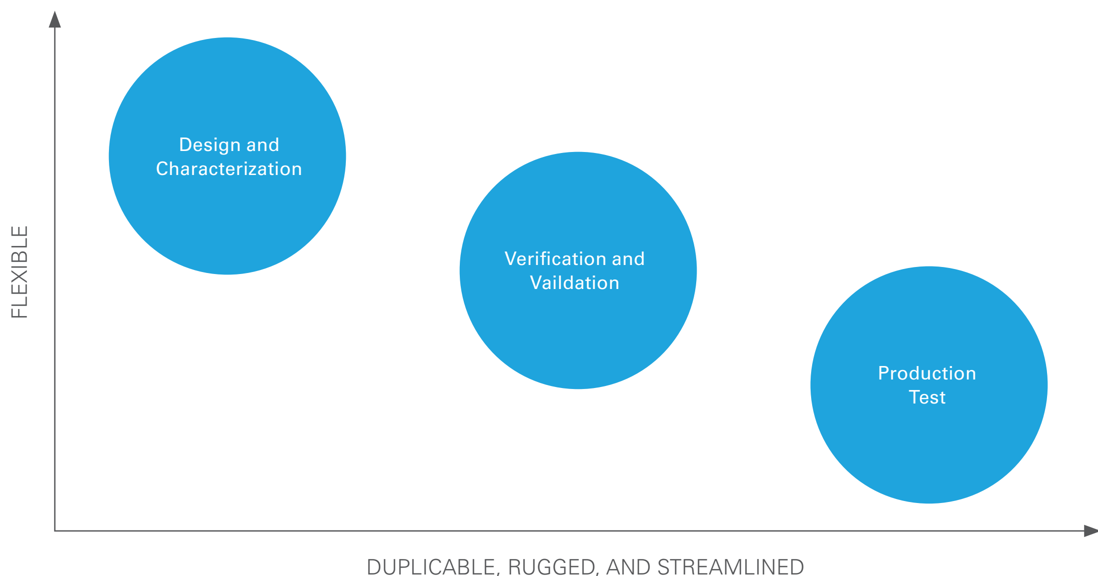


Figure 5. Production test systems require rugged, streamlined test fixtures that you can easily duplicate, whereas design and characterization instrument systems require the flexibility that cables, probes, and Kelvin clips provide.

Design and Characterization

During a product's design phase, you must have the flexibility to easily connect and disconnect one or more cables or pins from the DUT to the measurement instrumentation to test or troubleshoot any and all elements of the design. For this reason, you need a highly flexible way of interfacing with instrumentation rather than a streamlined or rugged test system with mass interconnect and fixture. Many design engineers use instrumentation with cables, probes, and Kelvin clips, which provide a way to easily change connections. In addition to troubleshooting, design engineers often characterize, or describe, the actual behavior of a device, which contributes to device specifications and guides test engineers in developing a V&V station or production test system.

Verification and Validation

V&V is the process of checking whether a specific product meets design specifications by testing a statistically representative set of the product.

Verification

Verification is an objective process to check that a device meets the required regulations and specifications. You often perform verification during the design and development phase, as well as after the development phase is complete. During the development phase, verification testing sometimes simulates or models the behavior of the rest of the system to predict or preview how a device might behave when it is complete. After development is complete, verification testing can take the form of regression testing, repeating many tests designed to ensure that the device continues to meet the design requirements as time progresses.

Validation

Validation is a more subjective process, involving subjective assessments that a device meets the operational needs of the end user by returning to the problem statement that created the need for a new product. Requirements for validation testing can come from user requirements, specifications, and/or industry regulations. When validating specifications, the goal is to ensure that the specification captures user requirements, not ensuring that a given device meets its specifications.

The test data collected during the V&V process is also used to determine the test limits for the production test system. Although only a small set of a given product undergoes V&V, almost all final products pass through production test. As a result, fixtures designed for V&V are often required to make fewer connections between the DUT and instrumentation, allowing them to be less rugged than those used for production test.

Production Test

Production, or functional, test is typically performed at the end of the manufacturing process and tests whether the product meets the published specifications and quality standards. Many times, functional tests are automated to optimize throughput and reduce errors from human interaction. Sometimes, production tests include the emulation or simulation of the environment in which the product will eventually operate. Most importantly, functional tests use the final connectors that a customer will eventually use rather than various test points directly on a PCB.

Because every device built and shipped undergoes production testing, you must design your fixture to be rugged to maximize uptime and be easy to use and ergonomic. You should also minimize the amount of interaction that the test operator must provide. Extra time spent aligning the DUT to the fixture or connecting cables from instrumentation to the fixture negatively impacts throughput and increases test costs, as well as increases the likelihood of errors because of human interaction. Finally, production test systems, including a test fixture, should be easy to duplicate for additional deployments.

Fixture Considerations

When designing a test fixture, ensure that it uses proper wiring types and techniques, uses PCBs rather than cables when possible, and automates as many connections as feasible. Also create a preventive maintenance plan for your test fixture to ensure a long and successful deployment.

Use Proper Wiring

Wires are often sources of noise and error, so you should select the best type of wires for your test fixture. To ensure signal integrity, some wires offer features such as insulation, shielding, guarding, or twisted pairs. Some instrument manuals suggest using specific cables, but it often depends on the type of measurements being performed. For example, twisted pair cables are ideal for rejecting noise when performing differential measurements. Using a shielded cable is also a technique for rejecting noise, but it is important to use the correct grounding scheme based on the grounding of your signal source and input configuration. Finally, guarding is often used to remove the effects of leakage currents and parasitic capacitances between the HI and LO terminals of a digital multimeter (DMM) or source measure unit (SMU).

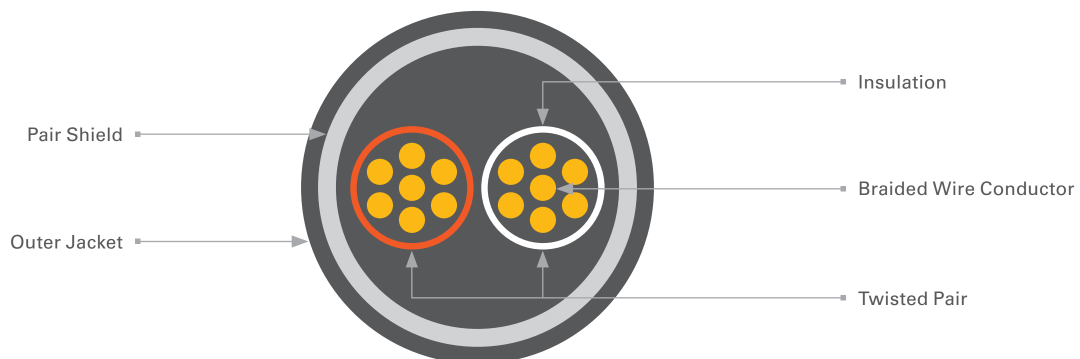


Figure 6. To preserve signal integrity, different cables offer different features such as shielding, insulation, twisted pairs, or guarding.

Minimize Operator Interaction

The goal of a mass interconnect system is to increase test equipment reuse and reduce the amount of operator interaction that can cause errors, lower throughput, and increase the total cost of test. In addition to limiting user interaction with a mass interconnect system, a good test fixture should maximize the number of connections made between the test fixture and DUT with a single interaction from the test operator. For example, some test fixtures complete multiple connections with a connection method that is either driven with a single operator handle, or automatically with an electric or pneumatic motor. For production test, the connections are typically made using the connector that a customer eventually uses to interface with the device.

Build a Scalable Fixture That Is Easy to Duplicate

Certain wiring techniques can preserve signal integrity, but consider using PCBs within your test fixture to improve signal integrity performance and reduce the wiring burden on the test operator or technician during the setup phase. Using a test fixture that makes many connections at once is good for throughput, test repeatability, and user ergonomics, but it is still important to reduce the amount of wiring within the test fixture and replace them with a PCB when possible, as this can further improve signal integrity and reduce the time required to set up and wire each duplicate test system. The trade-off is that designing a custom PCB requires more upfront cost and effort, but pays dividends during system setup for the first system and for each reproduced test system.

Create a Preventive Maintenance Plan for Your Test Fixture

To ensure long-term support for your test system, include test fixture maintenance plans in your overall test system maintenance plan. This should include inspection and/or replacement of connectors, cables, pogo pins, relays, and other components at regular intervals throughout the test system’s lifetime. When choosing inspection cadence, consider the failure rate of a given component. To judge failure rate, take into account the vendor-supplied mean time between failure (MTBF) or the in-service failure rate specific to your organization. Although both are useful when comparing or analyzing instrumentation, the in-service failure rate can be more useful, because it is indicative of how an instrument is actually used for a particular application. In the case that in-service failure rate data is unavailable, you can derive a theoretical failure rate from the MTBF data to plan an inspection interval that aligns with your cost and risk tolerance.

Next Steps

NI PXI Advisor

The PXI Advisor helps you compare instrument options and configure a PXI-based test system, including PXI chassis, controller, modules, software, services, and auxiliary items.

Configure your test system using our PXI Advisor

MAC Panel

MAC Panel is a solution source for companies looking to achieve reliable, cost-effective, electrical connections in a test or measurement environment. In addition to a full line of mass interconnect products, featuring the MAC Panel SCOUT mass interconnect system designed for PXI, MAC Panel also provides custom wiring services, sheet metal fabrication, and custom design assistance, providing a range of options to support automated test equipment on a global scale.

Learn more about MAC Panel

Virginia Panel Corporation

Virginia Panel Corporation (VPC) is an ISO certified manufacturer of Mass InterConnect solutions. With over 150 employees, VPC is able to design and manufacture large or small I/O connectors and interfaces for the Test and Measurement industry. In addition to Mass InterConnect solutions for PXI platforms, VPC also provides several value-add services like high speed PCB design, pre-configured test solutions, custom design assistance, web access to product support files, and online configuration tools.

Learn more about VPC

NI Alliance Partner Directory

The NI Alliance Partner Network is a program of more than 1,000 independent, third-party companies worldwide that offer complete products, as well as integration, consulting, and training services. Some of these companies, such as MAC Panel and VPC, offer custom cabling solutions, mass interconnect systems, and complete fixturing solutions.

Browse the Alliance Partner Directory

©2016 National Instruments. All rights reserved. National Instruments, NI, and ni.com are trademarks of National Instruments. Other product and company names listed are trademarks or trade names of their respective companies. A National Instruments Alliance Partner is a business entity independent from National Instruments and has no agency, partnership, or joint-venture relationship with National Instruments.

NATIONAL INSTRUMENTS™

FUNDAMENTALS OF BUILDING A TEST SYSTEM

Software Deployment

CONTENTS

Introduction

Managing and Identify System Components

Hardware Detection

Dependency Resolution

Release Management

Release Testing

Componentization

Summary

Introduction

Given more complex devices, test engineers need to create more complex and higher mix test systems, often with tighter deadlines and lower budgets. One of the most important steps in creating these test systems is deploying test system software to target machines. It is also commonly the most tedious and frustrating step. The abundance of deployment methods today typically adds to the irritation of engineers simply searching for the cheapest and fastest solution. In addition, test system developers face many considerations and sensitivities specific to their system.

Deployment, for the purposes of this guide, is defined as the process of compiling or building a collection of software components and then exporting these components from a development computer to target machines for execution. The reasons test engineers employ deployment methods rather than run their test system software directly from the development environment come down mainly to cost, performance, portability, and protection. The following are common examples of inflection points when a test engineer will move from development environment execution to a built binary deployment:

- The cost of application software development license for each test system begins to exceed budget limitations. Using deployment licenses for each system offers a more attractive and efficient solution.
- The source code for the test system becomes difficult to transport due to memory limitations or dependency issues.
- The test system developer does not want the end user to be able to edit or be exposed to the source code of the system.
- The test system suffers lower execution speed or memory management when run from the development environment. Compiling the code for execution provides better performance and employs a smaller memory footprint.

This guide recommends and compares different considerations and tools to address the difficulty and confusion that surrounds test system deployment. Although there are many different topics of test system deployment that could be addressed in this guide, such as source code control best practices or creation of installers, the selected topics should cover the majority of universal deployment concerns. The end of each section offers a best practices recommendation for a basic use-case and an advanced use-case:

- The basic use-case is a simple test system composed of an executable that runs test steps in sequence and calls a handful of hardware drivers. This type of system usually comprises less than 200 test functions.
- At the end of each basic use-case best practice, is a handful of warning signs or indicators for when one should consider the advanced use case.

The advanced use case represents a large-scale production test system that uses a combination of executables, modules, drivers, web services, or third-party applications to execute a high mix of different test sequences. This type of system is often in the range of hundreds or even thousands of test functions.

Managing and Identifying System Components

Defining Components

In software development, a component is any physical piece of information used in the system, such as binary executable files, database tables, documentation, libraries, or drivers. The first step to completing successful deployments is identifying the components associated with a test system and ensuring that each component has a deployment method in place. This step can vary widely in complexity. For example, components for a simple test system could be a single executable and necessary hardware drivers.

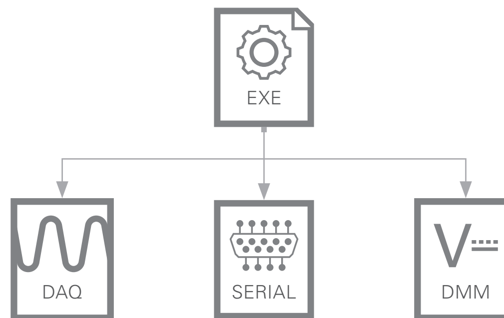


Figure 1. Simple Representation of a Test System Executable That Depends on a DAQ, Serial, and DMM Driver

Complex System Components

In a complex test system, however, these components are often XML configuration files, database tables, readme text files, or web services. This increase in a system's complexity opens the door for more advanced deployment options. For example, it's possible that the configuration file needs to be updated frequently to calibrate acquired data to seasonal weather changes, whereas the main executable rarely needs an update. It would be unnecessary to redeploy the executable along with the configuration file every time an update is needed, so the configuration file may employ a separate deployment method than the executable.

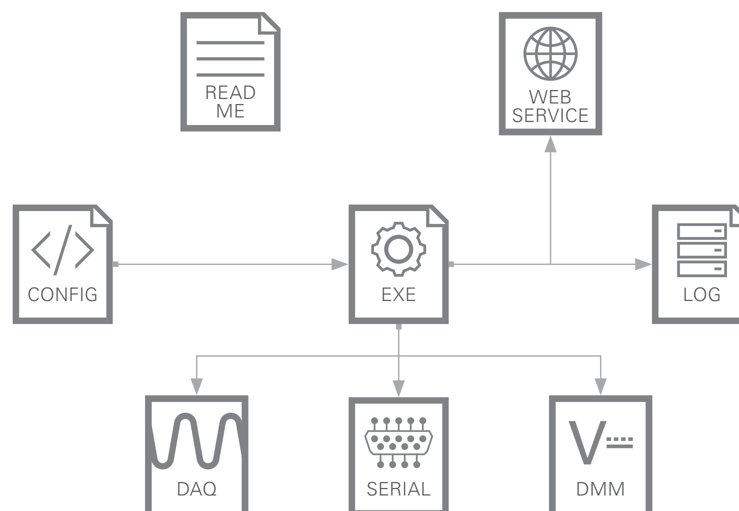


Figure 2. Example of a Test System With Complex Dependencies

In addition to identifying each system component and devising its deployment method, it is important to identify the relationships between the system components and ensure the deployment methods do not interrupt those relationships. In the example of the frequently updated configuration file, the engineer might have to install the configuration file to the same location on each deployment system so that the executable can locate it at run time.

Dependency Tracking

Maintaining the relationships between dependencies involves assembling a dependency tracking practice that ensures each component's dependency components are deployed. Although this may seem obvious after manually identifying each system component, dependencies can often be deeply nested and require automatic identification as systems scale. For example, if the executable in System B was dependent on a .dll to execute correctly, the engineer creating the deployment plan may have either forgotten to identify the .dll file as a necessary component or been unaware of the dependency. In these cases, build tools come in handy by automatically identifying most, if not all, of the dependencies of a built application.

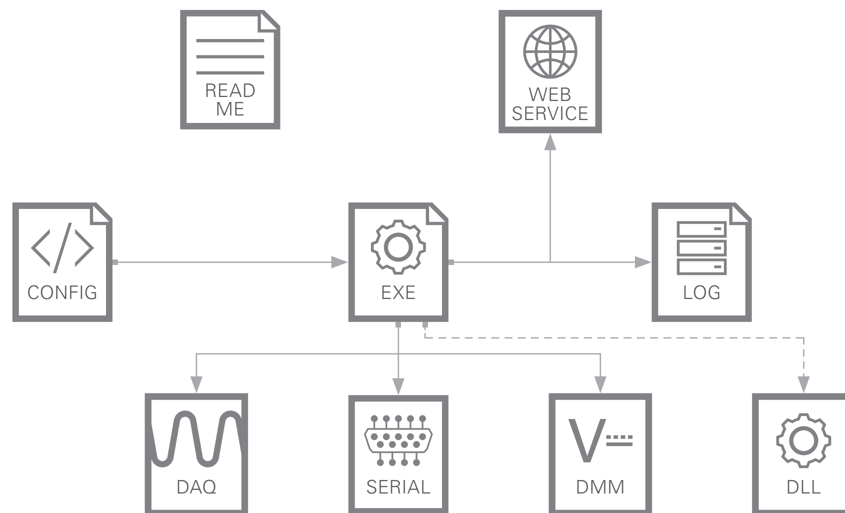


Figure 3. Unexpected Dependencies in a Complex Test System

Here are examples of build software applications:

- **LabVIEW Application Builder—**Identifies the dependencies (subVIs) of a specified set of top-level VIs and includes those subVIs in the built application
- **TestStand Deployment Utility (TSDU)—**Takes a TestStand workspace file or path as input and identifies the system's dependency code modules; automatically builds and includes these modules in a built installer
- **ClickOnce—**Microsoft technology that developers can use to easily create installers, applications, or even web services for their .NET applications; can be configured to either include dependencies in an installer or prompt the user to install dependencies after deployment
- **JarAnalyzer—**Dependency management utility for Java applications; can traverse through a directory, parse each of the jar files in that directory, and identify the dependencies between them

Relationship Management

Commonly, relationships exist not only between the main test program executable and its associated components but also between each of the individual components. This brings into question the nature of the relationships between different components or software modules. As systems scale, resolving dependencies between different libraries, drivers, or files can become extremely complex. For example, a test system could use three different code libraries with the following relationships, shown in the figure below, to each other.

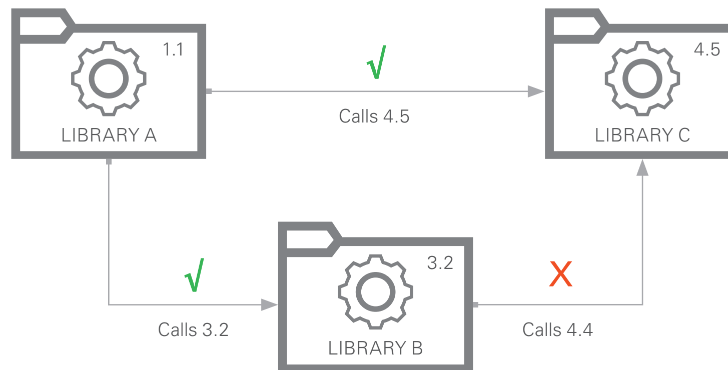


Figure 4. Library B's reliance on version 4.4 of Library C causes an unsolvable dependency issue as Library A relies on version 4.5 of Library C.

For these complex systems, it is usually necessary to employ a dependency solver to identify dependency conflicts and manage unsolvable problems. Although it is possible to write a dependency resolver in-house, engineers can instead put in place a package management system to manage dependencies. An example of a package manager is NuGet, a free, open-source package manager designed for .NET framework packages. Another example is the VI package manager for LabVIEW software that gives users the ability to distribute code libraries and offers custom code library management tools through an API.

Best Practices

Basic: For basic or simple systems, it is usually possible to keep track of all the necessary components manually. Using a software application or package manager to manage dependencies might be unnecessary and require too high of an up-front cost to set up. However, warning signs, such as consistently running into missing dependency issues or a growing list of dependencies, usually point to the need for more advanced dependency management.

Advanced: Complex systems are easier to maintain and upgrade when a scalable dependency management system is in place. Whether this means using a package manager to diagnose relationships between packages or a software application to understand and identify dependencies of various components, maintaining such a system is critical to long-term success.

Hardware Detection

Hardware Assertions

A test system that requires a specific hardware setup needs to determine that this hardware is present on the system and execute contingency plans for when the hardware is absent or incompatible in its deployment plan. Although developers frequently complete hardware assertion manually by visually inspecting the test machine and matching the hardware components to the original development system, it is good practice to assume the test system is being created for a third party. How would a customer of the test system know they have incompatible hardware? Can the system adapt to the correct modules in incorrect slots or ports? Can the system resolve or adjust for missing hardware? Answering these questions early on makes for simple scaling and distribution of test systems.

Hardware Standardization

The ultimate goal for hardware assertion is to find no differences between the expected system and the actual physical hardware system. To this end, it is often most efficient to first standardize each test system on the set of hardware components they will use:

- **Documented—**The list of components in the standard set of hardware should be accessible for every new system. It is critical that this documentation contain information about the provider, product numbers, order numbers, count, replaceable components, warranty, support policy, product life cycles, and so on.
- **Maintainable—**One of the most difficult issues for hardware standardization is ensuring that the hardware components used in each test system will still be available in the future. Often, older hardware is indicated as in end-of-life (EOL) by the manufacturer and requires a refresh of the standard set of test system hardware components. This refresh is often expensive in terms of both hardware upgrades and test system downtime. Working with a hardware manufacturer to discuss life-cycle policies for hardware components can offset challenges in the future. Most hardware manufacturers, such as NI, provide life-cycle consultancy and a slow roll-off in each hardware component's life cycle.
- **Replicable—**The necessity to distribute hardware globally or even regionally should be considered. Ensuring a hardware distribution method is in place to quickly construct new systems in remote locations is an important concern. Maintaining a pipeline for spare hardware components for maintenance or emergency replacement is also important for many systems.

Power-On Self-Test (POST)

Even though the correct hardware for the test system may be present and connected properly, it is also important to do simple testing of the hardware to ensure that it will behave as expected once the system is running. Fortunately, most hardware components contain a preconfigured self-test designed by the manufacturer to perform a simple check of the device's channels, ports, and internal circuit board. Upon providing power to each test system, a self-test procedure should be performed for all connected devices to act as an early check for malfunctioning hardware. For example, each NI device features a self-test that can be called programmatically through the device's driver API. The first step when powering the test system can then be calling a self-test on each device and warning the operator of any malfunctioning hardware.

Alias Configuration

Unfortunately, standardizing on a hardware set does not completely ensure identical configurations. Commonly, hardware configuration software, such as Measurement & Automation Explorer (MAX), is required to remap hardware devices to aliases. For example, upon installing all the hardware components and powering the system, engineers can use MAX to detect NI hardware present on the system and use Windows Device Manager to find non-NI hardware. Subsequently, a .ini configuration file can be edited to map hardware devices correctly to aliases. The figure below shows an image of a possible output of this process.

Alias	Device Name
PXI NI-4139	PXI 1 Slot 1
PXI NI-3245	PXI 1 Slot 2
PXI NI-2239	PXI 2 Slot 1

Table 1. hw_config.ini File Used to Map Physical Hardware to Test System Aliases

Programmatic Configuration

Libraries like the System Configuration API for NI hardware in LabVIEW software make it possible to programmatically generate a list of all available live hardware and configure an alias mapping. For example, a test system executable could call into the System Configuration API's Find Hardware function to generate a list of available NI hardware. From there, the alias property for each device could be set to a predefined name through the Hardware Node. This has the potential to cause issues in a system, such as mapping a hardware device to the inappropriate alias. Therefore, engineers should use it in conjunction with another safeguard like manual confirmation of the mapping list or a standardized hardware set.

Best Practices

Basic: For basic or simple systems, it is important to ensure that the expected hardware is present on the system. Hardware standardization is a best practice for all systems and especially important as the number of hardware systems begins to increase. The chassis, modules, and peripheral devices necessary for proper execution of the test system should be documented and revisited regularly. However, verifying that the right devices are live on the system can often be done through manual inspection with a tool like MAX instead of a programmatic or reconfigurable solution. As a hardware system grows in number of modules and devices, it may be necessary to move to a more advanced solution to prevent missing hardware issues.

Advanced: In complex systems, keeping track of what hardware is necessary or present on the system should be done with a combination of different solutions. Just as in the basic best practices, hardware should be standardized and documented across systems. To detect malfunctioning hardware, a power-on self-test (POST) should be developed to ensure the connected hardware will function as expected. In addition, a programmatic or minimally manual alias mapping system should be used to automatically remap the expected devices to the system's aliases when hardware standardization fails.

Dependency Resolution

Dependency Assertions

It is good practice for a plan to be in place to address existing and missing dependencies on deployed systems. Often, the test machine being deployed to will already have some of the test system image's dependencies installed to it. For smaller systems, it may be a good idea to simply reinstall all dependencies to ensure they are present. However, for larger systems, reinstalling all dependencies can potentially be avoided by first checking whether those dependencies are present on the system. This practice is referred to as dependency assertion and can help reduce deployment time but comes at the cost of needing to plan for dependency differences. The Componentization section further discusses componentizing for faster deployments.

For example, a test system might be compatible with both the 14.0 and 15.0 versions of the NI-DAQmx driver. Although the test system might call for NI-DAQmx 15.0 to be installed, it might allow the 14.0 version to act as this dependency. However, allowing the 14.0 version instead of the 15.0 version, although compatible, might change how the test system acts. Certain test steps may be skipped or different functions called. All of these changes would need to be documented and tested.

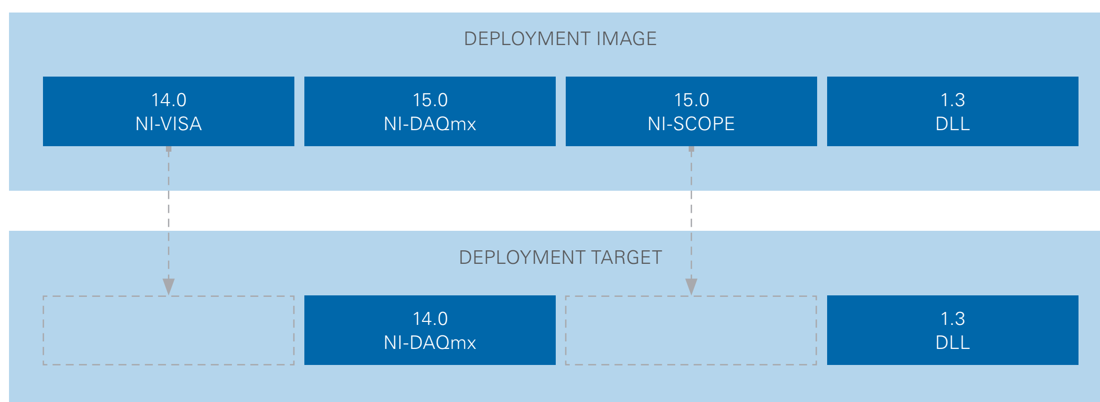


Figure 5. Dependency Assertion

The second element of dependency assertion is deciding how to handle missing dependencies. As stated earlier, a good practice to follow is to act as if the test system is being deployed to a customer's machine. Should the engineer completing the deployment be notified of the missing dependency? Should the missing dependency silently install in the background or will the user need to go find and install the dependency manually? Answering these questions early on can allow for faster deployments and appropriate handling of missing dependencies.

Best Practices:

Basic: For basic test systems, dependency resolution and assertion is often unnecessary. Installing all of the test system's dependencies, regardless of whether they are present in the system, is frequently simpler than attempting to identify missing dependencies and install only the missing elements. As the test system scales, total system install times may increase to the point at which developing dependency assertion and resolution tools becomes a more attractive solution.

Advanced: Deployment times can quickly scale to unreasonable amounts, even with solid network connections or compressed images. For most advanced test systems, some amount of dependency assertion is necessary to prevent a reinstall of all components. Tools like the System Configuration API to find NI software installed on a system or the wmic command set to generate a list of all programs on a Windows machine can be incorporated into deployment processes. This can allow installers to skip specific components or allow for version differences.

Release Management

Often, engineers need to know which version of software image is currently deployed to the test system or be able to provide a release deployment history. If these are necessary requirements, there should be a release management system in place to address each of the following questions:

- Which release is currently deployed to System A?
- What is the status of the most recent deployment to System B?
- Where have releases 1, 2, and 3 been deployed?
- What is the history of releases for System A?

In most test environments, engineers answer these questions with a pencil and clipboard system, however, tools exist to automatically record release metrics and provide documentation on the release history for a specific system. These tools for release management can be incorporated into an integrated development environment (IDE) or exist as stand-alone release management tools. Some examples include:

- **Visual Studio Release Management—**The Visual Studio IDE is shipped with tools to automate deployments, trace release history, and manage release security.
- **Jenkins Release Plugin—**With this plugin for the Jenkins continuous integration (CI) service, developers can specify pre- and post-build actions to manage releases for their Jenkins-integrated development.
- **XL Deploy—**This application release automation (ARA) software can scale to enterprise levels and provide visual status dashboards, security, and analytics for managing releases.

Although the above examples serve as good tools for IDEs and stand-alone deployment solutions, more commonly, release management tools are found in conjunction with CI servers and end-to-end deployment processes. This is intuitive because the question of what specific code is present on a certain machine is more applicable to deployment processes than which release version is on a certain machine. For compiled system images, this can be difficult to ascertain by manual observation. Tracking the code from development to deployment is necessary for release management best practices.

End-to-End System Automation

Efficient release management is a necessary component in developing a more complex end-to-end process for test system deployments. From development to deployment, each process in series relies on its predecessor; if source code is managed well, testing and building can be managed in turn. With good testing and build processing in place, release management can be a simple extension of the original system. The following diagram displays a typical end-to-end system.

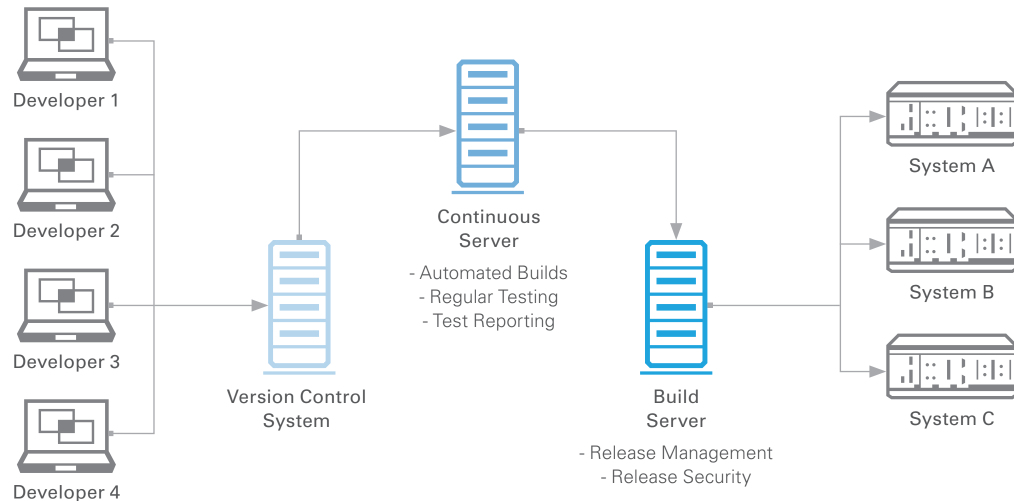


Figure 6. Developers submit code to a version control repository that can then be built and tested in a CI server. From there, the builds can be stored in a build server and undergo release management.

In this setup, test system developers regularly develop and commit source code to a version control repository. From there, a CI service can pull the source code into its own repository and build and test the code appropriately. At this point, either automatically or manually, developers can move and store builds that pass the CI tests to a build server or repository. Here, on the build server, release management takes place with reporting and tracking to link each software build to a specific test machine. Usually, the test machines initiate the deployment process through a request to install a specific release of the test system; however, developers can also configure build servers to push images onto a chosen machine.

In cases where even basic systems need to employ a level of release management, the most pragmatic solution will reflect the inherent complexity of the release requirements. If the requirement is to track which version is deployed to a system, manual versioning through a configuration file or as a component of building an executable can be sufficient. If requirements expand in scope, the number of test systems increases, or application version numbers grow, it will be necessary to use a defined release management system.

Best Practices:

Advanced: Frequently, a complex test system in need of release management will be most successful with some form of end-to-end automation. This can most simply be done through a CI service such as Jenkins or Bamboo that ties release management to release testing and source code control.

Release Testing

Regression Testing

In software engineering, regression testing refers to the process of testing a previously developed system after changes to the system are made. The purpose of regression testing is to maintain integrity for each release and track bugs in the system to specific updates or patches. For componentized systems, regression testing is especially important to determine if an upgrade to module A causes unexpected behavior in module B. For example, upgrading the NI-DAQmx hardware driver in the system could cause issues with a hardware abstraction library that called a function in the older NI-DAQmx version that is now deprecated in the newer. There are two types of regression testing: functional testing and unit testing.

Functional Testing

In testing systems, the most important questions to ask about a software update is, will this change break the functionality of the system and does the system still behave the way it was intended? Functional testing, which verifies that for a set of known inputs, the system produces expected outputs, can help answer these broad questions about the system as a whole. This type of testing usually takes a "black-box" approach; inner mechanisms of the system are not analyzed, only whether the output of the system is as expected. For test systems, this could be a verification that hardware configuration updates, driver changes, or test step additions do not change the original testing functionality. Engineers can perform functional testing on a test system using simulated devices under test (DUTs) that are calibrated to pass or fail certain tests. For example, a system that tests for whether an object is a circle is made up of four components: a camera controller, circumference sensor, diameter sensor, and volume sensor. If the system is updated from version 1.0 to 1.1 and a change to the diameter sensor is introduced, the second circle being tested, in the diagram below, would originally pass the circle tester and then fail after the update.

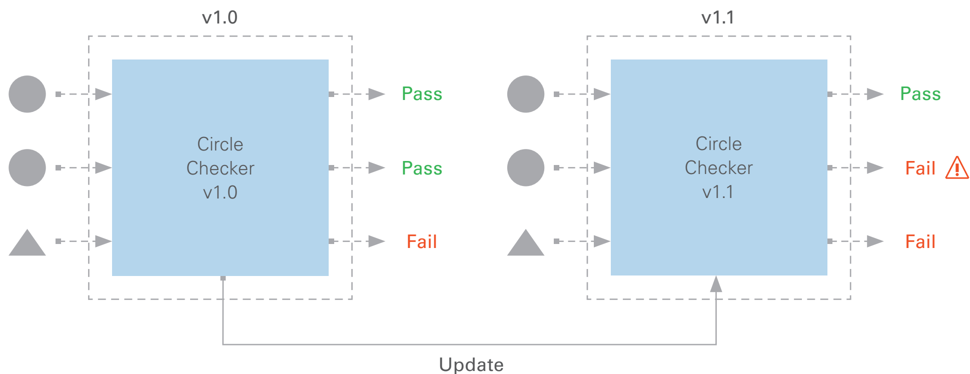


Figure 7. A small update to a module in the test system can cause the functional test to malfunction, resulting in false failures.

Unit Testing

Whereas functional testing is for the complete system, unit testing is for specific modules, components, or functions. This type of testing is intended to track the quality of specific portions of the test system as opposed to just correctness. For example, if test results are being logged to a database, a unit test may be done on the database controller to measure data throughput. In this way, any changes to the database controller not only can be analyzed for proper logging functionality but also answer the question of whether the software change sped up or slowed down the system's logging capability. In addition to helping find bugs, unit testing can link observed performance enhancements or diminutions to specific changes. The circle tester example from before can clarify the difference between unit testing and functional testing. Assuming the diameter sensor software component of the circle tester was upgraded as before, a unit test of the diameter sensor can be done instead of a functional test of the complete system. For the unit test, one might provide the specific component with binary image data that represents a circle with a specific diameter and test for whether the output matches the known diameter of the circle. In this way, the module's correctness can be verified and quantitatively measured, say, to measure the execution time of the module.

In this specific case, the upgrade slowed down the module significantly. It can also be deduced that, because the functional test of the system failed after the upgrade and the unit test passed, the software bug most likely resides in the communication between the camera controller and diameter sensor. This ability to verify system correctness and individual module functionality can ensure that only quality releases get deployed to test machines.

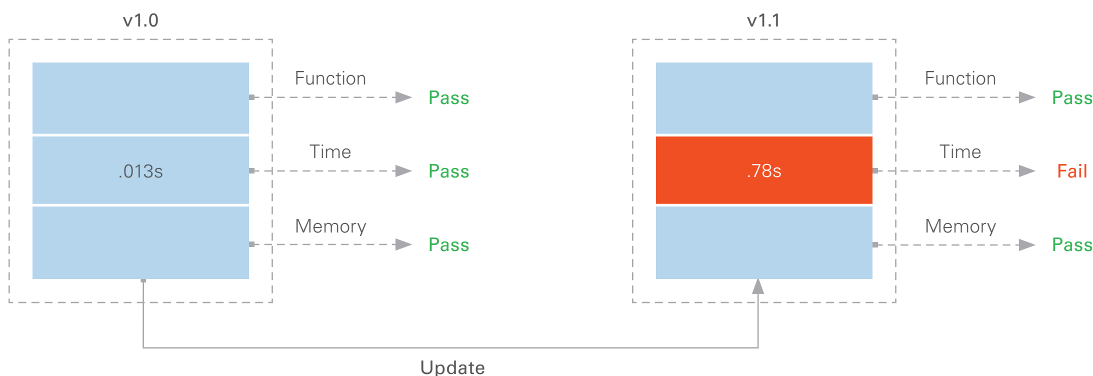


Figure 8. After performing a unit test of both the v1.0 and v1.1, the processing time is identified as a problem after the update, causing false failures in the process functional test.

Testing Process

To save development time, regression testing in most test systems happens in conjunction with source code control, building, or release management. This allows reuse of testing code that should require more infrequent updates. However, it is also important to plan and budget development time for building out test code. Commonly, regression testing is a component of either a CI service or IDE, where source code control, building, and testing all happen in sequence.

Best Practices:
Advanced: For all test systems, there should be some level of functional testing before deploying a system to a new machine. This functional testing can range from the simple case of manually running the application in the development environment using simulated hardware to a slightly more complex case of running through a series of functional tests based on a configuration file. Unit testing may be unnecessary for simpler, more monolithic applications but needs to be considered as a test system scales in complexity. As more modules are added, specific, customized tests are necessary to track bugs or ensure the system meets certain specifications.

Best Practices:
Advanced: Complex test systems should not only have functional testing over a wide array of inputs done for each new release of a test system but also have unit tests developed for each individual module of the system. Both methods of regression testing should be done at the most effective point in the deployment process. For example, performing functional testing after building each release and unit testing at each source code control submission point would represent a good mix of regression testing. Often, these tests are mandatory or self-evident for systems, especially in the aerospace and defense industry.

Componentization

Because deployment time is a common concern for large test systems, it is preferable to update only a single component of the test system that requires a change rather than rebuild the entire system. The Dependency Resolution section of the guide partially addresses this practice, but it deserves a separate discussion around developing modular or plugin-based architectures with the goal of more efficient deployments. Whichever architecture an engineer chooses, best practice dictates that there exist regularly updated peripheral modules and more core modules that, when developed, stay relatively constant without a need for recompile. This practice naturally leads to questions about update frequency, explored later in this section.

Deploying Plugin Architectures
A plugin in the context of software deployment is a code module whose installation is independent of the main application's installation, is functionally independent of other plugins, abides by a global plugin interface, and avoids name conflicts when used in a built application. The main application then should be able to load each plugin dynamically, call each plugin by a standard interface, and use each plugin as an extension without requiring a recompile. When developed successfully, a plugin framework allows for componentized deployments—updating or installing only specific or missing plugins and not recompiling the main application or any unaffected plugins.

For example, a plugin framework developed for a simple application might consist of a main executable that searches through a plugins directory at load time, or periodically during run time, and executes that plugin through a standard interface. In this way, plugins can be continually deployed into the plugins directory of the system without editing the main application.

Hard Drive Replication

Commonly, code libraries, hardware drivers, or specific files are part of a test system's core and do not need to be updated as frequently as other modular, peripheral components. In these instances, hard drive replication can be a good method for standardizing the environment as a baseline for further development. Engineers can replicate and clone the hard drive of a development machine or ground-zero test machine onto other test machines. When the drive has been duplicated, test machines have a common starting point that often includes a main test application or program, necessary hardware drivers, a system driver set, and critical peripheral applications, such as MAX for hardware configuration. It is important to recognize, however, that hard drive replication comes with its own caveats, such as requiring identical computer hardware between test machines, or memory-intensive image distributions that make it an unsuitable method for frequent software updates.

An example of using hard drive replication for laying a foundation for further test development is using Symantec Ghost, a popular hard drive replication tool, with the TestStand Deployment Utility (TSDU). In the first frame of the image below (A), the development machine replicates its core software stack (red) onto the target machine. This core software stack is a combination of the Windows OS, hardware device drivers, run-time engines, and MAX. After the target machine has been imaged, development on the development machine takes place (B) to create a test sequence using TestStand and LabVIEW (green). The developer can then move the test sequence to the target machine using the TSDU. For frequent updates to the test sequence, the developer can continually use the TSDU to save development time, as the core software stack does not need to be changed. Occasionally, development might occur on the development machine that is not deployed to the target machine (C). This system mismatch can potentially lead to problems with missing dependencies. In this instance, a developer could, instead of using TSDU to update the target machine, choose to reimage the development machine and replicate it onto the target machine to realign the two machines (D). Moving forward, the developer can continue to make frequent updates with the TSDU and whenever system mismatches arise in the future, can use Ghost to reimage the hard drive of the target machine.

Figure 9. TSDU and Hard Drive Replication Example

Continuous Integration and Continuous Deployment

Continuous integration (CI) refers to the practice of continuously submitting, building, and testing code, usually on a separate CI server. In most test systems, CI services are used to provide the necessary framework for building, testing, and deploying system software. These services run regularly and automatically on the CI servers with a wide variety of configuration options to create build schedules, automated testing rules, release deployments, and so on. One of the most obvious advantages to using a CI server is the ability to track and manage different builds and deployments.

BUILDS

VERSION	STATUS	LAST BUILD
1.2	Fail	May 24, 2016
1.1	Pass	April 2, 2016
1.0	Pass	December 7, 2015

DEPLOYMENTS

VERSION	STATUS	LAST BUILD	MACHINE
1.0	Pass	June 7, 2015	A
1.1	Fail	June 9, 2015	B
1.1	Pass	March 16, 2016	C

Table 2. CI services provide dashboards to track application builds and deployments.

CI tools range widely in capabilities and open-source developers and software companies both develop them. The latter of which has the added benefit of providing support for system setup.

- **Jenkins—**Termed the "leading open-source automation server," Jenkins is one of the most popular CI services today as it allows for easy installation and configuration. Jenkins can also be used with virtually all programming languages as it can interface with programs through their command line interface or through a wide array of Jenkins plugins.
- **Bamboo—** The software company Atlassian produces Bamboo, the leading proprietary CI service. In addition to the testing, building, and integration functionality that Bamboo provides, Atlassian boasts "first-class support for deployments" over Jenkins.
- **Travis CI and Circle CI—**These two open-source CI services offer great extension capabilities but only integrate with projects that reside in a GitHub repository.

Overall, the goal of CI is to provide automatic and configurable tools that give developers the ability to continue coding while their software is built and tested.

Best Practices: Basic:

Basic: Componentization is often not a large concern for simple systems. Although the system uses very few code modules or does not employ a plugin architecture, each test system can usually be deployed as a stand-alone application. However, if install times become very large and begin to slow down deployment times, it may be necessary to move to a more componentized approach that removes the need for a reinstallation of all components.

Advanced: When test systems become large, complex, or use a plugin architecture, it makes sense to move away from a monolithic deployment image and toward a modular deployment where each component can be updated separately. Using a plugin architecture is a quick way to achieve this modular setup but can also be accomplished through configuration of CI services.

Practical Scenario

An audio equipment production company that does functional electrical testing on its products using TestStand and LabVIEW is an example of a more advanced deployment framework. The test department of the audio equipment manufacturer has over 50 test systems distributed globally. Each system uses a PXI chassis that houses a high mix of modules, including data acquisition, digital I/O, digital signal acquisition, digital multimeter, and frequency counter cards.

The test engineer in charge of deployment follows the outlined procedure for every new test system to be brought online.

1. Creating the Base System Image

For each new test system, there is a list of necessary software, both company made and third party, needed to ensure security of the system. The company's IT department requires this software and it includes antivirus software, VPN security applications, and Windows Group Policy configuration specifications. Secondly, each system needs a base software set to execute its necessary test sequences. The primary component of this software is a set of drivers cross-checked with the published NI System Driver Sets. That is, one version of the test system might contain NI-DMM 14.0, NI-Switch 15.1, NI-FGEN 14.0.1, and NI-DAQmx 14.5 drivers. In addition, run-time engines for LabVIEW 2014 and TestStand 2014 are needed to run the main test system executable. The following chart outlines all the necessary software.

SOFTWARE	VERSION
NI-DAQmx Driver	14.5.0
NI-DMM Driver	14.0.0
NI-Switch Driver	15.1
NI-FGEN Driver	14.0.0
LabVIEW Run-Time Engine	2014
TestStand Run-Time Engine	2014
Internal AntiVirus Software	3.2

Table 3. When creating a base system image, it is important to explicitly list the versions of the drivers and run-time engines that will be included.

The first step of deploying to a new test system is to create this image on a development machine and replicate it using a hard drive imaging software. This software image may have been created previously, opening up the possibility to reuse an image across multiple machines. This helps reduce deployment cost as installation needs to be done only once per batch of identical test machines.

After the base system image has been generated by installing all of the necessary software, Symantec Ghost is used to replicate the hard drive and upload the new image onto a build server. The build server is located at headquarters and possesses the sole requirement of maintaining a large memory footprint for multiple system images to reside on the server.

2. Deploying the Base Image

After uploading the base image to the build server, the test engineer connects the new test system to the company network, and then uses a web interface to connect to the image server and browse the various base system images available for install. After selecting the appropriate version, Symantec Ghost images the new system's hard drive with the replicate image. At this point, the test system has the base necessary software it needs to execute test sequences.

3. Validating Hardware

After physically installing the necessary hardware modules to the PXI chassis and turning on the system, the test engineer needs to map the system aliases to the live devices in software. Although given a list of modules with associated slot numbers, the test engineer must use the configuration system setup to map the aliases so that module locations can change between systems. For this company, each test system uses a .ini file that the engineer edits to provide a mapping of live system hardware to test system aliases. This is done by identifying devices in MAX and manually editing the .ini file to create the appropriate map.

4. Installing the Application and Components

At this point, the test system has installed its base system image and validated its live hardware. Now, the engineer is tasked with installing the most recent version of the test application. In this case, the application is a TestStand installer, generated by the TSDU, that includes all of the necessary code modules, sequence files, and support files. To explain how this installer is generated, it is important to look at the development system employed by the production company. Each developer creates either a specific test step in LabVIEW or test sequence in TestStand and submits these to an Apache Subversion source code control repository. This repository is located on a server that is running a CI service, Jenkins. The Jenkins service is employed to run tests on submitted code modules, validate sequences with the TestStand sequence analyzer by command line, and then build the necessary test sequences into installers using the TSDU command line interface. After each installer is built, it is automatically deployed, along with its necessary support files, to a build server using the Jenkins Deploy Plugin.

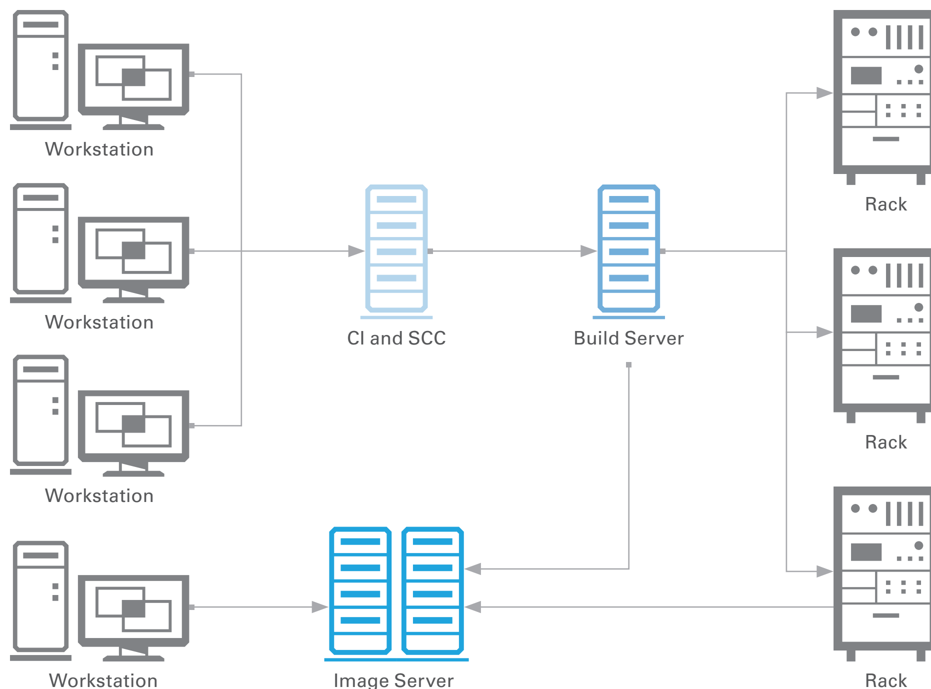


Figure 10. This test deployment system is using an image server to store and deploy base system images that keep the various test stations in sync with each other. Developers then regularly upload their source code to a continuous integration and source code control server that periodically builds and tests the submitted code. Once the submitted code passes all of the necessary tests, the built image is added to a build server that handles the large-scale distribution of the test software system image.

5. Executing

After the TestStand installer has been put on the build server, the test engineer can download the installer onto the new test system. The engineer can then run the installer, locate the main test executable, and begin running the base test system.

With this deployment system, the test engineer can quickly and easily make changes to each test system. The hard drive imaging system in place can be used for either large code revisions or driver set upgrades while the more lightweight build server can be used to deploy either small changes to the main test application or individual components and plugins.

Summary

Test System Deployment can often be a complex process especially as the complexity and number of the test systems scale. Establishing deployment processes early on in the development of a test system is the key to completing scalable and successful deployments. Ensuring that all of the necessary test system artifacts have been identified and defined and have a deployment method in place is the first step to creating a successful deployment process. Putting dynamic hardware configuration options in place can also be an important consideration of many deployment systems. For larger and more advanced systems, dynamically resolving dependencies between the deployment image and the target machine can help reduce both the complexity of the deployment process and the time required to upgrade or reimage a system. Managing and testing each release of a deployment image is another important consideration for test system developers. Whether this is done through a continuous integration service or configuration files, it is important to maintain a scalable release management system for distributed deployments. The deployment methods put in place for a test system will always be highly customized to the functionality and nature of the test system. The sections listed in this guide provide suggestions necessary for building a scalable solution, regardless of the tools used or the system's functionality.

TestStand Deployment Utility

The TestStand Deployment Utility simplifies the complex process of deploying a TestStand system by automating many of the steps involved in deployment, including collecting sequence files, code modules, and support files for a test system and then creating an installer for these files.

Learn more about the TestStand Deployment Utility

LabVIEW Application Builder Best Practices

LabVIEW Application Builder best practices make it simple to manage and organize LabVIEW applications. These recommendations help engineers to establish guidelines and procedures before beginning development to ensure that their applications scale for large numbers of VIs and multiple developers, saving development time and energy.

Start using Application Builder best practices for LabVIEW projects

©2016 National Instruments. All rights reserved. LabVIEW, National Instruments, NI, ni.com, and TestStand are trademarks of National Instruments. Other product and company names listed are trademarks or trade names of their respective companies.

NATIONAL INSTRUMENTS

FUNDAMENTALS OF BUILDING A TEST SYSTEM

System Maintenance

CONTENTS

Introduction

Concepts and Definitions

Design for Maintainability

Maintenance Strategies

Appendix: Cost of Maintenance

Introduction

In a perfect world, systems would never fail. You would set them up, turn them on, let them run, and never think about them until you are ready to stop using them in 10, 15, or 20 or more years. Unfortunately this is not reality, at least not yet. Systems fail and sudden, unexpected failures can be costly. Although you cannot completely remove the risk of failure, even with the most well-thought-out plans, you can reduce it. Maintenance strategies can help you to manage this cost and reduce the risk of failure.

A maintenance program is critical to ensure the lowest total cost of ownership across the life of an automated test equipment (ATE) system. A system designed for maintainability coupled with a sound maintenance program helps:

- Maximize capital investment by maintaining functionality and extending the useful life
- Minimize downtime costs by managing logistics, scheduling, and sparing inventory

The goal of any maintenance program is to keep the system working correctly for as long as possible, and get it back to working quickly if it stops. And, by the way, do this as inexpensively as possible.

Concepts and Definitions

Maintenance is the activity of preforming service to keep a system functioning and repairing a system if it fails. Maintenance is divided into three areas: predictive maintenance, preventive maintenance, and corrective maintenance.

Maintainability is the ease in which maintenance can be conducted. Some industries refer to it as serviceability. The better the maintainability, the easier it is to control maintenance cost.

Predictive maintenance uses condition monitoring to detect a system failure before it occurs and is something referred to in industry as condition-based maintenance. When a potential failure is predicted, maintenance activities are scheduled to service a system. These activities can extend the useful life and avoid unplanned downtime. Predictive maintenance activities are normally not scheduled until the need for maintenance is detected and result in planned downtime, which is typically much less costly than unplanned downtime. Planned downtime costs can be shared across many other systems receiving maintenance. The goal is to maximize the capital investment by using systems/components for as long as possible before a failure and minimize unplanned downtime costs. With the Internet of Things moving forward at an incredible pace, the concept of smarter machines that can monitor themselves and communicate to a network of other machines when they need maintenance is becoming the norm. Technology advances in sensors, embedded controllers, FPGAs, networks, and Big Analog Data™ analytics have made predictive maintenance easier and more cost-effective than ever. A measure of predictive maintenance is the downtime incurred; this time is referred to as the mean predictive maintenance time (MPdMT).

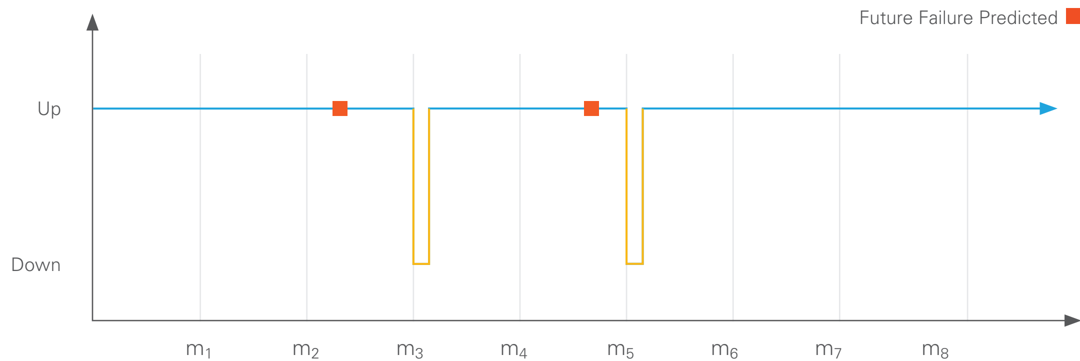


Figure 1. See predictive maintenance uptime and downtime over time. Predictive maintenance maximizes the use of your capital investment and minimizes downtime costs by lowering the frequency of downtime and converting expensive unplanned downtime to less expensive planned downtime, but requires failure monitoring equipment and prognostics software.

Predictive maintenance activities include:

- **Condition monitoring**—This ensures the system functions correctly, detects the onset of a failure, and identifies hidden failures in components or performance degredations that could lead to a system failure. With affordable embedded microprocessors and FPGA technology, built-in self-tests and conditioning monitoring techniques are commonly used. This is sometimes referred to as prognostics and health management (PHM) or system health monitoring. The concept is to detect performance changes and hidden failures in the system before they cause a much more serious system failure.

  Today, most cars have automated engine health monitoring that detects issues and flashes the check engine light, hopefully, in time to have the engine serviced before it is permanently damaged. A test system may monitor temperature, fan speed, memory usage, test times, measurement accuracy, count relay operations, and so on.

- **Servicing system components**—This helps to slow down wear and increase the useful life of the system.

  Some car tires have sensors that check the air pressure. Improper air pressure can shorten the life of tires and affect gas mileage performance. If a test system is used in a dusty environment, it may need to clean the dust from the air filters and the inside of the enclosure so it will not overheat and shorten the useful life of the electronics. Monitoring the internal temperature or airflow of the system can inform you to when you may need to clean dust filters.

- **Replacing system components**—Components are replaced before they fail to avoid unplanned downtime.

  A test system may use relays to switch signals for testing the device under test. Depending on the electrical load switched, a relay lasts for only an estimated number of operations. Therefore, monitoring the number of operations and replacing the relay modules before they fail is usually more cost-effective than waiting until a failure happens and experiencing an unplanned outage.

- Calibrating to compensate for drift—The purpose of a measurement system is to provide trusted measurements. If measurements are untrustworthy, then the system is functioning incorrectly.

  Most test systems contain electronics that need calibrating at some interval. If cutting-edge technology is used, however, the calibration interval may not be well understood, yet. Therefore, monitoring the measurement drift may be advised to understand when to properly schedule calibration maintenance.

- Verifying—This ensures the system functions correctly before bringing it back online. If it were brought back online only to malfunction, downtime would increase.

- Bringing the system back online—This must always be considered because, for some applications, it is not a trivial task.

  For example, if the test is part a manufacturing process, to bring this system back online may require stopping the line and resynchronizing the tester with the production flow.

Preventive maintenance is the activity of servicing a system to prevent a system failure and extend useful life. Preventive maintenance activities are normally scheduled and result in planned downtime. Planned downtime costs can be shared across many other systems receiving preventive maintenance. The goal is to minimize unplanned downtime costs. A measure of preventive maintenance is the downtime incurred; this time is referred to as the mean preventive maintenance time (MPMT).

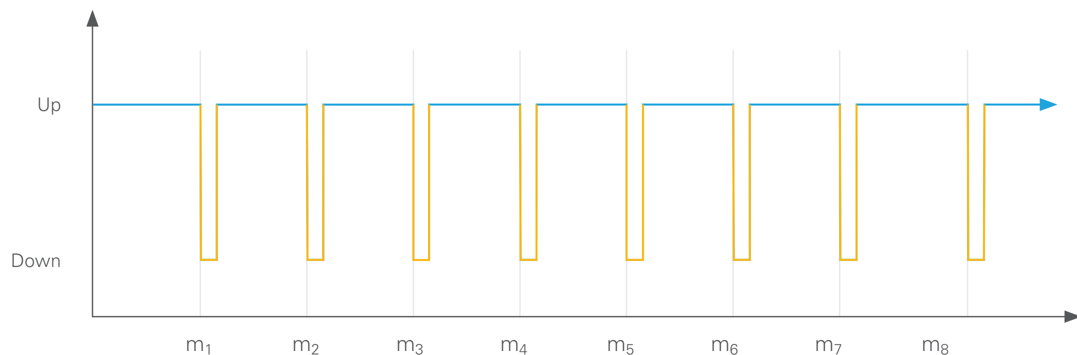


Figure 2. Preventive maintenance does not always maximize the use of your capital investment, but it helps to minimize downtime cost by avoiding expensive unplanned downtime.

Preventive maintenance activities include:

- Servicing system components—This helps to slow down wear and increase the useful life of the system.

  This is why a car's engine oil needs to be changed regularly. Test systems usually have complex software programs running in them that can have hidden resource leaks and or faults that eventually cause a system failure. A simple system reboot can refresh the software to a good-as-new state. If a test system is used in a dusty environment, it may need to clean the dust from the air filters and the inside of the enclosure so it will not overheat and shorten the useful life of the electronics. If the temperature and/or airflow cannot be monitored, then scheduling regular maintenance may be required.

- **Replacing system components**—Components are replaced before they fail to avoid unplanned downtime.

  The tires or break pads on a car are replaced at a certain mileage to avoid a failure that may cause an accident or strand the driver. A test system may have connector pins to test the device and they tend to wear out after 100,000 connections. If 50 devices are tested per hour, then the connector should last about 2,000 hours or 83 days before it wears out and fails. Preventive maintenance should be scheduled about every 80 days to replace the connectors. Replacing before a failure is usually more cost-effective than waiting until a failure happens and experiencing an unplanned outage.

- **Calibrating to compensate for drift**—The purpose of a measurement system is to provide trusted measurements. If measurements are untrustworthy, then the system is functioning incorrectly.

  Most test systems contain electronics that need calibrating at some interval.

- **Verifying**—This ensures the system functions correctly before bringing it back online. If it were brought back online only to malfunction, downtime would increase.

- **Bringing the system back online**—This must always be considered because, for some applications, it is not a trivial task.

  For example, if the test is part a manufacturing process, to bring this system back online may require stopping the line and resynchronizing the tester with the production flow.

Corrective maintenance is the activity of repairing a failed system to restore it to a functioning state. Corrective maintenance activities are usually not scheduled and result in unplanned downtime. The goal is to maximize the capital investment by using systems/components for as long as possible before a failure and after a failure to minimize unplanned downtime costs. A measure of corrective maintenance is the downtime incurred by a failure; this time is referred to as the mean time to repair (MTTR).

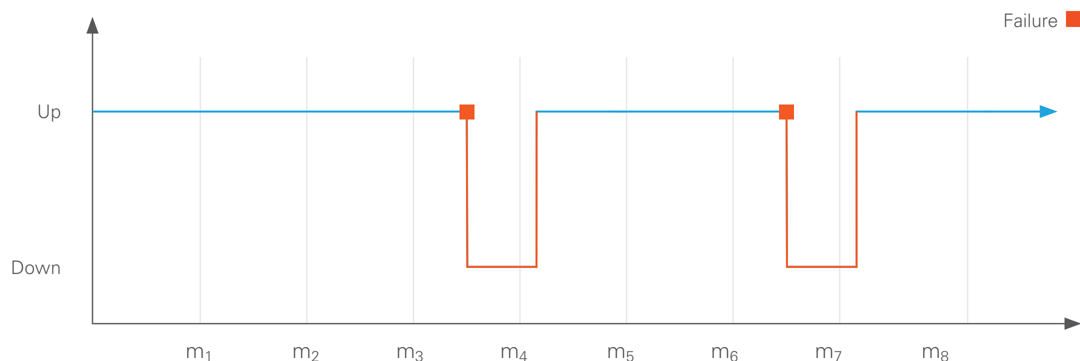


Figure 3. See corrective maintenance uptime and downtime over time. Corrective maintenance maximizes the use of your capital investment, but does not minimize the cost of downtime because it is unplanned. You can take steps to minimize the duration of the unplanned downtime or MTTR.

Corrective maintenance activities include:

- Detecting—Detecting a system failure as soon as possible minimizes costly unplanned downtime and possibly prevents damage to other components of the system and/or other systems that are used in the same process.

  Pressure sensors in a car can detect a drop in oil pressure as soon as possible to alert the driver and prevent permanent damage to the engine. Maybe the oil pump failed or the oil level is low because of a leak. It is much less expensive to repair an oil pump or seal a leak and add more oil than buy a new engine. For ATE systems, electronics may fail that can affect critical measurements and cause incorrect test results. If the failure took time to detect, a company could unknowingly ship bad products to customers, or a cooling fan could fail and chassis temperature may rise to a level that damages some of the electronics.

- Diagnosing and isolating—Diagnosing and isolating a failure correctly after it is detected can minimize unplanned downtime and save money by helping operators and maintenance personnel find and repair the correct component quickly.

  Automotive mechanics have sophisticated diagnostic equipment to help them diagnose problems efficiently and effectively. This saves time and money by lowering the risk of repairing or replacing the wrong component. The same can be said for complex ATE systems—hours and even days can be spent diagnosing a problem without proper diagnostic tools.

- Repairing—The system is repaired by repairing or replacing a failed component. The unplanned downtime is greatly impacted by having spares available. Depending on the application, environment, and skill level of personnel, having a spare system or spare components located nearby may or may not be cost-effective or practical.

  Most would not drive across the country without a spare tire in the car, but might if they need to drive only a few city blocks.

  A sparing strategy is essential to control costs. It is important to consider questions like, will the spares be kept on-site or in a nearby service center? Will you pay for the supplier to send an advanced replacement unit from the factory, or just wait until the failed unit is repaired and returned? The cost of unplanned downtime drives the answers. The number of units, the unit's mean time between failure (MTBF), and the time it takes to replenish the pool of spares determines the number of spares needed. Some companies provide levels of sparing services to assist with estimating the number of spares needed, helping with logistics, and managing sparing costs.

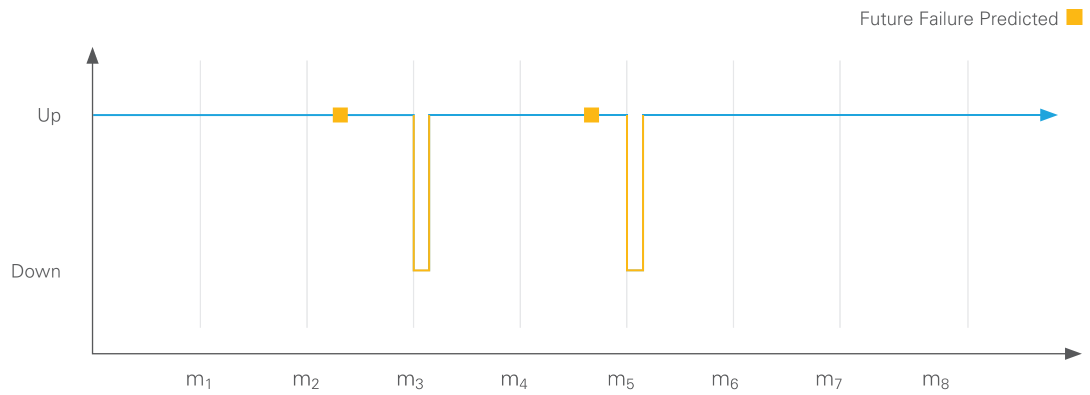
- Verifying—This ensures the system functions correctly before bringing it back online. Without this step, the system may still be functioning incorrectly and just cause more unplanned downtime.

  Imagine having the breaks on a car repaired, and then driving the car at high speeds on a freeway without first testing the breaks to verify they actually work.

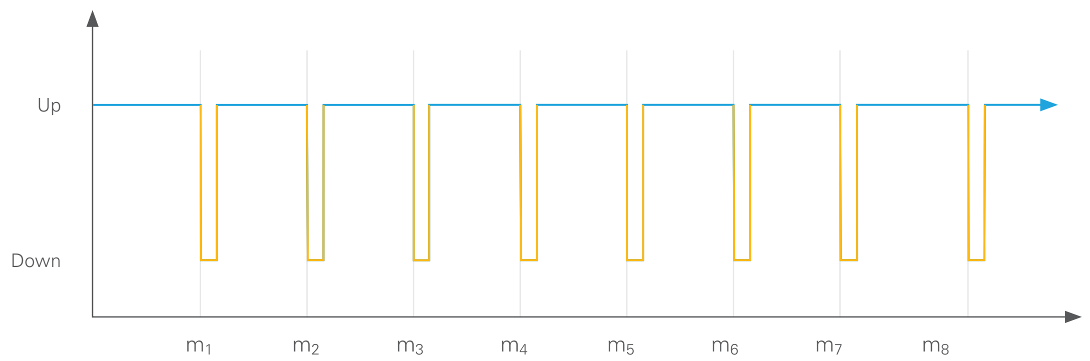
- Bringing the system back online—This must always be considered because, for some applications, it is not a trivial task.

  For example, if the test is part a manufacturing process, to bring this system back online may require stopping the line and resynchronizing the tester with the production flow.

PREDICTIVE



PREVENTIVE



CORRECTIVE

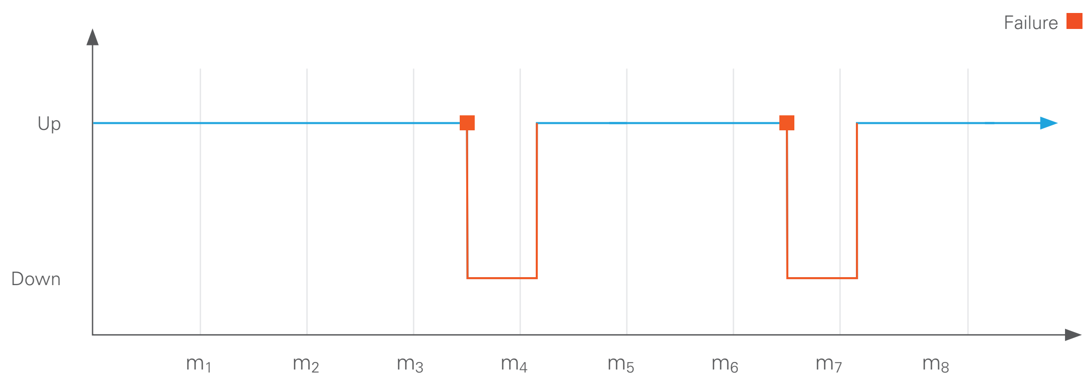


Figure 4. The cost of unplanned downtime is typically much more expensive than planned downtime as shown by the comparison of uptime and downtime.

Design for Maintainability

A system's design greatly affects the ability to achieve an effective, high-quality, and controllable maintenance program. When designing an automated test system for higher maintainability, consider the following best practices and guidelines.

Self-Test and Monitoring

Self-test and monitoring are essential for reducing downtime, planned and unplanned. Designing self-test and monitoring capabilities into the system from the ground up is key to having efficient and effective health monitoring, failure detection, failure diagnostics and isolation, and system verification.

Modular Designs

Modular designs simplify activities and reduce time associated with servicing, replacing, repairing, and calibrating system components. They also improve system diagnostics and failure isolation, saving valuable time during an unplanned outage. In addition, they reduce costs associated with spares. Instead of keeping several complete systems in the sparing pool, you can keep components, subsystems, or modules. Components usually have different failure rates—the components with lower failure rates need fewer spares, whereas those with higher rates need more.

Standardization

Standardization can greatly reduce costs because it simplifies logistics and reduces the number of spares, amount of maintenance tools and equipment needed, and training costs.

For example, some airlines employ 10 or more types of aircrafts in their fleet. Southwest Airlines, however, uses just one—the Boeing 737. This results in cost-savings. Mechanics need to be trained on and need spare parts inventory for only one type of airplane. They can swap out a plane at the last minute for maintenance. The fleet is totally interchangeable. All onboard crews and ground crews are already familiar with it. And, there are no challenges in how and where the planes can be stored, because they're all the same shape and size.

Standardization greatly helps to control the maintenance process. A well-controlled process is repeatable and predictable, thus designing a system with a maintenance process that can have only one way to complete the task is essential. If the Southwest Airlines maintenance crews all used different tools and conducted maintenance tasks differently, then each crew would produce different levels of quality and take different amounts of time to do the work, which makes it difficult to control and manage maintenance costs.

Simplicity

Keep it as simple as possible to operate and maintain. In other words, make it easy to do the right things. This reduces the amount of documentation and training required, improves the consistency of the work, and decreases the time needed to conduct maintenance.

Environment and Human Factors

Always consider the environment and human factors. For example, if the system is used in a dusty environment, it may need dust filters on the air vents. How easy will it be to service the filters? Does the system need castors so you can move it around for maintenance? If so, make sure they are appropriate for the weight and terrain. What is the skill level of the operators and maintenance personnel and how much training do they need? Can you design hardware and software interfaces in a user-friendly way?

DESIGN GUIDELINES	PREDICTIVE	PREVENTIVE	CORRECTIVE
Self-Test and Monitoring	▪ Condition monitoring ▪ Verifying functionality	▪ Verifying functionality	▪ Detecting failures ▪ Diagnosing and localizing failures ▪ Verifying functionality
Modular Design	▪ Condition monitoring ▪ Servicing ▪ Replacing ▪ Calibrating ▪ Verifying functionality	▪ Servicing ▪ Replacing ▪ Calibrating ▪ Verifying functionality	▪ Detecting failures ▪ Diagnosing and localizing failures ▪ Repairing ▪ Verifying functionality
Standardization	▪ Condition monitoring ▪ Servicing ▪ Replacing ▪ Calibrating ▪ Verifying functionality ▪ Improving consistency of work	▪ Servicing ▪ Replacing ▪ Calibrating ▪ Verifying functionality ▪ Improving consistency of work	▪ Detecting failures ▪ Diagnosing and localizing failures ▪ Repairing ▪ Verifying functionality ▪ Improving consistency of work
Simplicity	▪ Lowering documentation and training costs ▪ Improving consistency of work	▪ Lowering documentation and training costs ▪ Improving consistency of work	▪ Lowering documentation and training costs ▪ Improving consistency of work
Environment and Human Factors	▪ Lowering frequency of predictive maintenance events and/or the MPdMT ▪ Reducing human errors ▪ Improving safety	▪ Lowering frequency of preventive maintenance events and/or the MPMT ▪ Reducing human errors ▪ Improving safety	▪ Lowering failure rates and/or the MTTR ▪ Reducing human errors ▪ Improving safety

Table 1. This high-level summary shows how each design guideline benefits each maintenance approach.

Maintenance Strategies

Which approach should you use? Predictive strategies wait until a potential future failure is detected and then schedule service or replacement at a convenient time. Preventive strategies proactively service, replace, and/or calibrate system components on a regular scheduled interval to minimize the risk of failure and the cost of unplanned downtime. Corrective strategies wait until a component fails to maximize the usage of the capital investment and repair it as quickly as possible to minimize the cost of unplanned downtime, or minimize the MTTR. For each strategy, you can do it yourself, work out a service agreement with the suppliers, or do nothing and hope for the best when a failure happens, which is not recommended.

Here, see a combination of techniques that help explain which maintenance strategy is best to use for different subsystems or components. The approaches discussed are condition monitoring feasibility, reliability centered maintenance (RCM), and cost of failure analysis. RCM is based on having an understanding of the affect of runtime on the failure rate of system components and the cost of component failures. The failure rate as a function of runtime is shown in the three graphs below. Each graph depicts characteristics for different types of components. There are more scenarios than these three, but these are common ones that help demonstrate how RCM works.

Figure 5 shows the failure rate increasing overtime. In this situation, the component's failure rate may appear constant at first but starts to enter wear out well before the intended service life of the system. In other words, the useful life of the component is significantly shorter than the length of time the system will be in service. This is probably the most intuitive scenario because mechanical components like fans, connectors, electromechanical relays, solid-state hard drives, batteries, the calibration of electronics, and so on have this trend. After each preventive maintenance event, the failure rate is lowered back to a "good-as-new" level, thus restoring the reliability of the system.

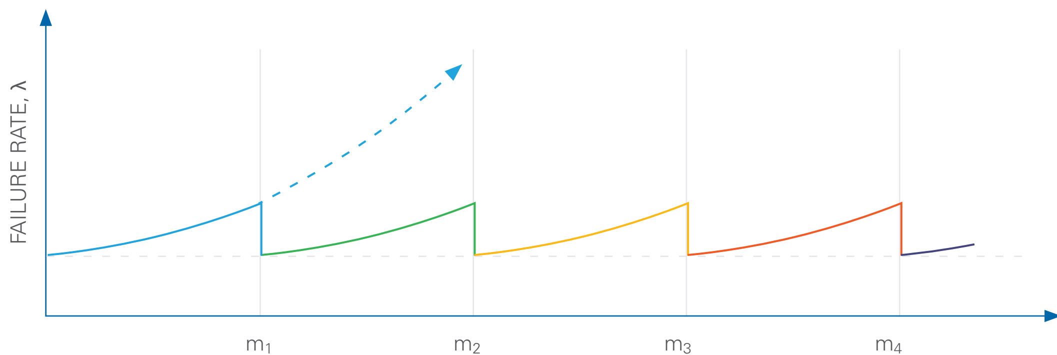


Figure 5. Preventive maintenance lowers the failure rate back to a "good-as-new" level at each maintenance event when the failure rate is increasing.

Figure 6 shows the failure rate remaining constant over time, sometimes called the steady-state failure rate. In this situation the component should not start to wear until well past the intended service life of the system (this does not include calibration). In other words, the useful life of the component extends well beyond the length of time the system will be in service. This is a typical scenario for electronic components such as ICs, resistors, ceramic capacitors, diodes, inductors, and so on that are in useful life. Modern electronics typically have a useful life well beyond 10 to 15 years. For all practical purposes, they do not wear out before the test system is obsoleted.

After each predictive maintenance event, the failure rate is not changed, so there is no benefit to replacing a component before it fails. Mathematically, this failure rate is treated as a "random chance". Therefore, replacing an older functioning component with a new component does not improve the system reliability.

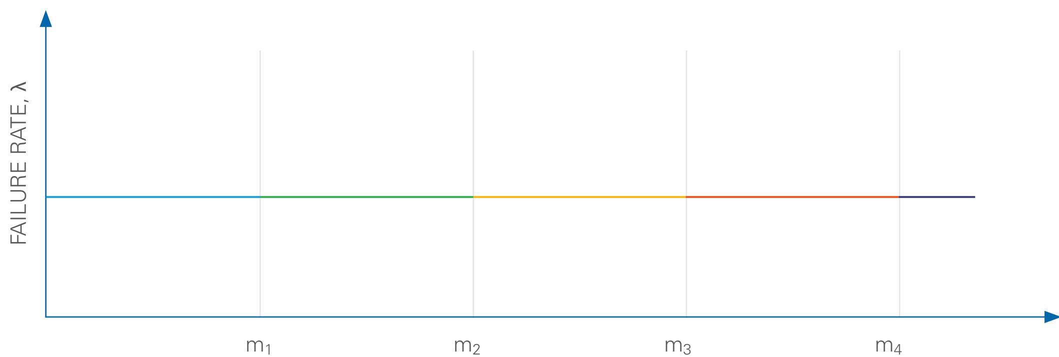


Figure 6. Preventive maintenance has little to no effect on the failure rate at each maintenance event when the failure rate remains constant over time.

Figure 7 shows the failure rate decreasing over time. This is probably the least intuitive scenario, but software and complex computer systems can exhibit this characteristic. Performing major upgrades to software and firmware or adding new features, new technology, and so on can introduce defects (bugs) that increase the probability of a system failure. After each preventive maintenance event, the failure rate is raised to a higher level, thus decreasing the system reliability. However, situations arise where you must upgrade software, such as OS updates or hardware obsolescence.

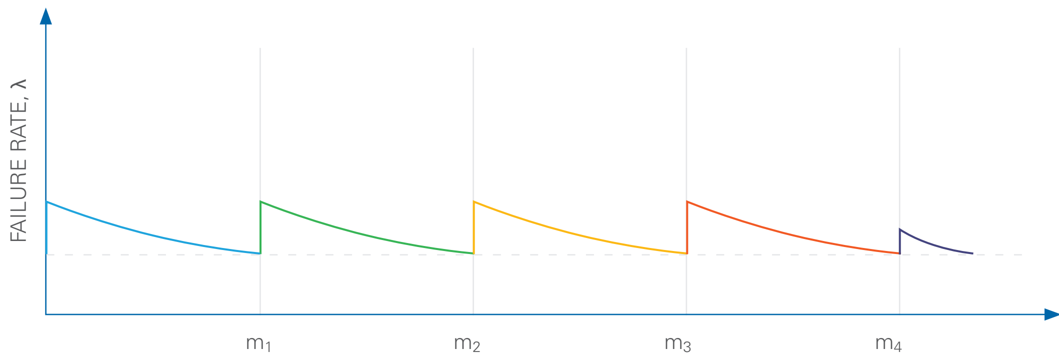


Figure 7. The failure rate decreases over time and the preventive maintenance actually raises the failure rate at each maintenance event.

In addition, there are situations when there is insufficient data to know whether the failure rate is increasing, constant, or decreasing over time. This may be the situation for new products, technologies, or designs. Using a predictive maintenance strategy of monitoring for failures over time can provide good insight into what the situation for a component might be if the cost of monitoring is effective compared to the cost of a failure. Even if a trend is not established, a predictive strategy usually maximizes your capital investment and minimizes downtime costs.

When using this approach to develop a maintenance strategy for a complete system, you can break down the system into subsystems and/or components, and then evaluate each component to see which maintenance strategy is best. The following are some good guidelines to work with:

- Can the onset of a component failure be detected before it causes a system failure?
  - Is it cost-effective to implement condition monitoring for this component failure, considering the cost of failure of a corrective maintenance event and the extra planned downtime from a predictive maintenance event?
- Is the failure rate of this component increasing, continuous, or decreasing over time or do you know?
  - Is the failure critical and the cost of a failure high?

The diagram below shows a decision flow chart to help you choose the best strategy for each component and failure mode of the system. The flow chart, however, should not override good human judgment.

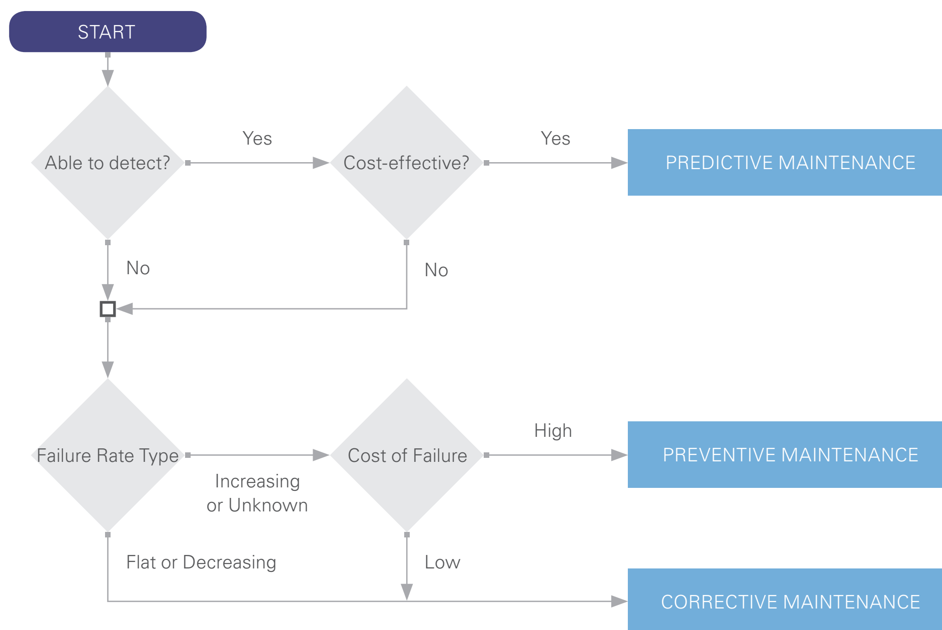


Figure 8. Maintenance Strategy Decision Flow Chart

Example ATE System

An ATE system based on PXI Express is made up of basic components or subsystems that can each be broken down into smaller components for a maintenance strategy of their own.

Chassis

The chassis backplane can be challenging to monitor for potential failures. It has a constant failure rate and should have a service life of 10 to15 or more years. The electronics are basically all digital and do not require calibration. A corrective maintenance strategy or a "run to failure" is the best approach.

The chassis power supply in this example does not provide monitoring capabilities. Power supplies typically use larger liquid filled capacitors and some may have their own cooling fans. These components have a typical service life of around seven to 10 years, depending on load and environmental conditions. A predictive or preventive maintenance strategy is a good approach.

The chassis fan speed and the chassis temperature can be monitored. If the speed starts to slow down or, if the chassis temp starts to increase, a warning can be sent and maintenance can be scheduled at a convenient time in the near future. A predictive maintenance strategy works well.

Controller

The controller's integrated circuits and electronic components (excluding hard disk and RAM) may provide tools to monitor for potential failures. For this example, it would require a lot of development time to implement these features and not be cost-effective. It also has a constant failure rate and should have a typical service life of 10 to 15 or more years. The electronics are basically all digital and do not require calibration. A corrective maintenance strategy or a "run to failure" is often the best approach.

The controller's RAM has an error correction code (ECC) that automatically runs and the amount of errors that are found and corrected can be monitored. If the frequency of these errors continue to increase, time to replace the RAM may need to be scheduled. RAM does not require calibration. A predictive maintenance strategy is the best approach.

The controller's hard drive in this example is a solid-state hard drive (SSD) that monitors the number of reads and writes. SSDs have only a certain number of reads and writes before they wear out. Thus, as the number of reads and writes approach the wear out numbers, the SSD should be scheduled for replacement. SSDs do not require calibration. A predictive maintenance strategy is the best approach.

Software has some unique characteristics—it does not wear out and is unaffected by the environment. It fails only because of design defects or bugs. Resource leaks, like memory usage and fragmentation, can be monitored; however, many faults cannot be seen before the crash. The wonderful aspect about software is that because it does not wear out, a simple reboot of the system is like starting fresh and good as new. This fixes everything until the bug raises its ugly head again. Therefore, a preventive maintenance strategy of rebooting the software once a week or once a month may solve many problems. A more challenging aspect to software reliability is upgrading. Software requires upgrades occasionally as new features are required, compatibility with other software packages is required, or perhaps a patch to fix a bug is needed. The challenge is that every time you introduce new software, it changes the ecosystem and may introduce more bugs. You don't know until you try. This dynamic makes the risk of failure for software go up immediately after a software upgrade, and then settle down after some runtime. The upgrade maintenance approach most commonly used for software is to delay an upgrade until it is necessary.

Instrument Modules

The integrated circuits on instrumentation modules can be challenging to monitor for potential failures. They have a constant failure rate and should have a service life of 20 or more years. The analog electronics can drift over time, thus they require calibration. A preventive maintenance strategy to address calibration is required to address drift. Many calibration labs can run a final verification test on the module after calibration to prove all is well. This test does a good job catching other electronic components that have failed or are failing. But no test is perfect and a corrective maintenance strategy or a run-to-failure approach may be best for some of the other failure modes of the electronics. There, a combination strategy is the best approach.

Switch Module

The switch module's base board is primarily made up of integrated circuits that usually do not have tools to monitor the health of the electronics. They have a constant failure rate and should have a typical service life of 10 to 15 or more years. The electronics are basically all digital and do not require calibration. A corrective maintenance strategy or a run-to-failure approach is best.

The switches' electromechanical relays have tools that monitor the number of operations. Relays have only a certain number of operations before they wear out, depending on the electrical load that is switched. You can estimate the number of switches by using data and formulas that the manufacturer provides. Thus, as the number of operations approaches the wear out numbers, the switch module should be scheduled for replacement. Switch modules do not require calibration. A predictive maintenance strategy is the best approach.

Cables

Fixed cables are basically connected and never disconnected, or reconnected so infrequently that it does not matter. Fixed cables hardly ever fail except form vibration or human abuse. The failure rate is constant and very low. A corrective maintenance strategy is the best approach.

Dynamic cables are connected and disconnected frequently and wear out after a certain number of reconnects. The failure rate is increasing over time and detecting a potential failure may not be easy, but it may be estimated. The number of reconnects may be understood by the manufacturer and is worth asking about. If the average number of reconnects is known and you know how many reconnects there will be per hour, per day, per unit, and so on, then you can schedule preventive maintenance. A preventive maintenance strategy is the best approach.

SUBCOMPONENT	PREDICTIVE	PREVENTIVE	CORRECTIVE
Chassis Backplane	—	—	√
Chassis Power Supply	—	√	—
Chassis Fans	√	—	—
Controller Mother Board	—	—	√
Controller RAM	√	—	—
Controller Solid-State Drive	√	—	—
Controller Software	—	√	—
Instrument Module	—	Calibrate	√
Switch Module Base Board	—	—	√
Switch Module Relays	√	—	—
Fixed Cables	—	—	√
Dynamic Cables	—	√	—

Table 2. You could use this maintenance strategy for each major component of an ATE example based on PXI Express. Note that the best strategy for each component is dependent on the unique situation for your application.

Conclusion

Each predictive, preventive, and corrective approach has its benefits, challenges, and appropriate situation. In most situations, the greatest expense associated with maintenance is the cost of unplanned downtime (the cost of a failure). Converting unplanned downtime to planned downtime through the use of condition monitoring and prognostics is usually advantageous.

Every year, condition monitoring equipment, networks, servers, and Big Analog Data™ analytics continue to decrease in cost and increase in performance, thus industry is trending toward smarter equipment and more predictive maintenance. For the situations when unplanned downtime is unavoidable, a good sparing and repair strategy is key to minimizing and managing maintenance cost.

A system designed for maintainability from the ground up coupled with good maintenance strategies will help you manage costs and reduce the risk of failures that lead to expensive unplanned downtime. This lowers the cost of maintenance, which lowers the total cost of ownership. Self-tests, modular designs, standardization, simplicity, and environmental/human factors are fundamental building blocks when designing for maintainability.

Appendix: Cost of Maintenance

Many companies base purchasing decisions for test equipment primarily on the price and do not consider the cost of deploying, operating, and maintaining the equipment. And they even less frequently consider the cost of equipment downtime. The cost of downtime (or failures) and maintenance over the service life of a test system can be much greater than the purchase price, frequently reaching two to three times more. The largest culprit is the cost of downtime or a failure. This is why maintenance programs exist and the maintainability of a system is becoming more important every day.

This appendix provides a straightforward total cost of maintenance (TCM) model that can be used to estimate the potential downtime and maintenance costs of a system over its service life. Calculating the TCM of a test system can become very tedious and complex quickly. This model provides a sufficient estimate at a level of complexity and detail that is adequate and manageable for most applications.

Total Cost of Maintenance (TCM)

$$TCM = CD + M$$

CD = Cost of Unplanned and Planned Downtime

M = Cost of Maintenance

You can measure a maintenance program's return on investment (ROI) by comparing maintenance dollars (M) invested to the reduction in downtime dollars (CD) spent over the service life of the system or to the overall reduction of TCM dollars over the service life of the system. Some companies combine the cost of planned downtime with the cost of maintenance and compare this only to the cost of unplanned downtime, because their main focus is to avoid unplanned downtime and failures. Each company may have its own way to estimate TCM and the ROI of maintenance, depending the metrics a company would like to track.

Cost of Downtime (CD)

Downtime costs can sometimes seem like "funny" money because some companies find them difficult to estimate. But the cost of downtime is very real. There are two types of downtime: planned (scheduled) and unplanned (unscheduled). The goal of a maintenance program is to minimize all downtime and convert as much unplanned downtime to planned downtime as financially feasible.

Unplanned downtime is always the most expensive, because it takes place when you need the equipment. It is never at a good time and can result in lost revenue from loss of production, product loss, collateral damage to other equipment, labor loss (the labor force may have to "'sit' around and" wait for the system to be repaired), and other logistical costs that are situation dependent. Some manufacturing companies estimate their cost of unplanned downtime to be around $8,000 per hour. Petrochemical, power, and transportation companies often estimate much higher cost per hour. This cost is different for various products, situations, companies, and industries. Time is money; this is why corrective maintenance plans with sparing strategies are put in place to minimize the mean time to repair (MTTR) of a failed system.

Planned downtime is costly as well, but less expensive than unplanned downtime because it is scheduled for times that will have the least impact on production, minimizes product loss, minimizes the risk of collateral damage to other equipment, results in no labor loss, and minimizes the cost of logistics (trained people, tools, and parts are on-site and ready to perform the maintenance). Planned downtime can be shorter than an unplanned outage and is shared across many other systems that need maintenance. Because unplanned downtime is usually much more expensive than planned, many companies put into place predictive and preventive maintenance plans.

$$CD = UD + PD$$

UD = Cost of Unplanned Downtime
PD = Cost of Planned Downtime

$$UD = \lambda \times MTTR \times T_U \times \text{Cost per Hour}$$

λ = Steady-State Failure Rate (failures per hour)

The steady-state failure rate of the system is the failure rate that is expected during the system's service life or it's useful life. This is the phase of life between early life (system burn in) and the wear-out phase of life when the system failure rate is expected to significantly increase and the system should be retired. During the service life phase of the system when the failure rate is considered to be steady state, this mathematical relationship can be used.

$$\lambda = \frac{1}{MTBF_{System}}$$

$MTBF_{System}$ = Mean Time Between Failure of the System (hours)

T_U = Total Amount of Run Time of the System During the Service Life (hours)

Run time for electronics usually includes the time that the system is powered on while doing its job and in an idle state.

MTTR = Mean Time to Repair (hours)

MTTR is not just the time to repair or replace a failed component. It includes the:

- Time to detect a failure
- Time to diagnose the system and understand which system component(s) failed
- Time to access and repair or replace the failed component(s) (having spares and/or redundancy will greatly impact this)
- Time to verify the system is repaired correctly
- Time to bring the system back online

It is easy to see that MTTR is very dependent on having spares, the system location, the design, and the type of failures that typically occur.

$$MTTR = \frac{\Sigma(\lambda_i t_i)}{\Sigma \lambda_i}$$

λ_i = Failure Rate for the ith Failure Mode

t_i = Time to Repair the System After the ith Failure Mode Occurred

The failure mode is defined as the type of failure that occurred or the root cause of the failure.

$$PD = (\lambda \times MPdMT + f_{PM} \times MPMT) \times T_U \times \text{Planned Downtime Cost per Hour}$$

The frequency that predictive maintenance should occur should correlate the to failure rate of the system. Instead of performing maintenance after a failure has occurred, maintenance is performed at a planned time after a potential failure condition is detected but before the failure occurs.

MPdMT = Mean Predictive Maintenance Time (hours)

MPdMT includes the:

- Time to access
- Time to service and or replace component(s) (having spares and/or redundancy will greatly impact this)
- Time to verify the system is operating correctly
- Time to bring the system back online

$$MPdMT = \frac{\sum(\lambda_i t_i)}{\sum \lambda_i}$$

λ_i = *Frequency of the ith Predictive Maintenance Activity*

t_i = *Time to Conduct the ith Predictive Maintenance Activity on the System*

f_{PM} = *Frequency of Preventive Maintenance (per hour)*

MPMT = Mean Preventive Maintenance Time (hours)

MPMT includes the:

- Time to access
- Time to service, replace, and/or calibrate component(s) (having spares and/or redundancy will greatly impact this)
- Time to verify the system is operating correctly
- Time to bring the system back online

$$MPMT = \frac{\sum(f_i t_i)}{\sum f_i}$$

f_i = *Frequency of the ith Preventive Maintenance Activity*

t_i = *Time to Conduct the ith Preventive Maintenance Activity on the System*

Cost of Maintenance (M)

$$M = PdM + PM + CM$$

PdM = Cost of Predictive Maintenance
PM = Cost of Preventive Maintenance
CM = Cost of Corrective Maintenance

Cost of Predictive Maintenance (PdM)

$$PdM = \lambda \times T_U \times PdM\ Event + Cost\ of\ Tools$$

PdM Event = Average Cost of a PdM Event

PdM Event = (MPdMT x Planned Downtime Labor Cost per Hour) + Service or Replacement + Spares + Logistic Cost

Planned downtime labor cost includes the cost of labor to perform predictive or preventive maintence for a system and the cost of training the labor force estimated on a per hour basis.

Cost of Tools = Cost of Software and Hardware Tools Needed for PdM

The cost of tools is typically a one-time expense that includes the cost of:

- Condition monitoring hardware and software
- Tools to remove and replace components
- Verification test equipment and software (which could be the same used for corrective maintenance)
- Maintenance of the equipment and software

NOTE: The tools can often be used for predictive, preventive, and corrective maintenance. If the tools can be used, then the cost of tools needs to be accounted for only one time and not for all three types of maintenance.

As shown above, MPdMT is greatly affected by having the right equipment/tools available and well-trained personnel.

Cost of Preventive Maintenance (PM)

$$PM = f_{PM} \times T_U \times PM\ Event + Cost\ of\ Tools$$

f_{PM} = Frequency of Preventive Maintenance (per hour)
PM Event = Average Cost of a PM Event
PM Event = (MPMT x Planned Downtime Labor Cost per Hour) +
Calibration, Service or Replacement + Spares + Logistic Cost

The smaller the MPMT of a system is, the lower the cost of predictive maintenance. As shown above, MPMT is greatly affected by having the right equipment/tools available, well-trained personnel, and a good calibration strategy. Many system suppliers can offer calibration options. Depending on the situation, it may be more cost-effective to have on-site calibration services or purchase a calibration service agreement from the system supplier. A standard supplier calibration program may be sufficient.

Cost of Corrective Maintenance (CM)

$$CM = \lambda \times T_U \times CM\ Event + Cost\ of\ Tools$$

CM Event = Average Cost of a CM Event
CM Event = (MTTR x Unplanned Downtime Labor Cost per Hour) +
Repair or Replacement + Spares + Logistic Cost

Unplanned downtime labor cost includes the cost of labor to repair a system and the cost of training the labor force estimated on a per hour basis.

The smaller the MTTR of a system is, the greater the system availability and the lower the cost of unplanned downtime. As shown above, MTTR is greatly affected by location, the system design, having the right equipment/tools available, well-trained personnel, and a good sparing strategy. Many system suppliers can offer sparing options. Depending on the situation, it may be more cost-effective to own spares or purchase a service agreement from the system supplier to provide spares or have some hybrid agreement of the two. If the cost of unplanned downtime is low enough, on-site spares may not be justified and relying on standard supplier repair times may be sufficient.

©2016 National Instruments. All rights reserved. Big Analog Data, National Instruments, NI, and ni.com are trademarks of National Instruments. Other product and company names listed are trademarks or trade names of their respective companies.