Module 55: Acceptance Testing for Advanced Transportation Controller (ATC) Application Programming Interface (API) Software

Ken Leonard: ITS standards can make your life easier. Your procurements will go more smoothly and you'll encourage competition, but only if you know how to write them into your specifications and test them. This module is one in a series that covers practical applications for acquiring and testing standards-based ITS systems.

Ken Leonard: I'm Ken Leonard, the director of the U.S. Department of Transportation's Intelligent Transportation Systems Joint Program Office. Welcome to our ITS standards training program. We're pleased to be working with our partner, the Institute of Transportation Engineers, to deliver this approach to training that combines web-based modules with instructor interaction to bring the latest in ITS learning to busy professionals like yourself. This combined approach allows interested professionals to schedule training at your convenience without the need to travel. After you complete this training, we hope that you'll tell your colleagues and customers about the latest ITS standards and encourage them to take advantage of these training modules as well as archived webinars. ITS standards training is one of the first offerings of our updated professional capacity training program. Through the PCB program, we prepare professionals to adopt proven and emerging ITS technologies that will make surface transportation safer, smarter, and greener. You can find information on additional modules and training programs on our website at www.pcb.its.dot.gov. Please help us make even more improvements to our training modules through the evaluation process. We will look forward to hearing your comments, and thank you again for participating, and we hope you find this module helpful.

Ralph Boaz: This is PCB (Professional Capacity Building) Module T308: Acceptance Testing for Advanced Transportation Controller (ATC) Application Programming Interface (API) Software.

Ralph Boaz: My name is Ralph Boaz and I am happy to be your instructor today. I have served many years as a project manager, system engineer, and domain expert for the ATC and NTCIP standards programs. I was project manager for the ATC API reference implementation software, and this software is a major advancement for transportation field equipment. Today we will discuss the testing of this software, as it's deployed in ATC controllers across the country.

Ralph Boaz: Now we'll talk about the learning objectives. Our first learning objective: we'll explain the purpose of the Validation Suite—that is the program that's used to test API software—and we'll explain the architecture of the APIVS and discuss its features. In the second learning objective, we'll examine the organization and content of the APIRI test documentation and how we can use it to test API software running in ATC units. In the third learning objective, we'll discuss how to prepare the APIVS software for testing and how to perform actual tests using a method we call the USB Test Package method. Finally, in the fourth objective, we'll show how the results can be analyzed and viewed and how reports can be generated with off-the-shelf tools. People taking this course may have various levels of technical abilities. You may have a manager taking this course just to understand what is involved or what kind of personnel he may want to have onboard, or you might have software engineers taking this course. So we're going to hit a lot of levels in this course, so I want you to all have courage. There will be aspects in here that may be a little over your head, but by the end this will all come back and should make sense.

Ralph Boaz: So here we're going to explain the purpose of the Validation Suite software. In this learning objective we're going to discuss the content of the APIVS architecture and talk about the features of the software.

Ralph Boaz: Some basics: a transportation controller is a computer. Now, traditional controllers run a single application program, such as the signal program or a ramp meter program, or maybe a data collection program. The API software that runs on ATC units allows many application programs to run simultaneously. Application programs may come from different vendors than the provider of the ATC unit. Consequently, working, consistent, and tested API software is essential to make this feature work for the entire industry.

Ralph Boaz: So this is just an illustration of what we just talked about. On the left you see, in this case, a Model 2070 controller that doesn't conform to the ATC standard—his particular model—and it's running a single program: a data collection application, a traffic signal application, or a ramp meter application.

Ralph Boaz: ATC units can perform numerous applications simultaneously. That's possible when we're using ATC API software. So in this case we have a NEMA-type of controller that's an ATC and we show a 2070-like controller that's ATC. Actually there is a model of 2070s you can get that will conform to the ATC standard. This is just some example programs that might be run on ATCs. Rather than add boxes to a field deployment, you could actually just be running various software packages right on the controller.

Ralph Boaz: API software is made up of three software libraries. There's one for the front panel of the controller; there's one for the field I/O device that is used; and there's one for the Time of Day, which has to do with the real-time clock. There's two resource management programs. We call one the Front Panel Manager, and that's kind of like Windows is on a PC, where it controls the view of the applications and how they work together—so that's what the Front Panel Manager does. Then the Field I/O Manager: that controls the outputs and inputs that come in and out of the controller from the field. And what this software does, it allows programs to be written that can safely share the resources of the controller.

Ralph Boaz: So this is an example of what the Front Panel Manager window looks like. In this case there's five programs running—and you see the numbers to the left of each of these programs. If you were on the controller and you selected, let's say, number 2, the Emergency Management Program would populate the screen and it would just look like that; or if you said 1, the signal program would come up and you'd see your traditional signal programs—whatever ones you're using in the display. So this allows you to share—allows applications to share this controller.

Ralph Boaz: Now I'd like to talk about testing. Traditional controller testing: there's a couple things that are tested. One tests the controller hardware, which may also include the operating system, and the other is one that tests the application program running on the controller. So on the right—you see the picture in the top right—those are 2070 units in the Caltrans lab, and in this case they're running through all kinds of steps and they're being heated and cooled and humidity applied and dried out, and all that's going on for many hours. This happens in the Caltrans lab. They're testing the controller hardware, and then they'll also run some software tests on the operating system at the same time. All that's going on in their environmental chamber. We also want to test the application programming, which, as I said previously, probably the three major ones are signal program, a ramp meter program, and data collection programs. Those are really testing applications. For signal programs, you see on the right—right bottom—is what we call a suitcase tester, and that can be hooked up right to the controller with cables. Sometimes you'll have a conflict monitor attached also, and you can flip switches and see what the application program does. On the lower left, that's basically a software version of the suitcase tester, and that's used also.

Ralph Boaz: So this is an important aspect to grasp. Here we have—I'm going to show you three kinds of testing. We have unit testing—and actually all three of these are a form of unit testing—but we have unit testing where we're just talking about the hardware and the operating system. This is typical of Caltrans when they're doing their procurement cycles, because they have their own software that they load on the controllers. They're testing the hardware and the operating system. For the most part, that covers things like 170 users as well as 2070 users, which makes up about a third of the industry. Then we have NEMA units—or non-ATC 2070 units—that are running application software. Sometimes you get those bundled, so the testing that occurs there is that we're not only testing the hardware and operating system; in this kind of unit testing, we're also testing the application software. And now with the ATC units, we not only need to test application software and the hardware and operating system, we also need to test the API software that makes all these applications be able to run safely together. That's what we call the APIVS, or API Validation Suite. Note that this has never needed to be tested before because it did not exist. Traditional controllers did not test API software, because they only had one application running. Now we have this body of code that needs to be tested so that we can make sure everything works properly.

Ralph Boaz: There's four methods of software validation: that's inspection, demonstration, analysis, and test. What we're trying to do is validate that the API software conforms to the ATC 5401 standard. We developed this APIVS software to test the API software that's running on the controllers. Testing is pretty simple at the core level. It involves initiating a test and comparing the results of that test to a known correct result. And of course we don't want to do this one time; we want to be able to do this over and over and over again.

Ralph Boaz: Now, we interviewed users before we began the APIRI project. One of the things that they asked for is they wanted to not have to hook up a bunch of equipment to the controller to test this API software. We utilized the computational power of these ATC units, which are thousands of times more powerful than 170 controllers, and we used that computational power to allow internal testing of the API software. We needed to emulate the front panel and field I/O devices because we're not going to require the user to interact with the screen, so we don't want to go through the front panel and to the screen and require push buttons. And we didn't want to have this hooked up to a suitcase tester or to actually be hooked up to field I/O devices in the field, so we emulate those within the controller and it's part of the APIVS software. We have something we call "loopback drivers" that cause the API software to respond to stimuli that was initiated from another part of the software. So it's like routing the outputs into the inputs of the software. And then this APIVS software captures the test results and compares them to known results that we captured previously.

Ralph Boaz: Now we're going to review the layered architecture of ATCs—the layered software architecture—here in this slide. You recall we have what we call the hardware layer, which is essentially the engine board that resides inside the transportation controller. Then we have a Linux operating system and device drivers—we call that the board support package layer. Then we have the API software, which is the API software layer. Then there are applications that run, and this is the layer of software inside the controller that's used for things like data collection or emergency management or ramp metering, etc. Then we have the user that interacts with those programs, as well as—if they're operating with the Front Panel Manager, for instance, they'll be actually interacting with the API software itself. This is the architecture we've talked about in the previous modules. If we talked about the flow of information, we're talking about this like from the user through the application software and then out to the field—the program's interacting going out to the field through the hardware—as well as the inputs come back through and, if the user is watching the screen, he can get the information such as what the signal program is doing. So we have this flow of information through this software architecture. Now again, as I said previously, we can't have the same architecture because we don't want to have to have the controller hooked up to other things.

Ralph Boaz: So we modified that architecture for the APIVS. Here's the same bottom layers as we had before; but instead of an application program, we have the API Validation Suite running in that space. And then we have the tester—he's a particular kind of user that's trying to test the API software; and then we have these loopback drivers that take the output from the API software and send it back to the

Validation Suite. So we get this operation of it—kind of internal looping through inside the controller without the need for human interaction.

Ralph Boaz: Here's a little more detailed architecture, just to illustrate the point or enforce the point. We have the APIVS software. Now, the actual executable module for this program is called the Validation Suite Engine, or VSE; that's shown here. And that VSE then talks to the API software libraries and managers; the outputs of that software go to the loopback drivers, and that information is pumped back to the VSE. Now, to make this VSE work, we need some other files for it to use. We have the test scripts that says what the API software is supposed to do, so you feed that into the Validation Suite; and then we also need to feed into it what the expected results should be, so those are the expected result files—and we call those flat files, for kind of a generic term; and then we have the API conformance report—in other words, how did it do? These are logs. We're going to go into details of this in the next few slides.

Ralph Boaz: We have a command window—the interface that's used in the APIVS software, it's called a command-line interface—and you'll recall, if you've done much on a Windows PC, sometimes you'll see this black window come up, or those of you that are more savvy, you will have typed different things into the Windows command prompt. This one is actually called a Linux shell, and it’s equivalent to a Windows command prompt window except that it runs under Linux. Here we see in this line right here we have the Val Suite Engine that calls up the program; and then we have a bunch of options that are indicated by the rest of the text on this line. This VSE runs those test scripts and uses the known results and runs the test. So now we're going to break down what's in here.

Ralph Boaz: We talked about the VSE, and that's the name of the executable program.

Ralph Boaz: We have this option L, and option L, that's the level of output: how much do we want to put out on the conformance report. 1 stands for just a pass/fail indication; 2 indicates that we want some high-level tracing and a summary result and the pass/fail indication; a 3 indicates that we want every line, every trace that's executed during the test to be logged.

Ralph Boaz: And we have dash-c: that's a configuration file, and that specifies a series of configuration items for the VSE software, such as what's the size of the screen, how many lines, and various other things.

Ralph Boaz: Dash-i: this is the input test script to the VSE; this is where—you put the file here that says, "Okay, I want the VSE to perform these operations using these functions of the API software.”

Ralph Boaz: Dash-o: that's the output file. That's where we put our logs, and both the input file—the XML of the APIVS XML file—and the output file are written in Extensible Markup Language. This is a kind of method of writing human-readable but somewhat technical specifications. It's not as difficult as writing C code, and that way a user could actually modify these files without having to have a compiler, for instance.

Ralph Boaz: The other options that are less used is the Test Suite Name. We actually don't use this in any of our examples, so I won't go into it, but it was a way of sub-setting tests in a method in one way.

Ralph Boaz: We have dash-R count, which is the repeat count. We do use this sometimes if we want to repeat a test over and over.

Ralph Boaz: Dash-H is a halt on error. Most of the time you want to run all the tests and try and collect all the information, but sometimes you may want to run a test and halt it as soon as the first anomaly occurs. That's what the dash-H does.

Ralph Boaz: The last one is called capture mode. We use this in actually formulating tests; that's kind of out of scope for this presentation.

Ralph Boaz: So now we'll do an activity.

Ralph Boaz: What type of controller software is not traditionally tested by agencies? The answer choices are: A) Data collection software; B) Signal control software; C) Application programming interface software; or D) Ramp meter software. What type of controller software is not traditionally tested by agencies? Please make your selection.

Ralph Boaz: If you said the application programming interface software, you were correct. Until recently, it was not possible to test the API software, and vendors were just starting to deploy the API software in their controllers. But now this APIVS software discussed in this module provides this ability. If you said data collection software, that was incorrect, because that's an application, and agencies usually have methods of testing their applications. The same is true for the signal control software, as well as the ramp meter software. Now in previous versions of this course, we had questions arise, so I'm going to weave those into this module at the end of various learning objectives. One of the questions was: How does this compare to the diagnostic and test programs that come with 2070 units that use loopback cables on the serial ports? We call that the DAT programs; those are used to test the hardware and the operating system. When loopback cables are used, they are involving the hardware aspects of the controller. You can imagine the controller and big black cables connecting the inputs and the outputs. Actually there's a way to use that kind of configuration and test the API software, because those loopback drivers are effectively doing what those cables do. But we don't teach that method in this module because there's really no need for the cables.

Ralph Boaz: Now we're going to go into our next learning objective.

Ralph Boaz: We'll talk about using the APIRI project test documentation to specify acceptance testing. In this section we're going to go into details of how all the testing documentation—the various files that make up the testing of API software—possible, we're going to discuss all those and show how they all work together. If all you're trying to do is do some basic testing of all the API software in an easy method, we're going to talk about that later. Right now we're going to go into some details about how all these documents and things work together, and that's for those who are more software-oriented people or for non-software people who might be making modifications to XML files and such.

Ralph Boaz: So we have this APIRI open source project and we developed two bodies of code: the API Reference Implementation and the API Validation Suite. Having it as an open source project, that allows all the entire industry to help support this work of software that is deployed into controllers around the country. We have manufacturers, software vendors that will write apps for the controllers, the consulting firms that have products that go there, and agencies that have their own software or just want to be able to test the equipment that they have. All those people can contribute to this open source project. That makes all our products better as that software—the RI software—is deployed as API software in the traffic controllers.

Ralph Boaz: The APIRI project was completed in October 2016. We had OSS implementations of the RI and VS software, and those links there—you can go to actually register and get more information about these bodies of software. It followed a formal verification and validation process, so that anybody can test API software on a controller. We used IEEE 829 in creating the software and creating the test documentation.

Ralph Boaz: Benefits. It's consistent with the Linux operating system, because that is an open source project. So those in the manufacturing and software development worlds that are familiar with Linux: this is natural for them. It promotes collaboration of developers, which is good for the industry. It provides a forum for users. If you had a problem or a question, I recommend registering on those websites, because that way you can ask questions and get experts to answer them for you and help you along. It promotes quick bug fixes. If somebody finds a bug, everybody gets the bug and can work on it, or somebody will offer various alternative solutions to their situation or offer a bug fix directly. And this facilitates introduction of new application developers. This software will eventually all run on a PC, for instance, and you could actually have people in schools developing software for ATCs and knowing that it works. And it's incorporated all the software—the software is incorporated on ATC units by manufacturers. There's not additional costs for the product itself. It's all free. The most important thing is that it provides the best opportunity for consistent API software behavior. Right? Everybody is using the same base code so it all behaves consistently. If everybody developed their own libraries separately, we'd have nuances and odd things happen that just couldn't be foreseen.

Ralph Boaz: So now we'll go down into the test documentation, and you'll recognize the terms here from the prerequisites that had to do with testing. We actually use the names from IEEE 829-1998, but we followed 829-2008 in the project.

Ralph Boaz: So first we have the test plan: that specifies the scope and approach for testing. It identifies the features to be tested.

Ralph Boaz: In our case, we included the test design specifications, which are next; this is simply a refinement to the test plan. In the APIRI project, we had a test design specification for each of the software libraries: the front panel user interface, the field I/O, and the time-of-day libraries. Now these TDSs are very simple in our project, because they were included as part of the test plan, and the test plan really covered the details.

Ralph Boaz: Now the next one: this one's very important and we'll talk a lot about this in the remainder of the module. This is the test case specifications. What these do is they define the inputs and outputs of a test, and that's the simplest way to say it. There's about 40 of these test case specifications developed in the APIRI project, and there's room for others to be created. We're going through these details just so that those who are game could potentially offer up more tests that we could apply to the software.

Ralph Boaz: Then we have the test procedure specification. That simply describes, from a user's point of view, how to perform the tests: what to plug in, what to do, what to copy, those kinds of things. So this is oriented towards the tester. The test case specification is oriented towards the test—the inputs and outputs of the test—and the test procedure is oriented towards the tester.

Ralph Boaz: This shows the arrangement of the FPUI test documentation. So we have the test plan—in our case, the FPUI test design is part of that plan—and then we have test cases and test procedures that are used to perform the testing. Then we go and—all this is done and set up ahead of time before we actually execute the test.

Ralph Boaz: Here's a similar picture for the field I/O software. We have the field I/O test cases and test procedures.

Ralph Boaz: Then we have the time-of-day test cases and test procedures.

Ralph Boaz: This is the outline of the test plan; actually, I think it's a portion of the outline. We have the introduction. We talk about the thing we're testing in Section 2. We talk about the features to be tested and the features not to be tested—our approach, pass/fail criteria, etc.—through the remainder of this outline here. I want to talk about how we show the features to be tested.

Ralph Boaz: This is a table that's in the test plan. We have different columns in this table. We have the test ID—or identifier—that says what document it is; then we have the name of it so we know what it is; and a brief description of what's in it. If we go across we can see that, "Oh, okay, so we have this test ID 2040. That's Test Case Specification number 4, and that has to do with the front panel user interface reading and writing of the data." So this is how it shows the organization of the features that we're going to test.

Ralph Boaz: This is the remainder of that outline: the responsibilities, staffing and training needs, the schedule possible—in your case, the schedule would be yours. If you wanted to use this outline—you could actually use this whole outline if you had a formal test process that you wanted to set up for your agency and fill in the information to be specific towards your agency. Approvals, appendices—the appendices are there. We have these Requirements to Validation Description Matrices, and now I'm going to show you that.

Ralph Boaz: So how do we know that we've actually tested each requirement? We have the requirement IDs that come straight from the ATC 5401 standard—known as the API standard—and those are written on the left; and the requirement description; and then we have the function involved that is used as part of that description—or part of solving that requirement; and then we discuss how we might—we have a design narrative that discusses how that function is implemented. Then we identify which test case is associated with testing that function, and which test procedure. So here you see it goes across. The requirement is that the API shall provide a function to read a queued character or key code from the input buffer of a window. That's called FPUI Read Character. Then we have the description that says it makes the use of the Linux system call to return a single character from the input buffer of the FrontPanelDriver. So that describes what it does. Then we have our test case and test procedure.

Ralph Boaz: We said the last appendix to the test plan were the test design outlines. This is an example for the FPUI features. So this is our Test Design Specification 1 for the FPUI—the features to be tested. Actually we capture those features to be tested not in this TDS portion but back up, as I described in the test plan.

Ralph Boaz: We talk about the approach refinements; then we have the test identification; and then the pass/fail criteria. Simply put, the pass/fail criteria for the software is that it's going to pass every individual test case and it will fail if it does not—so it's pretty simple. Again, we have one of these for each of the libraries.

Ralph Boaz: Now here's the test case outline. This particular test case is called FPUI Reading and Writing Data; it has identifier; and this is our—test the operation of the API functions used to write to the display and read key presses from the front panel. Now remember that all those key presses are actually emulated in the software, because we built those emulators so that the user does not have to interact with the controller. This is really important: the next thing is the test items. We actually list the requirements that are validated using this particular test case. The reason why there's multiple requirements is that to test one particular function—let's say it's to write a string to a window—you actually have to use a lot of other API functions to do that: you have to create an application—it does this behind the scenes; it has to then write to the screen, and various other things that are tested all at the same time. So we actually, in these test cases, will fulfill or validate numerous requirements.

Ralph Boaz: We talked about the test cases defining the input and output specifications. Here is a list of the inputs. In this case we have the XML file—which is the input which says what the test is supposed to involve—and then we have these other files: the CX files. We have a—the CX files, all those files are for setting up the emulators. Then we have the C2040_vd_1F file: is just a representation of what the display should look like when the test case is done, so that way we know that the tests function properly. We have this configuration file, and that file, as I said earlier, was to tell the VSE how big is the screen, how many lines is it, and other parameters. The output specification is this log file; we identify it, we use the scheme—you'll notice the input file, the 2040_in and the 2040_log—we code those so that when we capture all this information, we know what goes to what. So that's the conformance report—the log file at the bottom there is the conformance report or output of the VSE program.

Ralph Boaz: The test procedures: this is something that's oriented towards the tester and this—we have the identifier and the purpose.

Ralph Boaz: Any special requirements—we list those—talks about, okay we need a hard disk and a USB flash drive, etc. Then we have the steps that the tester needs to do, but we're not going to go through all those now. We'll actually give you a more condensed and show you a picture—a method of doing that—in the next learning objective.

Ralph Boaz: So I want to talk about these other files that I just mentioned. I'll step through these quickly. This may be more detail than you want, but we may have some software people that want to see what's inside them.

Ralph Boaz: We have the test scripts that's written in XML. It's human-readable but you don't have to compile it. It's used to specify the testing to be done.

Ralph Boaz: We have a generic term of flat files. These are just simple text files that are used to configure the emulators and to represent the known correct results of the output.

Ralph Boaz: We have the configuration file for the Val Suite Engine—I told you what that was.

Ralph Boaz: We have Linux shell scripts. So we have test scripts that are used to design the test, and then we have shell scripts. That makes it easy to run successive executions of the APIVS software, so that you don't have to type in every command—you'll appreciate that as we go along here.

Ralph Boaz: We have the output files, which are also in XML format.

Ralph Boaz: This is the beginning of the test script. You'll see that we put the same kind of information we used in some of our tables and we put that in here and—sorry, you'll see the requirements that are listed that it tests, etc.

Ralph Boaz: This is down further in the program, and this kind of stuff is just used to set up the variables. More interesting are these commands where we have—we're testing actually three different API functions called FPUI_write, FPUI_write_at, and FPUI_write_character.

Ralph Boaz: Now we want to be able to show what the expected result is. We had to have a way to represent that. This is a flat file that represents the front panel manager and if you focus—or represents the front panel. Here we're testing the front panel manager; so what the software—what the test did was to create ten programs and have it running. It wants to make sure the front panel manager shows that. We put what the front panel manager should look like if it was displayed on the screen; we put it in this file—that's in a circled area—and when the test is done, the Validation Suite will take its window buffer and compare it to what's shown here in this file and see if they match perfectly.

Ralph Boaz: In a similar fashion, this is used for field I/O types of tests. This is a hexadecimal representation of an electronic message that would go out to the controller. Here we just capture it. This VS software that compares it to this file to see if it was the right one.

Ralph Boaz: This is the configuration file; it sets up the paths where the software is—screen height, as they said, etc.—so that's configuration information.

Ralph Boaz: So now we have an activity.

Ralph Boaz: What document is used to specify the inputs and outputs for a particular test of the API software? Your answer choices are A) Test design specification; B) Test procedure specification; C) Test plan; or D) Test case specification. What document is used to specify the inputs and outputs for a particular test of the API software? Please make your selection.

Ralph Boaz: If your answer was D, the test case specification, you were correct. The test case specification specifies the inputs and outputs for a particular test; in the case of the APIRI software, we combine all the test TCSs into one document so that they're easy for users to find. If you said A, it's a test design specification, that was incorrect. The TDS is used to refine the test approach that was specified in the test plan. If you said B, the test procedure specification, you were incorrect. The TPS specifies the steps for the tester which is he's executing the test cases. And then we have the test plan: that was incorrect because the test plan specifies the scope and approach for testing. We had another question that came in. It says, "This is pretty complicated, do I really need to do this?" No, as I said: you don't all have to do this, it's been done for you. Now if you're a person that's willing to contribute to the project, you'll need to know what we just discussed, but most users will just take what was done and exercise the test. You'll see more of that in the next learning objective.

Ralph Boaz: We've explained the purpose of the validation suite software, and we discussed using the APIRI test documentation to specify testing. Now we're going to use the API software to actually test—the APIVS software to actually test API software.

Ralph Boaz: In this learning objective we'll cover the open source software environment that hosts the APIVS and APIRI. We'll discuss how to prepare the APIVS software for testing. You'll have the fun experience of seeing how to perform actual testing of the API software using the USB test package method.

Ralph Boaz: This is what the repository looks like for the APIVS. We're just going to run through these pieces of this just to highlight. This tells you what repository it is. Here it says, okay, there's issues here, there's maybe a bug or a question that has been submitted by those that are registered; so those are ongoing. Here we talk about there's been 123 commits, which means there's been 123 times software has been submitted here to be developed. This is a representation of the file system or organization of the files that are in the APIVS repository. And this is a way to clone it or download it from the repository onto your system.

Ralph Boaz: We're going to talk about the equipment that's required. So the basic equipment is you need to have an ATC unit with API software running on it—that's pretty basic. You'll want to have a PC (personal computer) with at least one gigabyte available of hard drive storage and a USB port—that's pretty easy to do, too. Then you'll need to have the VSE executable program; you can get that from your ATC vendor—if you're a power user, you might compile the software yourself from the repository. But most users will just get this from their vendors. There's two methods of testing that can be done; if you're using the command-line interface method, you're going to need to have a serial Ethernet cable connected to the PC and to the ATC unit. Recall, the command-line interface method would be where you're typing in each command and, if you remember, we had 40 different test cases. So it's an arduous task to do that. So we recommended the USB test package method. Here you'll need—in addition to the basic equipment, you'll just need a USB flash drive that's formatted in the typical Windows fashion.

Ralph Boaz: The best time actually—just to make a point here—to get the VSE software is to include that in your procurement. So you'll want to have that when you request for bids—or something like that—that you have that written in there that the vendor will supply that VSE. But they'll want to supply that anyway, because they want to be able to stand behind their product and don't want to have fingers pointed at them from other vendors. This is just an illustration of the command-line interface method. What you're running is actually a Linux shell from the PC hooked into the ATC. It allows complete control and execution of a test. It's the best method if the tester wants to do a lot of variations on a single test. You have to be comfortable working in a Linux environment to do this.

Ralph Boaz: Next method is the USB test package method: here what we have is pre-configured tests that can be downloaded from the web to your USB flash drive. You simply take the USB drive, plug it into the ATC unit, and turn on the power. Now we also allow it, using this method of simple variations to it, using edits of the runAPIVS file, which we'll show you—and that's if you're a little more game, you want to try that—it's great to be able to isolate tests. And this can be done with a Windows or Linux environment on your PC.

Ralph Boaz: The steps are this: we download or clone the APIVS repository to your PC; install a flash drive into your PC; and then depending if you're running Linux or Windows, you run package.sh or package.bat. What that does is it'll configure the USB drive so that it can test your software on your controller. So you run that package. Then you copy the VSE executable and Loopback Drivers that you received from your vendor—you copy those to the USB drive. This next step is only if you want to make some modifications to tests: you edit this file—the runAPIVS file. Then you install the USB flash drive unit—the controller unit—turn on the unit, wait for it to complete, and the test results will be loaded onto the USB drive when it's finished. You can take that out and view the logs on your PC.

Ralph Boaz: So let's give you a little graphic illustration. We have our APIVS repository. We're going to download or clone that software and files to your PC. You're going to get your executable file and the Loopback Drivers from your vendor, and you'll load that onto your PC. You're going to run your package.sh or package.bat program or script—depending on which operating system you have on your laptop—and that will load it all onto a USB drive. You'll plug that USB drive into the controller and turn it on. You'll remove the USB drive when it's done. You'll copy the results back onto your PC, and then you'll be a happy tester. I can hear the clapping going on from here.

Ralph Boaz: Now we're going to transition a bit, just to talk a little bit about ways you can modify that runAPIVS file. So a little more sophistication: you'll recall the command line—what that looked like, we discussed in the very first learning objective—the VSE and all of those other options. We had the options of what level, a conformance level for the output, we had an option for which particular test you wanted to run, where it goes, the output of the file. We also had the option for repeating the test over and over again.

Ralph Boaz: Let's talk about editing this runAPIVS script file. It's in the root of the flash drive; it's a thing that runs when it's booted up. It defaults to running all of the tests on the API software one time, with a Logging Level of 1, which says each test will just tell you if it passed or failed. But there's easy edits you can do to actually change the Logging Level or increase the iterations of a test, or to subset the test cases that are in there in case you don't want to run them all. That's what we're going to show you here.

Ralph Boaz: So this is what the runAPIVS file looks like. This is the beginning of it, you'll see that it's loading the Loopback Drivers. Here you'll show that the level of testing—the level of the output—is set to 1, which is just the pass/fail.

Ralph Boaz: Now we'll get into the gory details. This is a collection of runs of the VSE with the various XML input files. We'll look at the first one. There's the VSE program and it's set to Level 1—remember, because we set this environmental variable level to one—and there's the configuration file; there's the input file that's representative of the test case; and there's the output file. Now if you can imagine typing this in 40 times while you're sitting there at your laptop every time—you don't want to do that. So that's why we have these script files, and that's why the USB package method is ideal for doing this efficiently.

Ralph Boaz: Now let's look at—here we're editing it, here we changed the level of output to 3. Now we'll get everything that was in the test script and the design of that test script—every single trace will be listed in the output file. In this case, we wanted to repeat this next test 10 times, because maybe we were having some problems in this area. So we wanted the VSE to execute ten times on this particular test script. Here we just commented out some tests, and you could do that throughout the file. For instance, if you found just one failure performing your runAPIVS over and over again, and there was only one test—set of tests that were failing, then you could comment out the rest of them and just do those. So this is a way of sub-setting the tests that are being done. We just increased the level, we increased the repeat count, and here we sub-setted the test.

Ralph Boaz: Now we're going to actually show you a little bit of the execution of this. What we do is we take the USB and we plug the drive into the ATC. We turn it on and then we follow the screens that appear on the front panel. So this is what first comes up. It says, "Do you want to begin the test, yes or no?" Of course we said yes.

Ralph Boaz: Then what happens, it starts going through all of those calls of the VSE using the various test scripts; and those, as they go through, they come up here on the screen. You'll see at the bottom it'll record how many have passed and how many have failed.

Ralph Boaz: When it's all done and assuming everything passed, this is what the screen will look like. You can congratulate yourselves because you will have just tested an important element of your entire agency's architecture.

Ralph Boaz: Let's do an activity.

Ralph Boaz: What is not an appropriate reason to edit the runAPIVS shell script? Turn off all test output, that's A; B) Change the number of iterations on a test, that's B; C) Change the conformance report logging; or D) Select a subset of the existing test cases. What is not an appropriate reason to edit the run APIVS shell script? Please make your selection.

Ralph Boaz: Let's take a look at our answers. This was a little tricky. If you said A, turn off all test output, you're correct. The options are level 1, 2, or 3; there's no way to turn off all the test output. So that was a little tricky. If you said change the number of iterations on a test—remember the dash-R option—that's a good reason to modify the runAPIVS file. Change the level of conformance, remember we went to a 1 to a 3 there. Or D, if you said that, that was incorrect, because we can subset the test cases by simply commenting them out. Now to hit another question that came in in a previous module that we ran. We have all this test documentation of software to test the API software, but what about testing the test software? Actually we gave the same rigor that we're talking about here for testing API software: we did it for testing the APIVS software. But since this module is about testing the API software—which is what comes inside the controller and is used by all those different applications—that we didn't go into that here.

So we didn't go into the testing of the APIVS software. This is using the APIVS software to test API software.

Ralph Boaz: We explained the purpose of the APIVS software; and we talked about the APIRI project documentation and how it's used to specify testing; we just showed you how to use the APIVS software to test the API software. Now we're going to talk about interpreting and reporting results.

Ralph Boaz: In this learning objective, we'll cover how to analyze results using off-the-shelf tools and discuss test reports.

Ralph Boaz: Outputs of the APIVS software are in XML. In the simplest case, the users are looking for—all they're looking for is a pass/fail indication. But if you want more than that, you're going to need to use some sort of tool. These are off-the-shelf tools that are free. One's called Notepad++ and it's a general purpose editing tool, has a bunch of color coding and formatting of XML files. There is a site called XML Differences, so that you can compare one test against another. There's another product called XmlGrid: that's an online editor and it displays XML in grids or tables. And another product called XML Viewer and that shows XML in kind of a tree form. I really want to recommend Notepad++ because it's smart about if you're on a Linux machine or if you're on a Windows machine—it's smart about those files and won't change the caricature and line feeds at the end of a text line to the different formats used in those operating system. So it's smart about that and keeps your files in the right format for what you're doing.

Ralph Boaz: This is what an output of—one of our runs of the APIVS. You can see it's quite extensive. Here you can look at calls with timestamps for the various function calls. Here we have a timestamp for FPUI open, that opens up a window on the controller and it gives us a timestamp when that was executed. This is actually a level 2 output; it's a summary result as well as the pass/fail indication. The summary result includes the top level function names of the API function calls. By the way this was used in—using Notepad++, so you can see the color coding and it makes the things stand out.

Ralph Boaz: Now this particular one is the XML diff and this shows the differences between XML runs. What we did was we had two different runs, and then I diffed them together. You see one had a date—well it's indicated here that the dates were different from the two different runs—so those are called out here in this file. But the rest of the results of the test were the same.

Ralph Boaz: This is what XmlGrid looks like. Here you take the XML that was output from your run of the APIVS and it'll take each of the lines and put it into a grid form.

Ralph Boaz: And then this is XML Viewer, which is a tree-like structure that's used to view XML. You can output—these things that are displayed on the screen, you can output those to a file and look at them on your PC.

Ralph Boaz: So what do we do with all this? Okay, so testers may include the test logs in their test reports. So if you have just a Level 1, you're going to get conformance and nonconformance indication only. When I ran this, I ended up—remember there's 40 different test scripts—I ended up with 304 lines of output and it took about 16 minutes to do it. Now if you did Level 2 on each of those same tests, I ended up with 9,693 lines of output. That still took about 16 minutes. So that's a lot of output. But if you did a Level 3, which include the conformance/nonconformance indication, summary result, and all logs and traces, we ended up with over 73,000 lines of output and that took about 22 minutes. So the timing of doing this is not such a big deal but you'll note that the size of these files is quite different. Frankly, if you have 73,000 lines of output, it's going to be difficult to make use of all of that. So it's recommended that you have a Level 1 or a Level 2 as your general reference point and that you only put a Level 3 test on a particular test, not do it for all the tests in the runAPIVS file, but just do a Level 3 on a particular test that maybe has been having problems.

Ralph Boaz: So this is what IEEE 829-2008—the test report outline is. You have the introduction—that's typical; you have the details, so we had an overview of the test results and the next section is the details of the test results. And that's kind of a place where these logs could go in that could fill out your report and be something that you keep on file or show your boss that he knows you did your job. At the end of your testing, you may have a conclusion or recommendation that, if it failed, that the vendor needs to go back and see what's going on. Or you may conclude that the unit is ready for procurement.

Ralph Boaz: So each of the test scripts end up with your pass/fail condition. This is the results from simply the pass/fail indications—there'll be 40 of these, if you're using the canned tests that are already there. You'll see each section of these. You can just add this to that section two, section of the test report.

Ralph Boaz: Okay, here's an activity. It's our last quiz.

Ralph Boaz: True or false: it is a good idea to always log as much information as possible on all tests? Please make your selection.

Ralph Boaz: If you answered false, you are correct. If you log all the information, you create a voluminous report that is so big that it's difficult to understand what's going on or weed out what's passed and what not passed. While the full logging on a test can be done, it's advised that testers repeat their tests with the logging for tests that failed only previously. If you said true, that was incorrect, it's better only to do full logging on selected tests.

Ralph Boaz: We explained the purpose of the API Validation Suite software; we used the APIRI test documentation to specify acceptance testing; and then we used the APIVS software to test the API software; and we discussed interpreting and reporting results. So there was a few other questions that came in. Does all this testing guarantee that there are not bad interactions of application programs? No, it does not, because when you're using multiple applications on the same unit, you should have done—or you should be doing—some sort of integration or operational testing that give assurances. And in fact, it doesn't matter if there's just one program or multiple ones, you should be testing your applications on any controller and not just ATCs. How soon is all this software available? This testing capability is available today, and the summer of 2017 we'll be doing some other updates to it. Then someone said, "I want to know more." See your student supplement for the links to the IT website in the standards area and then also the GitHub repositories for the APIRI and APIVS software. You can register as a user there and then get to ask questions, get help, make comments, etc.

Ralph Boaz: So let's look at what we've completed to date. Modules 207 and 208 were really the background and understanding of what the ATC controller is and the API software, so these are probably the most important places that you started on. Then we went into these other modules on user needs and requirements and this is oriented towards those spec-ing an ATC unit. Module T307, that's for testing the controller itself. And now you've just completed acceptance testing for ATC, API software.

Ralph Boaz: Thank you so much for completing this module. I hope you've learned something, if it's just general application of testing or if you just want to do simple testing in your agency or if you're one of these—a software guy that wants to get into the weeds in this, we hope that this module served you. Please provide the feedback at the link below and give us comments. That will help us improve our training. Thank you very much.