FUNDAMENTALS OF BUILDING A TEST SYSTEM

Software Deployment

CONTENTS

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.

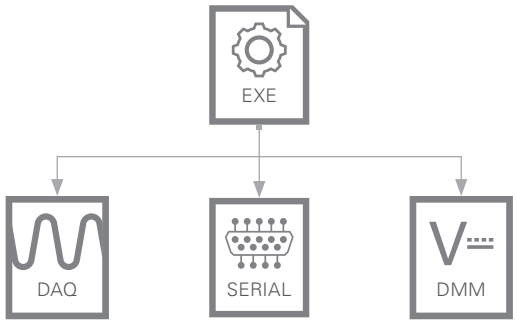

Figure 1. Simple Representation of a Test System Executable That Depends on a DAQ, Serial, and DMM Driver

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.

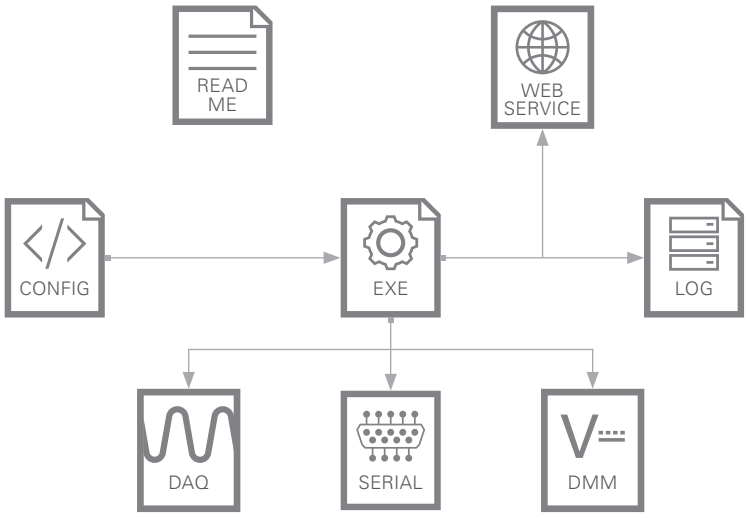

Figure 2. Example of a Test System With Complex Dependencies

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.

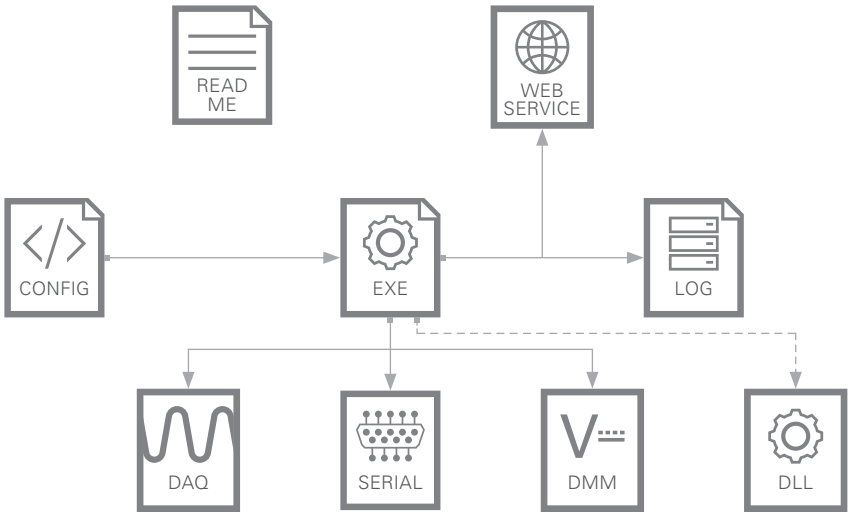

Figure 3. Unexpected Dependencies in a Complex Test System

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.

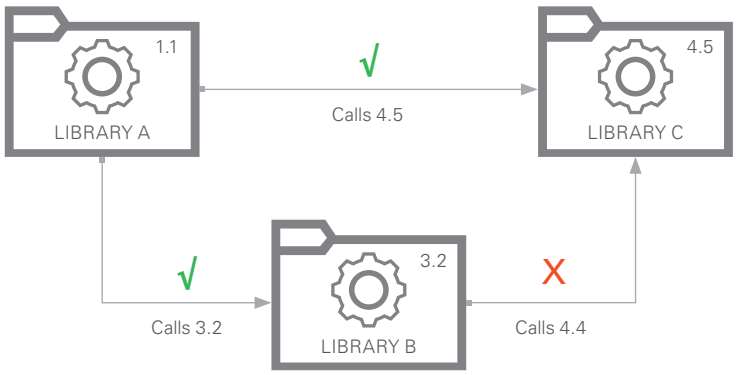

Figure 4. Library B's reliance on version 4.4 of Library C causes an unsolvable dependency issue as Library A relies on version 4.5 of Library C.

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.

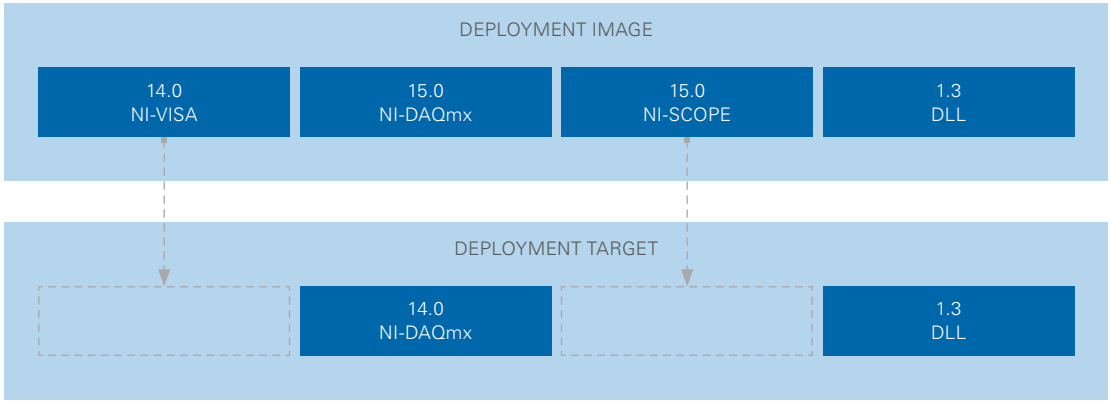

Figure 5. Dependency Assertion

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.

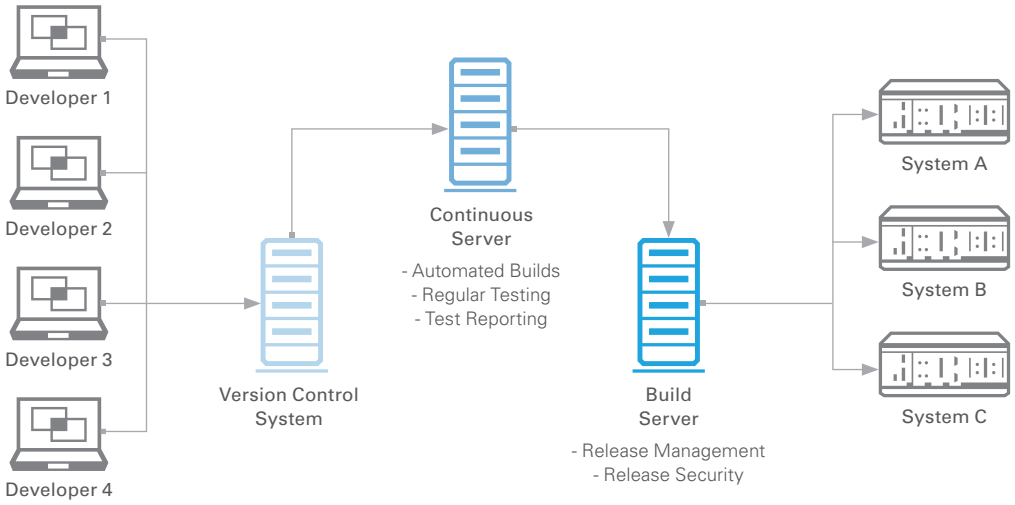

Figure 6. Developers submit code to a version control repository that can then be built and tested in a CI server. From there, the builds can be stored in a build server and undergo release management.

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.

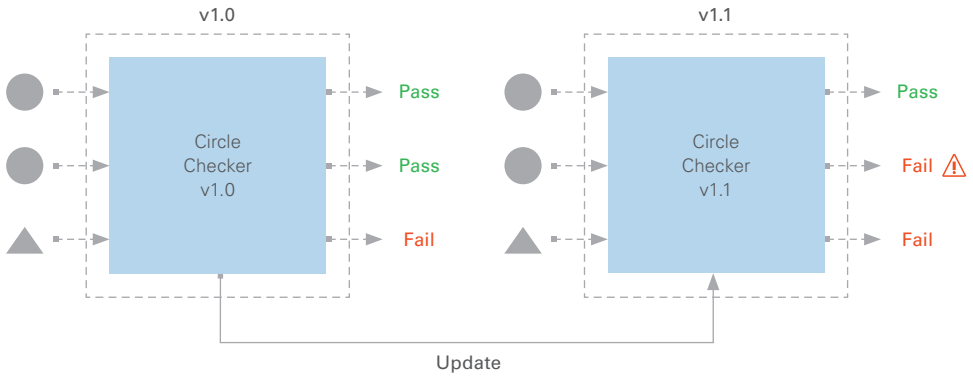

Figure 7. A small update to a module in the test system can cause the functional test to malfunction, resulting in false failures.

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.

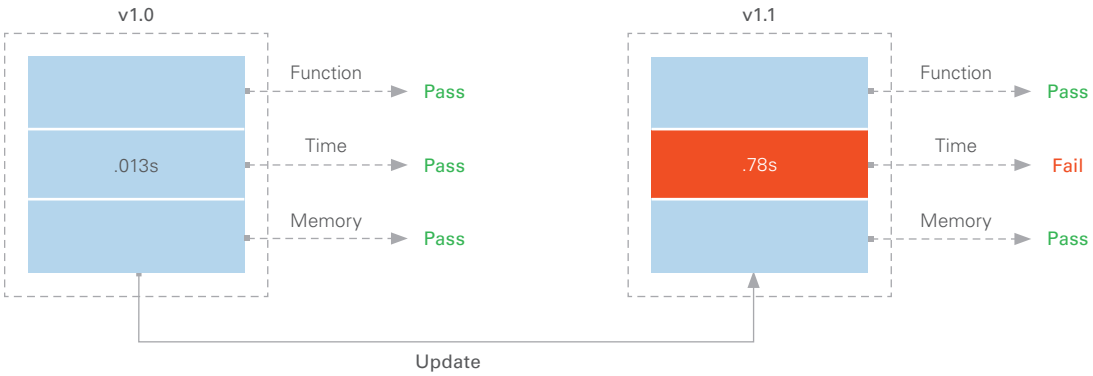

Figure 8. After performing a unit test of both the v1.0 and v1.1, the processing time is identified as a problem after the update, causing false failures in the process functional test.

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.

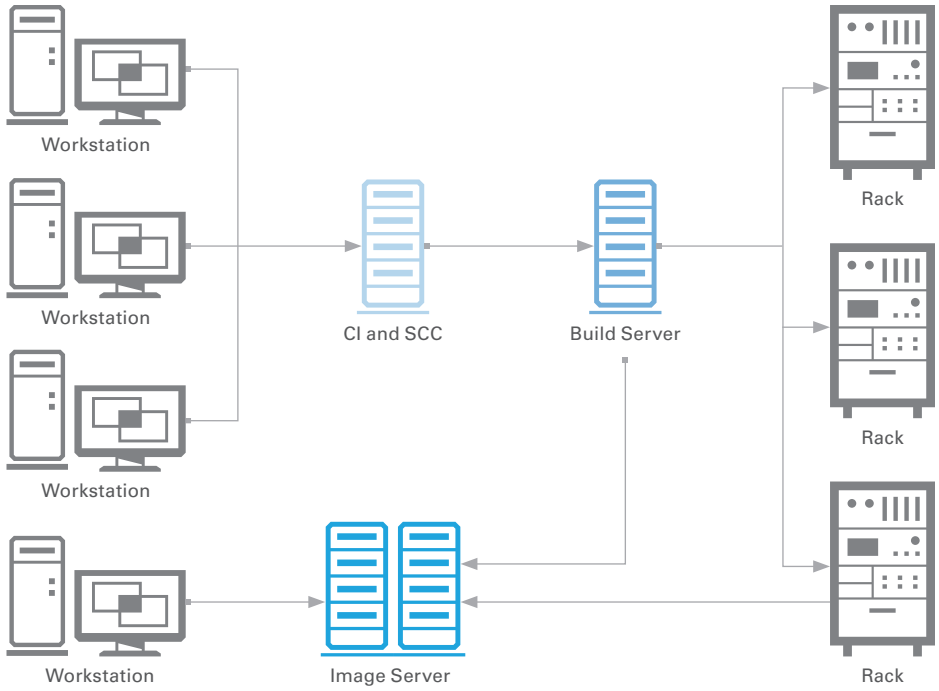

Figure 10. This test deployment system is using an image server to store and deploy base system images that keep the various test stations in sync with each other. Developers then regularly upload their source code to a continuous integration and source code control server that periodically builds and 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