A Guide to Developing Effective, Maintainable TestStand Sequences

Updated Feb 4, 2020

Overview

This application note suggests some guidelines for developing effective sequences and test code in TestStand. Using these techniques to develop sequences, you can achieve faster overall sequence development, reuse of more test code, and a highly scalable overall test system.

Contents

Sequence Organization

A TestStand sequence is typically used by a number of people, such as engineers working on a shared test project, operators running the sequence on a deployed system, and potentially developers on next-generation products years in the future. Thus, it is important to establish standards for organizing a sequence to make it readable, reusable, and maintainable.

Isolate Nonproduct-Specific Operations in a Callback or the Process Model
In general, the MainSequence in a TestStand sequence file should contain only the tests for the product or product family you are testing. Communicating with a handler, scanning a serial number, and generating a test report, for instance, should be done in either a sequence file callback or a TestStand process model. This practice results not only in a more readable sequence, but also in one that can easily be reused and maintained because it does not contain any code that ties it to systems that might change. If these elements of the test system do change, you can modify them in the callback or process model sequence rather than in each test sequence. For more information on callbacks and process models refer to the *Use of the Process Model* section.

Using the Setup and Cleanup Step Groups
Each sequence in TestStand has the following three step groups -- Setup, Main, and Cleanup. During normal execution, these step groups execute sequentially in that order. Within each MainSequence, it is a good idea to use the Setup step group to contain your initialization steps and the Cleanup step group to contain your error-handling and shutdown routines. In this way, the Main step group consists primarily of test steps rather than housekeeping operations. The Cleanup step group is particularly useful, because if a run-time error occurs in any step, the execution skips immediately to the Cleanup step group. This ensures that your error-handling and shutdown steps execute regardless of where the error occurs.

Setup steps and Cleanup steps run even in an interactive execution, which you initiate by selecting the Run Selected Steps or Loop on Selected Steps command. The Setup step group runs first, followed by the steps you select, followed by the Cleanup step group. In this way, the necessary initialization and shutdown operations occur automatically before and after the selected steps. If you do not want the Setup and Cleanup step groups to run in an interactive execution,

you can disable this feature in the Execution tab of the Station Options dialog box.
The steps you place in the Setup and Cleanup step groups should be specific to the product or subcomponent sequence. More general setup and cleanup routines should be handled in a process model or a sequence file callback.

Modularize Test Steps into Subsequences
You should organize the steps in each MainSequence into logical groups of tests. If several tests are often together, it is appropriate to encapsulate these steps into a subsequence you can reuse in other sequences. Use the parameters of the subsequence to pass inputs and outputs. By specifying values as parameters rather than as explicit values in the subsequence, you can make your subsequence more generic and thus reusable. Furthermore, because each subsequence can have its own Setup and Cleanup step groups, modularizing your test steps into subsequences is beneficial because you can specify setup and cleanup routines that apply only to a subset of the product tests. It also provides you with more places to handle errors and ensure that the system shutdown is graceful, if an error occurs.

Another benefit of grouping steps into subsequences is that you can dynamically choose at run time which subsequence to call. For example, if three versions of the same product differ in only a few tests, you can make a single top-level testing sequence that runs all the common tests and then dynamically runs a subsequence that has the tests specific to each version. Thus, you create as few sequences as possible with little or no duplication of test steps between sequences. Reducing the number of sequences and minimizing duplication makes maintenance easier because you can modify a particular test step in only one place, and because you can make changes to the component-specific tests without affecting the other components in any way.

Share Data Using Local Variables, Global Variables, and Step Properties
TestStand gives you the freedom to share data among sequences at several different levels. With local variables, you can share data among the steps within a single sequence. With sequence global variables, you can share data among all of the sequences in a sequence file. With station globals, you can share data among all of the sequence files on one computer. With step properties, you can associate a variable with a particular step and still make it available to other steps in the sequence.

Knowing when to use local variables, global variables, and step properties is an important consideration in creating an effective sequence. For sharing data between steps, sequences, or files, use local variables, global variables, or station globals, respectively. If you want to pass information to, or return information from, an individual TestStand step, consider using a step property. If you require a unique variable for each instance of a step, you should probably create a step property instead. To add a property to a step, you must first create a custom step type for the step. Refer to the *Custom Step Types* section for more information.

Use Custom Data Types to Represent Common Sets of Data
For common sets of data that you need to store, consider creating a custom data type in the Types Palette. For example, you might create a data type called phone that contains a number to represent a frequency band, a string to denote model type, and an array of Boolean values for the numeric keypad. You can then create variables and properties of type phone. For example, if at a later time you need to change the definition of the phone data type to add another property, TestStand automatically updates all instances of the phone data type to include this change.

Document the Sequence
Finally, just as with any code, a sequence should be well documented. In TestStand, you can document a sequence with comments and labels. Many objects in TestStand, including variables, steps, sequences, and sequence files, have a comments field. You can access the comments field of each object in its Properties dialog box. Use the comments field to describe the purpose of the object to future developers and to document the changes you make. Use Label steps to make a sequence more readable. You can place a label anywhere in a sequence. Label steps are also useful as a target for a Goto step when implementing a loop within a sequence. Using a label as a target, you can insert additional steps into the beginning of the loop at a later time without having to change the target in the Goto step.

Use of a Process Model

TestStand process models provide architectures for customizing your test system to the specific needs of your production line or design verification laboratory. To promote a maintainable architecture, it is important to use a process model correctly. The purpose of a process model is to provide a test system development framework that includes all things that remain the same from one product to the next, such as interfacing with the product handler, generating reports, and communicating with a corporate database. Thus, a process model provides connectivity to the outside world and helps enforce corporate standards for data storage and report generation. In general, you use a single process model on a test system or group of similar test systems.

In designing a process model, it is useful to group your product test routines into the following three categories:

- Product-specific tests
- Company or group-wide standards
- Product-line-specific routines

A simple example makes this clearer. Final testing of a mobile phone requires a suite of tests to make measurements on components such as the LCD screen, the keypad, the antenna, and the battery. These tests likely differ, at least to some degree, for each model of mobile phone and thus are *product-specific tests*. In addition to testing each of these subsystems for each phone, the test executive must run many other routines, such as routines to scan in a serial number, communicate with the unit handler, generate test reports, and communicate with a corporate database. These types of routines are generally kept constant from one product to the next and are thus *company* or *group-wide standards*. Finally, some routines might be more or less standard among most products but different for only certain models. For example, one set of phones might require a slightly different test report as specified by a customer. These deviations from the company standards are categorized as *product-line-specific routines*.

As a general rule, you should implement product-specific tests in the MainSequence of a TestStand sequence file. Depending on the nature of the test system and the deviations from one product to another, a sequence file may contain the tests for only one product or the tests for many products.

You should implement company or group-wide standards in a process model to complete a standard set of routines. In this way, you can establish a standard set of routines to execute before and after each sequence to complete routines such as serial number scanning, reporting, line communication, and so on.

For the routines that deviate from the established company or group-wide standards for a specific product or set of products, use TestStand callbacks. Callbacks are routines defined in a process model that can be overridden by a specific TestStand sequence. Using the mobile phone example, we would designate the test report generation portion of our process model to be a callback. The sequence that requires the special reporting would use its own callback to override the test report routine of a process model. By using callbacks in your process model, you decide which routines, if any, can be overridden by a sequence that uses that model. For instance, if there are only a few possible types of reports that you would generate, it might be more appropriate to implement them all in a process model. Set a flag to determine which type of report TestStand generates for a given sequence, instead of, or even in addition to, using a callback. In fact, the default TestStand process models use a similar technique to generate either a text or html report.

Thus, process models give you ultimate control over the test system. Using callbacks and intelligent sequencing within the model, you can make it as rigid or flexible as is appropriate for your system. For more information on process models and callbacks, refer to Chapter 14, *Process Models*, in the *TestStand User Manual*.

Use of Custom Step Types

Step types provide a very powerful tool for shortening sequence development time. Step types are an effective solution for the following tasks:

- Defining a template for the inputs, outputs, properties, limit comparison, and error handling for a user-written test
- Providing a drop-in tool to use in a sequence
- Providing a link to test code that you cannot call directly with a TestStand adapter

All step types shipped with TestStand fall into the above three categories. The Numeric Limit step type, for example, defines a set of outputs (numeric value, error information), properties (high and low limit), and comparison for any code inserted into a sequence as a numeric limit step. The Property Loader is an example of a tool that you can insert into the sequence. The Property Loader step type contains code to load limits and properties from a text file, Excel file, or database at run time. The Call Executable step type provides a link to test code that you cannot call directly with a TestStand adapter. The Call Executable step type uses the DLL Flexible Prototype adapter to call a DLL function that in turn calls the executable you specify. In this way, the DLL Flexible Prototype Adapter allows you provides to call executables within a sequence.

Consider using a custom step type to make sequence development as productive as possible and to promote code reuse. For more information on how to create and use custom step types, consult Chapters 9, *Types*, and Chapter 10, *Built-In Step Types*, of the *TestStand User Manual.*

Test Code Development

Equally important to the organization of your test sequences and process model is the organization of the test code. TestStand gives you the flexibility to use many types of test code and pass data freely between TestStand and the code modules, but this flexibility does not replace the need to have uniformity among code modules. Regardless of which application development environment (ADE) you use, the following guidelines will help you create modular, maintainable code:

- Use a source code control program for projects with more than one test developer.
- Separate measurement and analysis code from the communication with TestStand.
- Encapsulate commonly used routines within a shared function or step type.
- Use code templates to promote uniformity.
- Have a uniform, robust strategy for error handling.

Source Code Control

Source Code Control (SCC) programs are designed to manage shared development of test code. An SCC program tracks revisions and protects against two or more users making changes to the same test code at the same time. Using an SCC program is beneficial for a single developer and almost essential when development is shared among several

engineers. TestStand integrates with any SCC system that uses the Microsoft SCC interface. You can check files and projects in and out of your SCC system from a TestStand workspace. Specifically, National Instruments has tested TestStand with the following SCC providers:

- Microsoft Visual SourceSafe
- Perforce
- MKS Source Integrity
- Rational Clearcase

Separate Measurements from TestStand Communication

As a general rule, the test code should be as independent as possible from the test executive. Having modular tests that perform a single measurement with perhaps some analysis are not only easy to read, but also the easiest to reuse in other applications. TestStand provides a means to pass data freely between a sequence and the code modules it calls. In the code module, data is exchanged using calls to the TestStand API. Encapsulating these calls into a function or subVI makes the test code easier to use within other programs. To take this one step further, you can use a custom step type to provide a wrapper to the actual test code. In this step type, the code that is part of the step type handles all interaction with TestStand, leaving the test code module completely independent from TestStand.

Encapsulate Commonly Used Functions Within a Function or Step Type

As a general programming practice, you should place commonly used functions in a shared library for maintainability so that changes to the code within the function are reflected in all places where it is used. In TestStand, commonly used routines can also be part of a step type. As an example, consider a step type that is designed to bring back a waveform of data and make measurements to characterize the waveform. In TestStand, you can design a step type with a postsubstep routine that makes the measurements on the acquired waveform. Thus, every instance of a step using this type would use the same analysis routine. In fact, for the developer using this step type, the analysis routine executes transparently.

TestStand Code Templates

Each step type in TestStand can have associated code templates for the LabVIEW, CVI, DLL, and HTBasic adapters. The code template defines the skeleton code for a particular step type that TestStand creates when the sequence developer uses the Create Code option for that step. You can also define multiple templates for a single step type, in which case a menu appears from which the developer can choose a template to use. For the DLL Flexible Prototype Adapter, you can use a code template to define the default mapping between the DLL prototype and the properties of the step. For more information on the use of Code Templates, refer to Chapter 9, *Types*, in the *TestStand User Manual*.

Error Handling

An important element in any test system is error trapping and handling. As discussed earlier, the Cleanup step group is useful for handling errors and, if necessary, shutting down a test system gracefully. In addition, a certain amount of error trapping must be done in the test modules themselves. All the standard TestStand step types include a structure for error information. This structure includes a Boolean value to signal when an error occurs, a numeric for an error code, and a string for an error message. In TestStand, a run-time error is not a test failure; instead, it indicates that there is a problem with the testing process itself and that testing cannot continue. When a test module traps an error, it then reports this error condition back to TestStand. To be useful to the individual running the test, it is important that the error have a useful message. One way to handle this error is to establish a set of error codes and associated messages. You can use a function to lookup the message based on a code and pass both the code and message back to TestStand when an error occurs.

Another aspect of error handling is the ability to shut down gracefully or attain a known state when an operator detects some incorrect condition and manually shuts down the test system. You can stop execution in two ways. When you *terminate* an execution, all the Cleanup steps are executed and the process model continues to run. Depending on the process model, it might continue with the next UUT or generate a test report. When you *abort* an execution, the Cleanup steps do not run and the process model stops executing. You abort an execution when you execution must cease as

soon as possible and you are not necessarily concerned about the resulting state of the system.

Whether you choose to terminate or abort, TestStand does not automatically stop execution in the middle of test module code. To enable a test module to shut down automatically when a sequence is terminated or aborted, you must use the TestStand API to check the status of the execution. The LabVIEW Get Monitor Status.vi and the LabWindows/CVI CancelDialogIfExecutionStops function provide ways to monitor the execution state. In other ADEs, you can use the TestStand API Execution method GetStates to check the status of an execution to see if it has been terminated or aborted. It is especially important to use these functions when displaying a dialog box or running a particularly long test so you can exit these routines in response to an operator's attempt to stop the execution.

Summary

Think modularity when creating your TestStand sequences. Design them using the techniques described in this application note along with some common sense, and they will be easier to read and maintain for you and for others. In addition, using built-in TestStand tools, such as custom step types and code templates, decreases your sequence and test code development time. Refer to the *TestStand User Manual* or **ni.com/teststand** for more information.

- Learn More About NI TestStand
- Get Started with NI TestStand
- TestStand User Manual
- See the NI TestStand Featured Examples