CUSTOMER TRAINING

ADVANTEST

V93000
SmarTest 8 Basic User Training

Lab Manual

Version 8.2.5

January 2020

ADVANTEST®

V93000

SmarTest 8 Basic Training Lab Manual

Version 8.2.5

Published on 2019-12-09

Contents

Notices

Legal Notices

Trademarks and Registered Trademarks

- ADVANTEST is a trademark of Advantest Corporation.
- All other marks referenced herein are trademarks or registered trademarks of their respective owners.

Warranty

The material contained in this document is provided "as is," and is subject to being changed, without notice, in future editions. Further, to the maximum extent permitted by applicable law, Advantest disclaims all warranties, either express or implied, with regard to this material and any information contained herein, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Advantest shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this material or of any information contained herein. Should Advantest and the user have a separate written agreement with warranty terms covering the material in this document that conflict with these terms, the warranty terms in the separate agreement shall control.

Technology Licenses

The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

Restricted Rights Legend

If software is for use in the performance of a U.S. Government prime contract or subcontract, Software is delivered and licensed as "Commercial computer software" as defined in DFAR 252.227-7014 (June 1995), or as a "commercial item" as defined in FAR 2.101(a) or as "Restricted computer software" as defined in FAR 52.227-19 (June 1987) or any equivalent agency regulation or contract clause. Use, duplication or disclosure of Software is subject to Advantest's standard commercial license terms for 93000 systems and non-DOD Departments and Agencies of the U.S. Government will receive no greater than Restricted Rights as defined in FAR 52.227-19(c)(1-2) (June 1987). U.S. Government users will receive no greater than Limited Rights as defined in FAR 52.227-14 (June 1987) or DFAR 252.227-7015 (b)(2) (November 1995), as applicable in any technical data.

Safety Notices

⚠ **CAUTION**	A CAUTION notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in damage to the product or loss of important data. Do not proceed beyond a CAUTION notice until the indicated conditions are fully understood and met.
⚠ **Warning**	**A WARNING notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in death or severe injury. Do not proceed beyond a WARNING notice until the indicated conditions are fully understood and met.**

Lab 1: Preparation and Prerequisites

Learning objective

In this lab you will learn how to start SmarTest 8 and import a first test program into SmarTest 8 to work with it. The required steps are:

- Start SmarTest 8 with the correct model file.
- Create a workspace
- Change the view
- Import the SmarTest 8 project (i.e. test program) used in the lab.

Getting started

`Classroom training`: Your instructor should have already assigned you an account with a user name and password. Log in to your system controller. Make sure SmarTest is running in the OFFLINE mode.

`LMS`: If you are working in the LMS, please continue with Task 1 LMS and Task 2 LMS. You will execute your labs in the `SmarTest Playground`.

Note	The task 1 is about importing the required files into a workspace. Check with your instructor whether the files are already available or not.

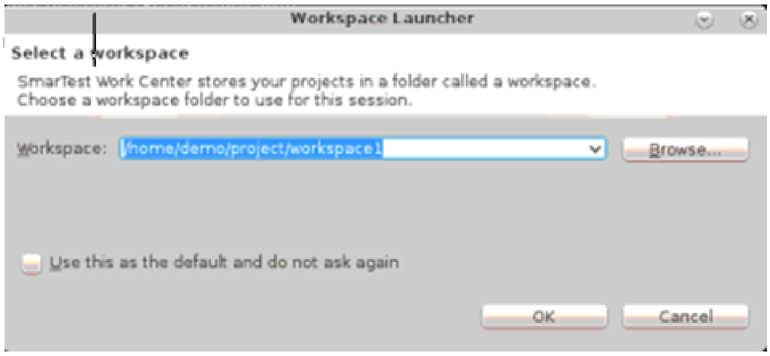
Task 1: Import Lab Files

Make sure that you have the necessary access rights to the test program or the project folder that you want to import, and to the file system on which it is stored.

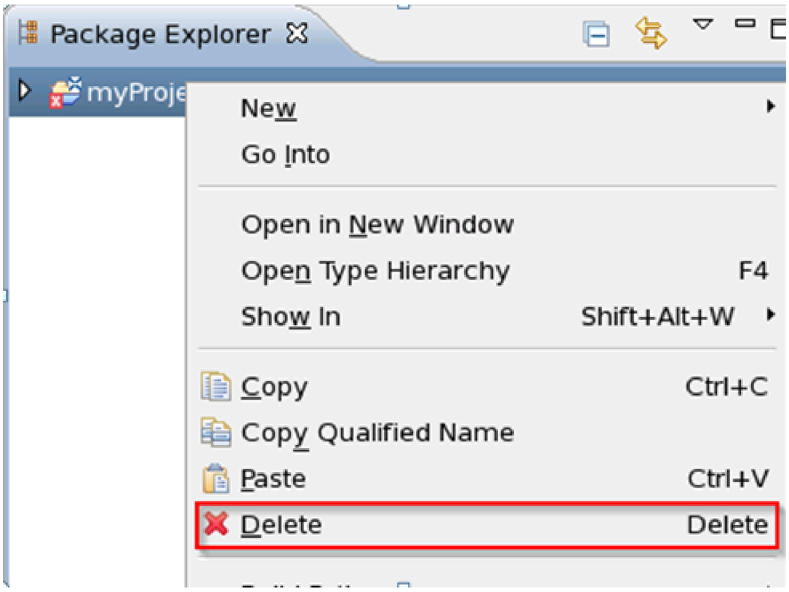
Set the correct model file on the command line:

1. `export V93000_MODEL=<labs_main_dir>/offline.model`
2. Start SmarTest 8, for the first tasks in offline mode:

   `/opt/hp93000/soc/prod_env/bin/HPSmarTest -o &`
3. Launch workspace

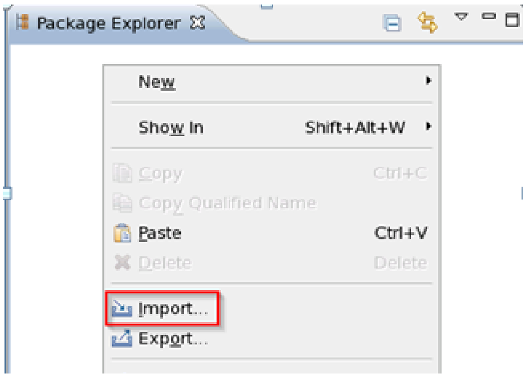


   Create workspace
4. In the **Package Explorer** remove all projects that probably have been already imported.



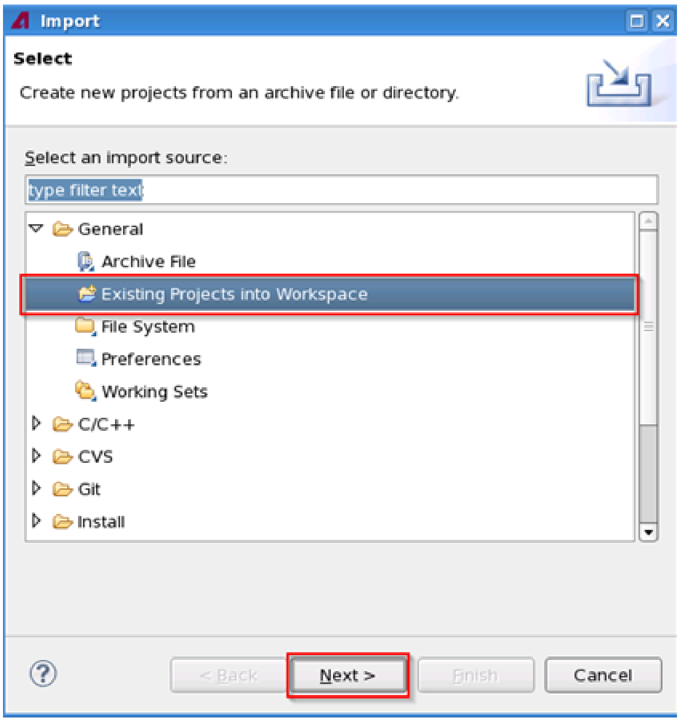
   delete project files

5. Import the project for the labs, stored in <labs_main_dir>/labsCrossConnect



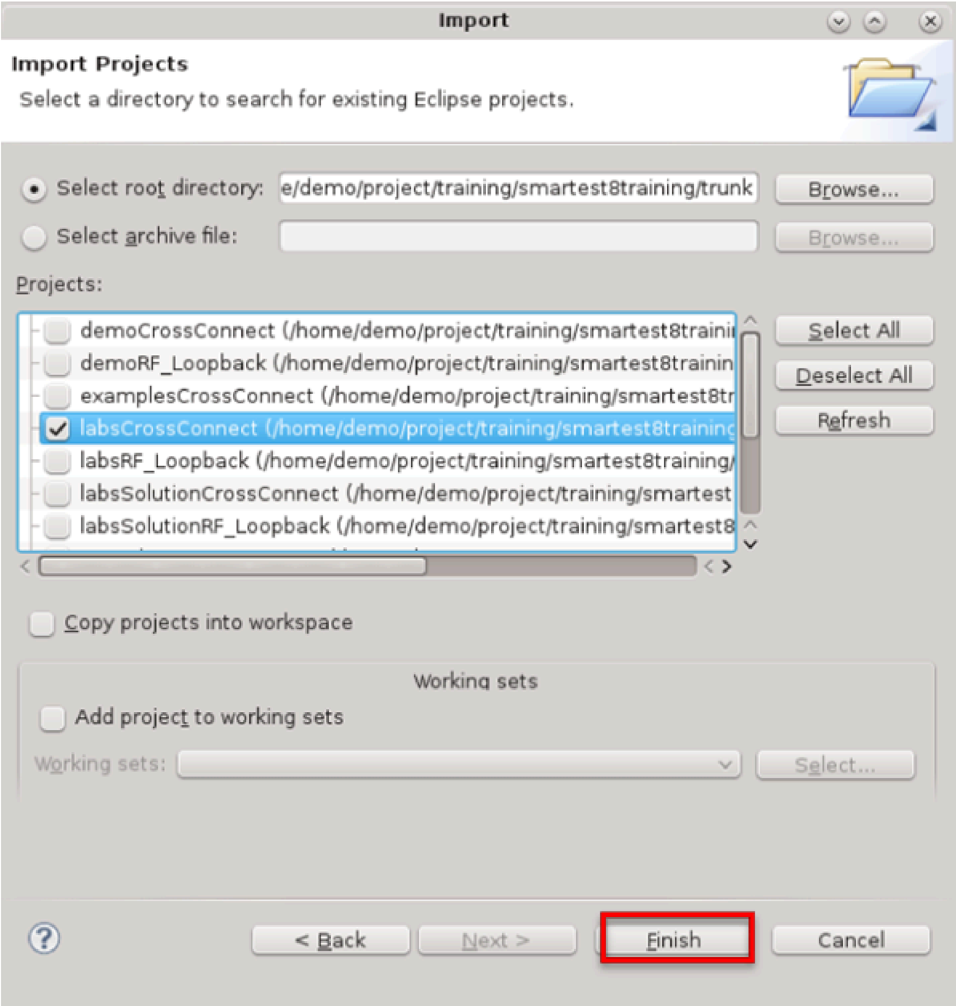
Import GUI

6. Browse to select project files.



Select projects to import

7. Select the project named labsCrossConnetc

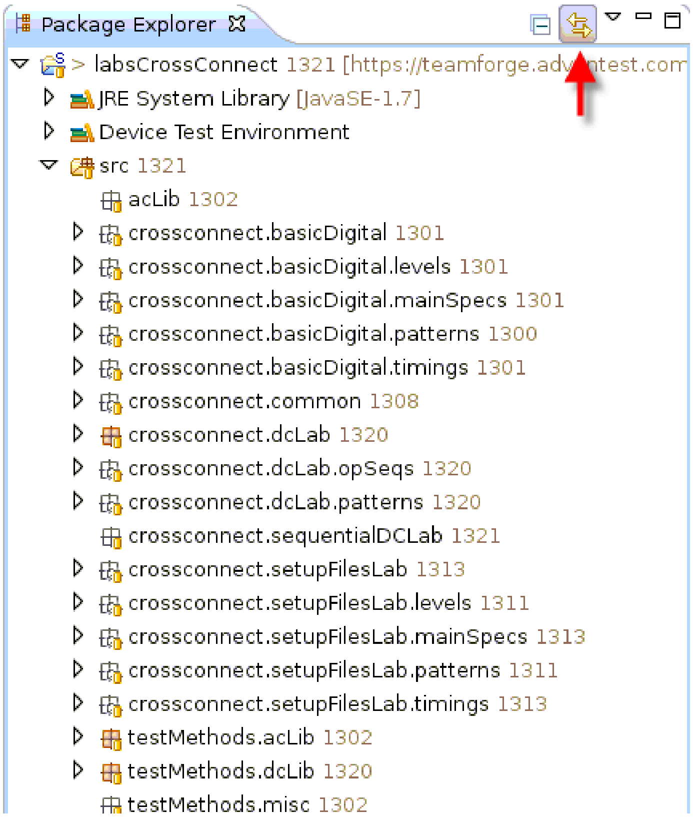


Select CrossConnect project

Task 2: Setting Package Explorer View Preferences

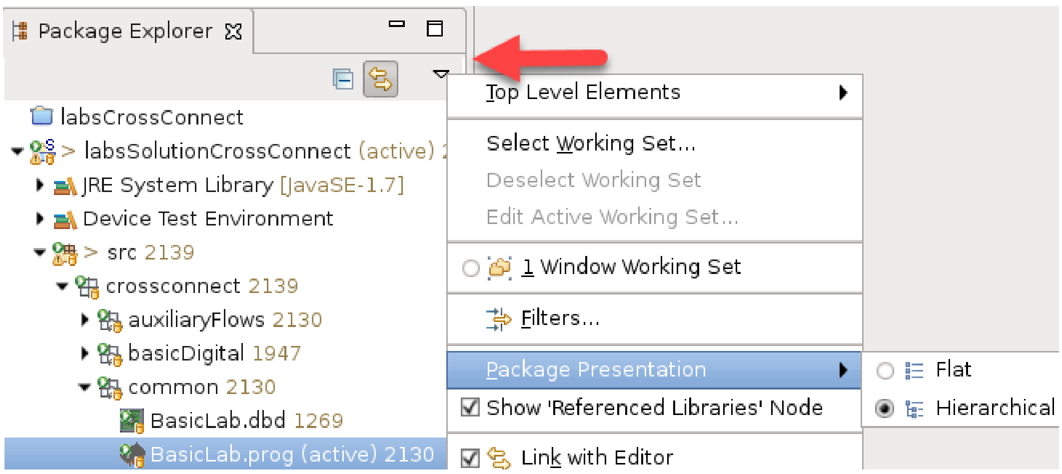
In this task you will set some preferences and configure the representation of the **Package Explorer**.

1. Check what the button **Link with Editor** does. Some views have a **Link with Editor** button (Navigator view - link with editor button). Enabling it helps you to keep a better overview if you have multiple files opened. Moving from one file to another will update the file location in the Package Explorer.



Link with Editor

2. Set the Package presentation mode to **Hierarchical**. You may want to set it back to **Flat** if you prefer this presentation. For more details about Package Presentation see TDC#244288.



Flat vs Hierarchical view

Task 1 LMS: Starting labs in the LMS

When working in the LMS you will setup and execute your lab exercises in the SmarTest Playground accessible from the LMS (Learning Management System).

Before you begin

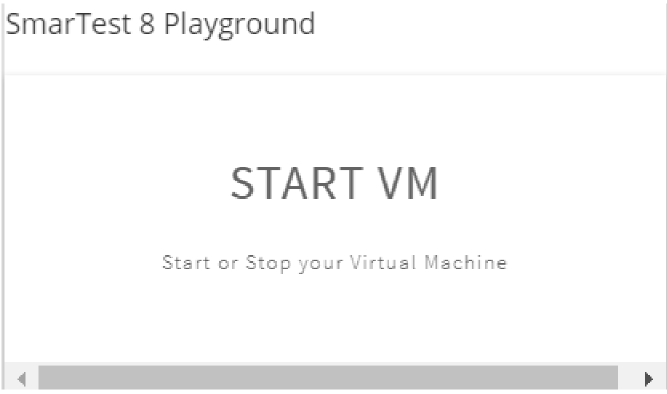
Access the V93000 Training page of the `myAdvantest Dojo` and locate the widget **SmarTest 8 Playground** on the right side of the page. If you have e-learning access and you cannot access the **SmarTest Playground**, please send a request to your Advantest representative.

About this task

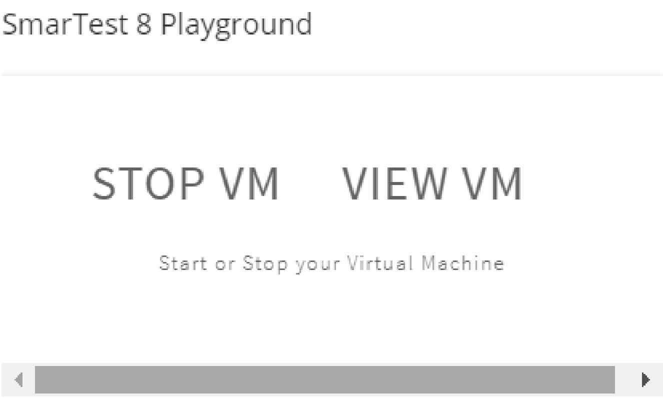
This task is about starting the **SmarTest Playground** and access the lab files.

Procedure

1. In the SmarTest 8 Training Center, click **START VM** to start the virtual machine assigned to your account.



2. Allow some time until the virtual machine is booted up. When ready you will see the entries **STOP VM** and **VIEW VM.**



3. Click **VIEW VM** to open the virtual machine. Now you have access to a virtual machine with linux, SmarTest and all files needed to execute your labs.
4. Start SmarTest Offline: From the task bar, click **Red Hat > Applications > Advantest V93000 > SmarTest offline**
5. When the Workspace Launcher is prompted, select or create a Workspace.

Results

SmarTest is running in the Playground and you can execute your labs.

What to do next

Verify your lab files.

Task 2 LMS: Verify and Import Setup Files

Locate the setup files you need to execute the labs.

Before you begin

Access the `V93000 Training Center` and start the **SmarTest Playground**.

About this task

Locate and identify the setup files needed to perform your offline labs. All files needed are located in your home directory under: ~/home/user/V93000_Trainings/LabFiles.

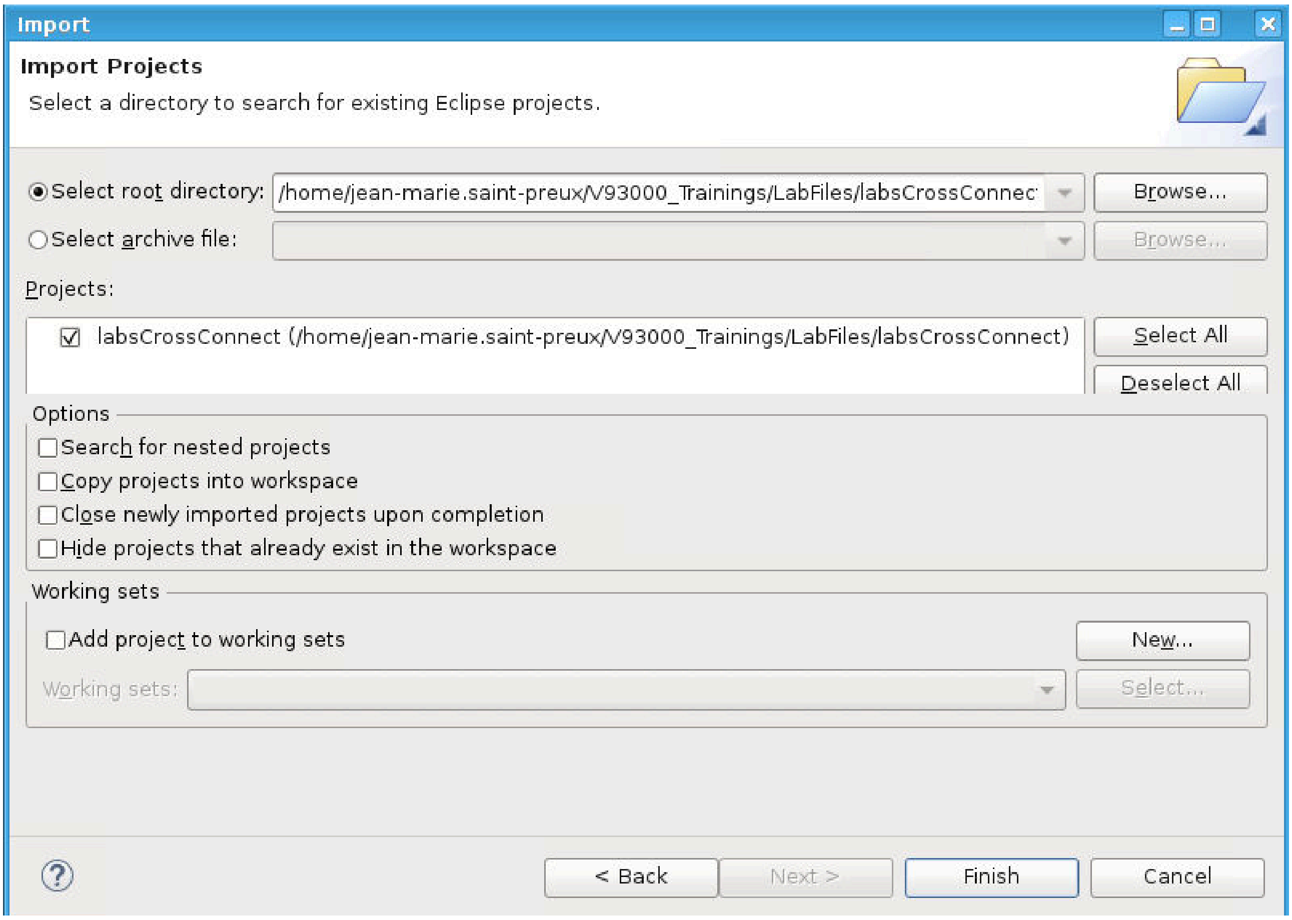
Procedure

1. Open a Konsole and locate the files required for your exercises.
2. Go to the folder `~/home/user/V93000_Trainings/LabFiles`
3. Use the commands `ll` or `ls` to verify the content of the folder. The following files are available:

labsCrossConnect.tar.gz	**Files to start from**
labsSolutionCrossConnect.tar.gz	Solution package
offline.model	offline model file
SmarTest_8_Basic_User_Training_Lab_Manual.pdf	Lab Manual with instructions

4. Use the linux commands `gunzip` and `tar xvf` or `tar zxvf` to expand the content of the tar package (For example tar zxvf labsCrossConnect.tar.gz).

5. Access the `SmarTest WorkCenter` in the SmarTest Playground and import the project `labsCrossConnect` into the `SmarTest WorkCenter` and follow the lab instructions.



Results

Now you are ready to follow the lab instructions as for the offline classroom. Execute your labs according to the assignment in your learning plan and the lab manual instructions. When done, you will be requested to upload some files or screenshots as proof of your learning progress. Take the required screenshots using your preferred tool. Make sure you save the file according the file name convention suggested by the labs. This helps to identify the single labs and tasks results for example Lab3_task2_upload.png. Go back to your Learning plan in the Training Center (LMS).

When done with your session, Exit SmarTest then Go to the Training Center in the LMS use **STOP VM** to stop the virtual machine. Note that the virtual machine will automatically shut down after 2 hours of inactivity.

What to do next

Follow the lab manual.

Task 3 LMS: Adjust Model File (optional)

If the virtual machine does not start with the correct model file, you may need to adjust the model file of your SmarTest Playground to be able to work with the training device.

Before you begin

Start the SmarTest Playground.

About this task

The default location of the model file tester.model is /etc/opt/hp93000/soc_common/. However, you can start SmarTest offline using a model file that is located in another folder.

Procedure

1. Open a system console and access the training files located in `/home/user/V93000_Trainings/Lab_Files`.
2. Locate the file offline.model.
3. Use the following command to setup the model file: `export V93000_MODEL=/home/user/V93000_Trainings/Lab_Files/offline.model`. User is your own user name.
4. Start SmarTest offline in the same shell using the command: `/opt/hp93000/soc/prod_env/bin/HPSmarTest -o&`

Results

SmarTest starts offline with your model file.

What to do next

Work on your lab.

Lab 1 Summary

What you have learnt

- How to determine the model file used by SmarTest 8 and how to start SmarTest 8.
- How to import projects into SmarTest 8.

Lab 2: DUT Board Description

Learning objective

In this lab you will learn how to work with the DUT board description file which specifies the electrical layout of the DUT board. A DUT board description includes specifications of the electrical connections between pogo pins and device pins, the numbers of sites, and the fixture delay data.

Getting started

Your project files are imported and available in the **Package Explorer**.

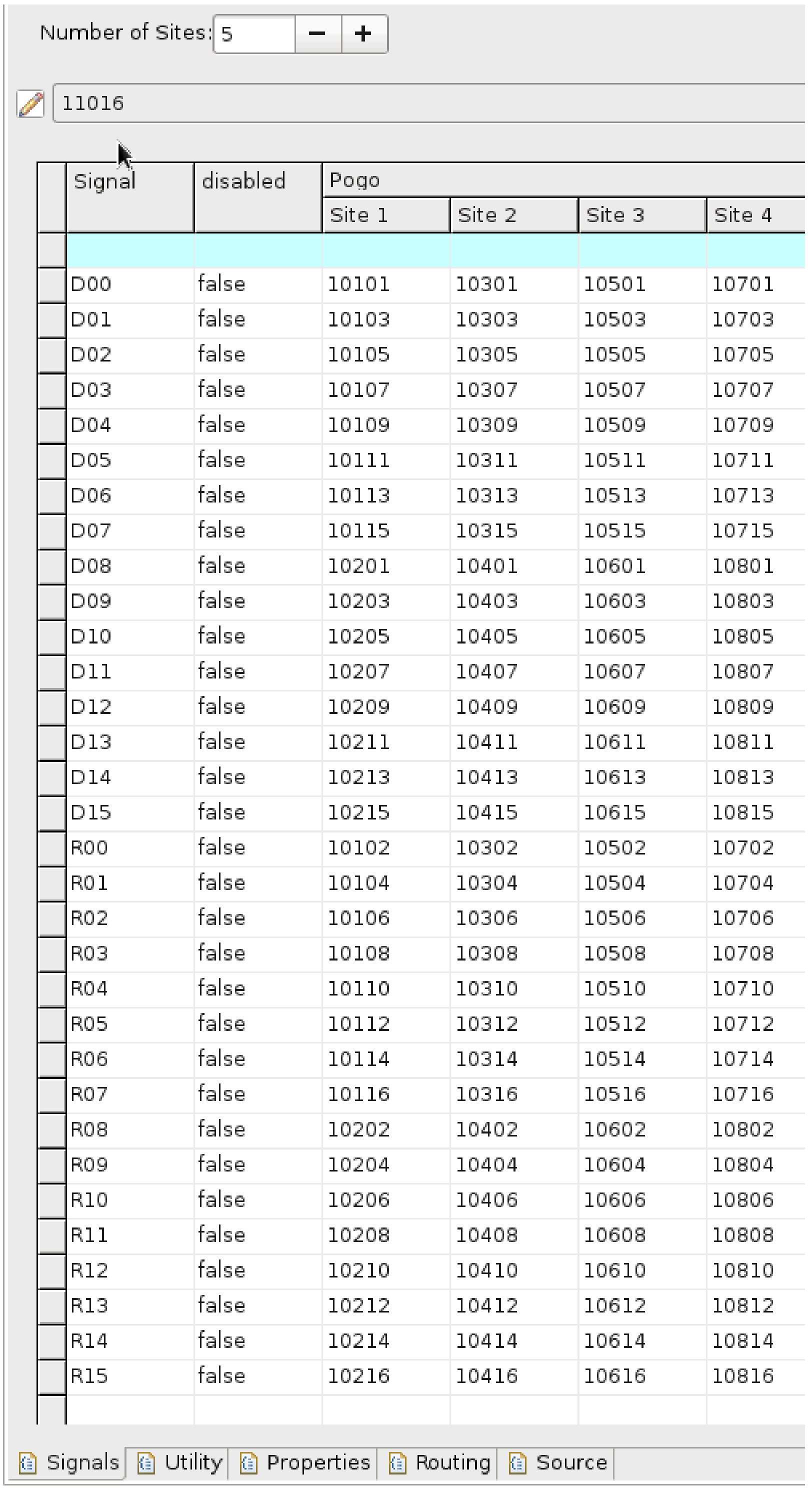
Task 1: Add Signals to the prepared DUT Board Description File

The given DUT board description contains configuration for sites 1 to 4. In this task, you will add the signal for the site 5 and specify how the signals of site 5 are mapped to pogo pins.

Procedure

1. In the **Package Explorer**, go to the folder `labsCrossConnect>src>CrossConnect>common` and locate the file `BasicLab.prog`. Open the DUT board description file from the test program file with **F3**: `crossconnect.common.BasicLab.dbd`.

2. Look at both, **Signals tab** and **Source tab**, and see correspondence of the contents between both tabs.



Signal	disabled	Pogo			
		Site 1	Site 2	Site 3	Site 4
D00	false	10101	10301	10501	10701
D01	false	10103	10303	10503	10703
D02	false	10105	10305	10505	10705
D03	false	10107	10307	10507	10707
D04	false	10109	10309	10509	10709
D05	false	10111	10311	10511	10711
D06	false	10113	10313	10513	10713
D07	false	10115	10315	10515	10715
D08	false	10201	10401	10601	10801
D09	false	10203	10403	10603	10803
D10	false	10205	10405	10605	10805
D11	false	10207	10407	10607	10807
D12	false	10209	10409	10609	10809
D13	false	10211	10411	10611	10811
D14	false	10213	10413	10613	10813
D15	false	10215	10415	10615	10815
R00	false	10102	10302	10502	10702
R01	false	10104	10304	10504	10704
R02	false	10106	10306	10506	10706
R03	false	10108	10308	10508	10708
R04	false	10110	10310	10510	10710
R05	false	10112	10312	10512	10712
R06	false	10114	10314	10514	10714
R07	false	10116	10316	10516	10716
R08	false	10202	10402	10602	10802
R09	false	10204	10404	10604	10804
R10	false	10206	10406	10606	10806
R11	false	10208	10408	10608	10808
R12	false	10210	10410	10610	10810
R13	false	10212	10412	10612	10812
R14	false	10214	10414	10614	10814
R15	false	10216	10416	10616	10816

Signals in the DUT board configuration file

3. Change Number of Sites to 5.
4. To avoid typing, proceed as follow:
   - **a.** Copy and Paste the content of the column Site 4 to Site 5
   - **b.** Go to the Source page of the DUT board description tool and use **Edit > Find/Replace**
   - **c.** **Find** site 5 { pogo = 107 ….**Replace with** Site 5 { Pogo = 109
   - **d.** **Find** site 5 { pogo = 108 ….**Replace with** Site 5 { Pogo = 110

5. Assign the pogo signal number 109xx and 110xx to site 5 sites using similarly **Find** site 5 { pogo = 107 and **replace with** site 5 { pogo = 109). Use the following picture as reference.

Signal	disabled	Pogo				
		Site 1	Site 2	Site 3	Site 4	Site 5
D00	false	10101	10301	10501	10701	10901
D01	false	10103	10303	10503	10703	10903
D02	false	10105	10305	10505	10705	10905
D03	false	10107	10307	10507	10707	10907
D04	false	10109	10309	10509	10709	10909
D05	false	10111	10311	10511	10711	10911
D06	false	10113	10313	10513	10713	10913
D07	false	10115	10315	10515	10715	10915
D08	false	10201	10401	10601	10801	11001
D09	false	10203	10403	10603	10803	11003
D10	false	10205	10405	10605	10805	11005
D11	false	10207	10407	10607	10807	11007
D12	false	10209	10409	10609	10809	11009
D13	false	10211	10411	10611	10811	11011
D14	false	10213	10413	10613	10813	11013
D15	false	10215	10415	10615	10815	11015
R00	false	10102	10302	10502	10702	10902
R01	false	10104	10304	10504	10704	10904
R02	false	10106	10306	10506	10706	10906
R03	false	10108	10308	10508	10708	10908
R04	false	10110	10310	10510	10710	10910
R05	false	10112	10312	10512	10712	10912
R06	false	10114	10314	10514	10714	10914
R07	false	10116	10316	10516	10716	10916
R08	false	10202	10402	10602	10802	11002
R09	false	10204	10404	10604	10804	11004
R10	false	10206	10406	10606	10806	11006
R11	false	10208	10408	10608	10808	11008
R12	false	10210	10410	10610	10810	11010
R13	false	10212	10412	10612	10812	11012
R14	false	10214	10414	10614	10814	11014
R15	false	10216	10416	10616	10816	11016

DUTboard file with 5 sites

6. Save the file (`Ctrl s`).
7. Click the **Source** and the **Signals tab** of the DUT board description file to verify the updated content.

Note	Use **Ctrl Shift F** to properly format your code.

Note	Use **Ctrl /** to toggle comment/uncomment for highlighted code block.

Lab 2 Summary

What you have learnt

- The purpose of the DUT Board Description File and how to edit the source code.
- How to make use of the tables.

Lab 3: Test Program

Learning objective

A test program includes a testflow, a DUT board description, test methods, and test data, among other things. In this lab you will practice some typical test program operations.

- Activate a test program.
- Check the contents of the test program.
- Execute a test program.
- Modify the main testflow of a test program.
- Disable a site.

Getting started

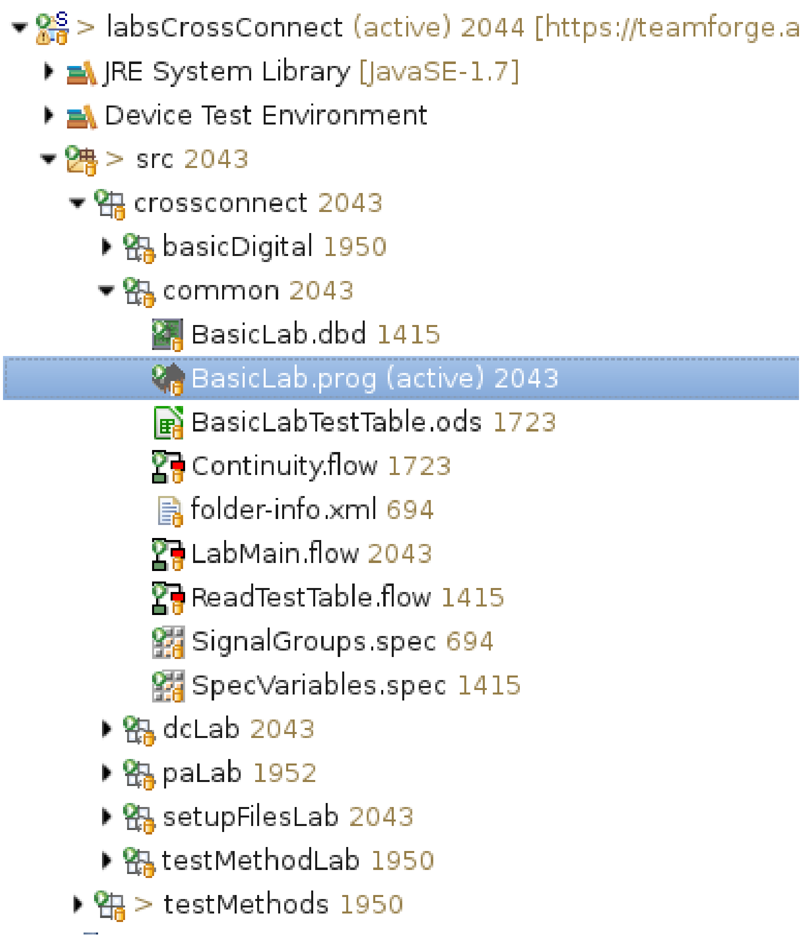
Make sure SmarTest is running in the OFFLINE mode and you completed the lab 2 DUT Board Description.

Task 1: Activate Test Program

The active test program is the one you have selected to work with. Only one test program can be active at anytime. In this task you will activate the `BasicLab.prog`. When a test program is activated, SmarTest has collected the test data belonging to the active test program and read the license requirement files. It has read the setup files into memory.

Procedure

1. In the **Package Explorer**, go the folder `labsCrossConnect>src>common` and locate the file BasicLab.prog.
2. Right click `BasicLab.prog` and select **Activate Test Program**.
3. The test program is marked (active) and the icons (coloration) change in the **Package Explorer.**



Test Program after activation

Task 2: Execute Test Program

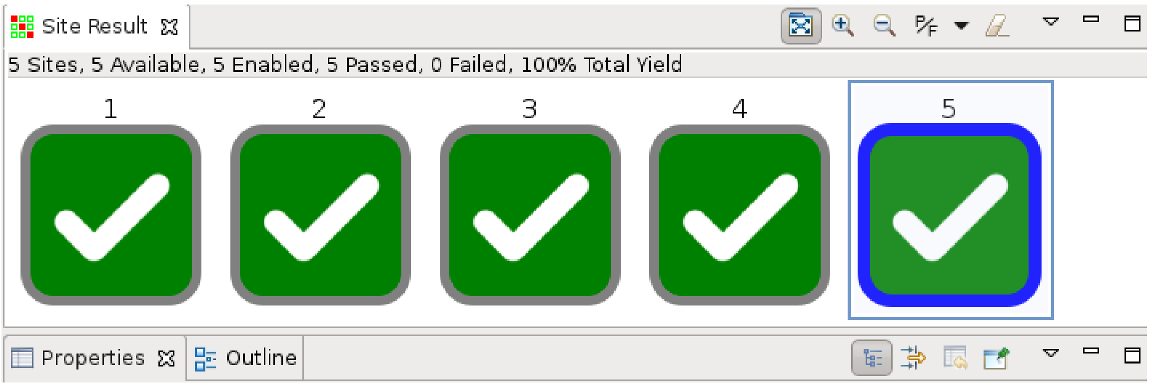
In this task you will execute a test program for the first time. Executing a test program with SmarTest is the process of preparing all corresponding test data, converting them into an executable test, and executing the test. If a SmarTest 8 session is online, it means that physically various changing voltages and currents are applied and measured according to the setups in the executed test program.

Procedure

1. Open `BasicLab.prog` and check what testflow is specified as main testflow. The following test program `BasicLab` uses the DUT board file `BasicLab` and the main testflow `FunctionalTests`.

```
testprogram BasicLab {
    dutboard = crossconnect.common.BasicLab;

    //
    // Lab Instruction: Change the main flow to "LabMain.flow"
    //
    testflow Main {
        flow = crossconnect.basicDigital.FunctionalTests;
    }

    testflow Continuity {
        flow =  crossconnect.auxiliaryFlows.Continuity;
    }

    //
    // Lab Instruction: Add the PowerUp flow
    //

    testflow PreBind {
       flow = crossconnect.auxiliaryFlows.ReadTestTable;
    }

    //
    // Lab Instruction: Ignore site 5
    //

}
```

2. Execute the test program. In the `Package Explorer`, right click `BasicLab.prog` and select **Run as > Test Program**.
3. Verify the **Site Result View** to ensure that all 5 sites are PASS. See picture below.



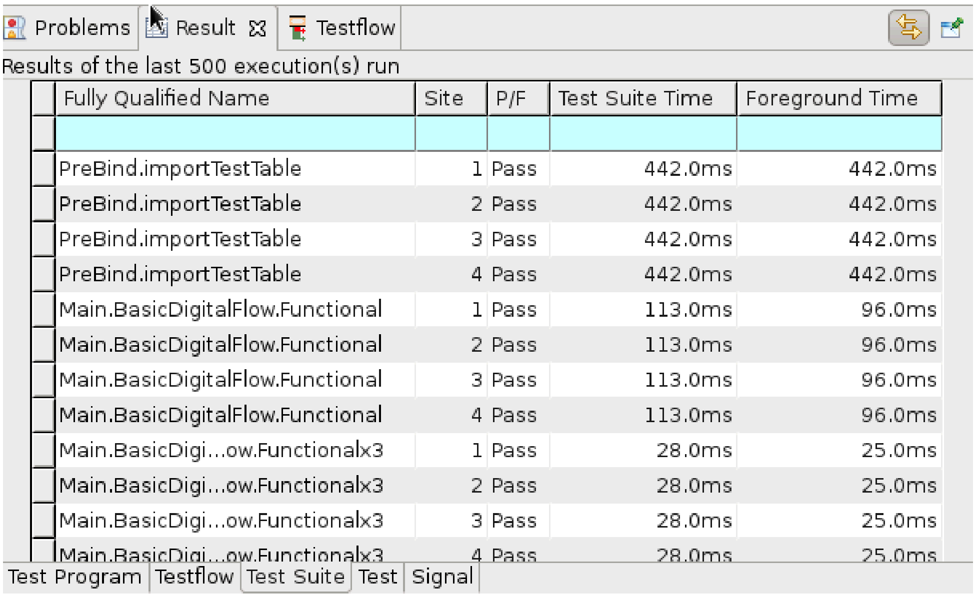
Site Result view with 5 passed sites

Task 3: Open Result View

The Result View is included by default in the Setup perspective and in the Debug perspective of the SmarTest Workcenter. Change perspectives using the buttons in the top right corner of the SmarTest Workcenter. The Result View is used to examine in detail the test results: The purpose of the Result View is to display the results that are written to the datalog.

Procedure

1. See the **Result View** and check each tab and see what information is shown.
2. The executed testflows are shown in the **Testflow** tab of the **Result View**. Make sure the testflows specified in the test program file are executed.
3. The **Test Suite** tab shows the executed test suites with the full paths of the execution hierarchy.
4. The **Test** tab shows all executed tests.
5. If a single site is selected in the **Site Result View** (blue box) then the **Result View** shows only data for this site.
6. Type into the light blue head line to filter data in the **Result View**.



Result View with executed testflow

7. You can use more filter capabilities of the **Result View** to influence the representation ot the data. Right click in the **Result view** and select **Preferences**, then select the columns you want to see and click **Apply**.

Note	When you set a specific site in focus in the **Site Result View**, it automatically sets a corresponding filter in the **Result View**. What you see in the Result View follows the site in focus.

Task 4: Modify and Execute Test Program

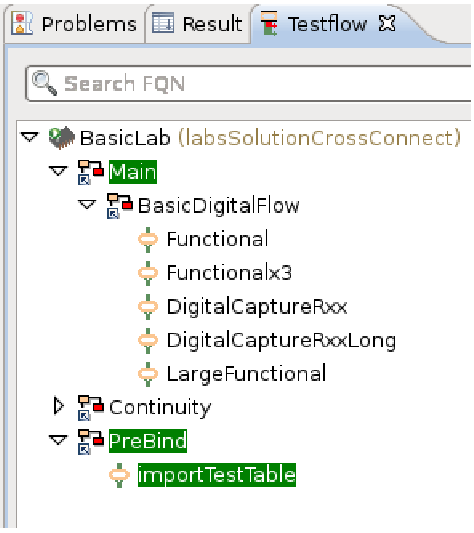
In this task you will change the assigned testflow and thus modifying the test program. Later you will ignore one of the configured sites and execute the test program again. Execution can be done in the Testflow View or in the Flow Chart.

Procedures

Assign testflow

1. Edit test program `BasicLab.prog` to change the main testflow. Use the content assist (`Ctrl Space`).

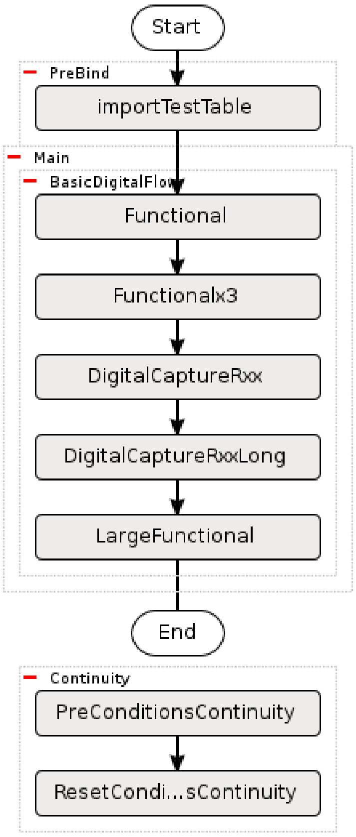
   `crossconnect.basicDigital.FunctionalTests > crossconnect.common.LabMain`
2. **Activate test program** `BasicLab.prog`. It is necessary to reflect the change in test program file.
3. Open the **Testflow view** and look at the elements (program, flow, suites) of the activated test program.



Testflow View with test suites

Note	Use the icon **Show Only Active Test Program** to filter out projects or programs you don t want to see.

4. Right click the `BasicLab.prog` in the **Testflow View** and select **Show In > Flow Chart** as shown in the next picture.

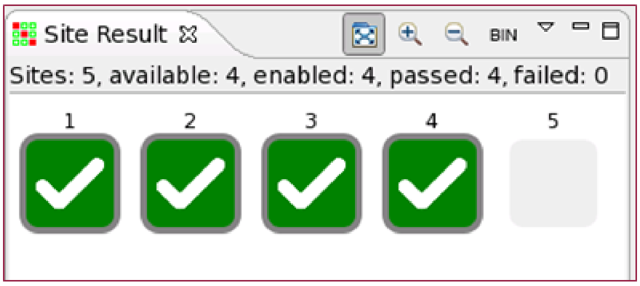


5. Use the **+/** icons in the Flow Chart to expand/collapse the representation of the flow.
6. Execute the Test Program in the Flow Chart using the **Run** icon.
7. Click the **Summarize** icon to check the execution summary.
8. In the Flow Chart, right click one test suite and select **Parameters** to view its parameters.
9. Right Click the main testflow and select **Show In New Flow Chart**. This opens a new instance of the Flow Chart and lets you focus on a specific flow.
10. Right click `BasicLab.prog` in the **Testflow View** and select **Run**. See also screenshot above.
11. Look at the **Result View** > `Test Suite tab/Testflow` tab and check what testflow was executed.

Ignore configured site

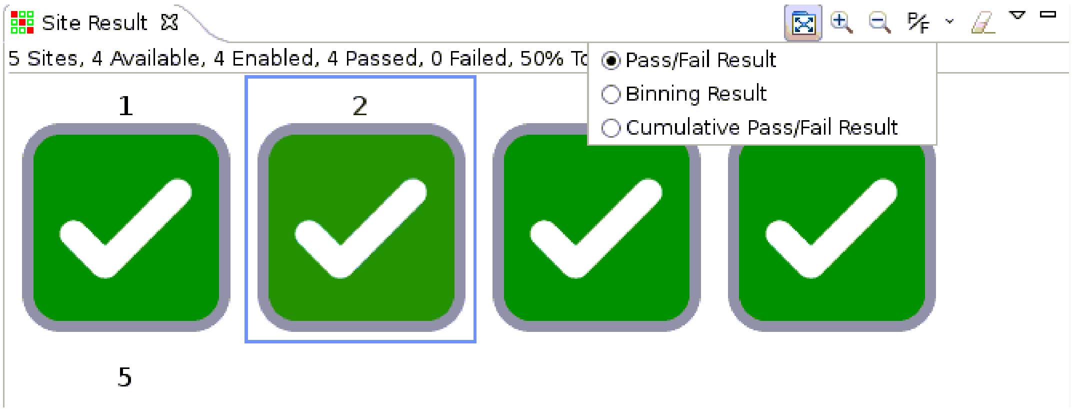
1. Open the file `BasicLab.prog.`
2. Use the content assist to an add `ignore configured site` statement and let SmarTest ignore site 5.
3. Save the test program file (Ctrl s).
4. Activate test program `BasicLab.prog.` It is necessary to reflect the changes in test program file.
5. Execute the test program from the **Testflow View** or in the **Flow Chart.**
6. Check the **Site Result view** and make sure the disabled site is not executed.

7. The results of the ignored site are not displayed in the Result view.



ignore site 5

8. In the `Site Result View`, experiment with the following settings: `Pass/Fail Result, Binning Result, Cumulative Pass/Fail Result.`
9. Use the filter functions of the `Site Result View` (right Click and Select Filter By State) to remove the ignored site 5 from the `Result View`.



Change Settings in Site Result View

Lab 3 Summary

What you have learnt

- How to activate and execute a test program.
- How to check, which testflows, test suites and tests of a test program have been executed and how to access results.
- How to view and execute a test program in the Testflow View and in the Flow Chart.

Lab 4: Testflow

Learning objective

A testflow consists of test suites, bins, and control structures that allow for conditional execution and loops. A testflow is a part of a test program. In addition to the main testflow, auxiliary testflows can be specified in a test program that are executed in response to certain events.

Getting started

Make sure SmarTest is running and you have completed the lab 3 `Test Program`.

Task 1: Add a Subflow to the Main Flow

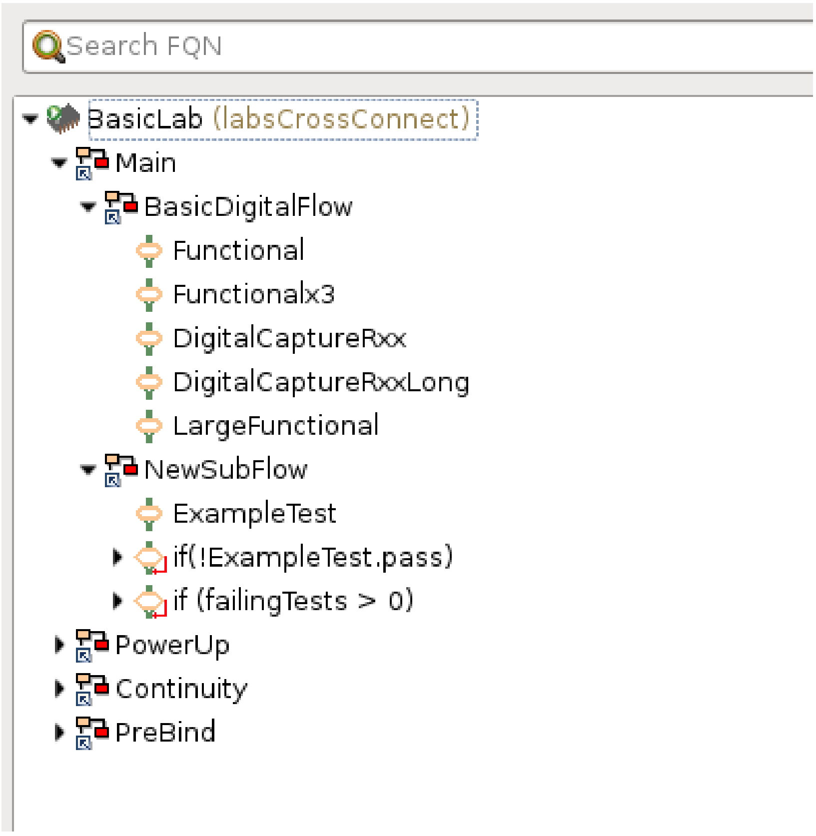
A subflow is a testflow that is part of another testflow. In this task you will add a subflow call to the main flow.

Procedure

1. Open the `LabMain.flow (with F3)` from the Test Program file.
2. Add a subflow calling statement for `LabMain.flow` in the `setup{}` section, and add subflow execution statement in the `execute{}` section.

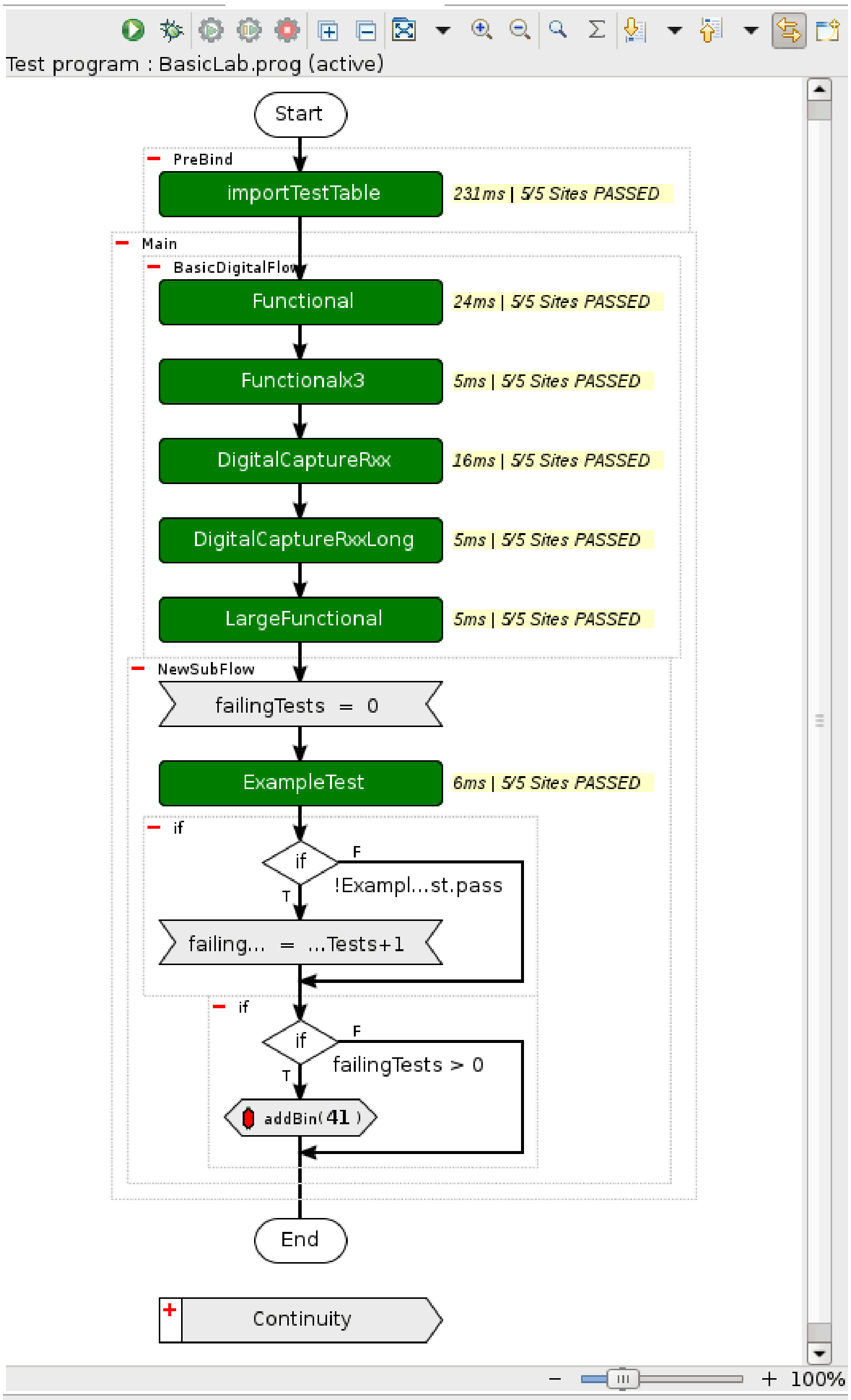
```
flow LabMain {
    setup {
        flow BasicDigitalFlow calls crossconnect.basicDigital.FunctionalTests {
        }
        flow NewSubFlow calls crossconnect.setupFilesLab.SetupFilesLab {
        }
    }
    execute {
        BasicDigitalFlow.execute();
        NewSubFlow.execute();
            }
}
```

3. Save the file.
4. Compare the content of the Test Program in the **Testflow View** and in the **Flow Chart**.



Testflow view with subflow content

5. Execute the test program `BasicLab.prog` from the Flow Chart using the green **Run** button.



Test Program Execution in the Flow Chart

6. Check the `Result View` and make sure the added subflow is executed.

7. Mark sites 1 to 4 in the Site **Result View** and select the testflow tab of the **Result View**. Note that the showed results are linked with the selected Sites.

Results of the last 60 minute(s) run

Fully Qualified Name	Site	SW Bin	P/F	DUT Time	Time	Testflow File
PreBind	1			N/A	71.0ms	crossconnect.common.ReadTestTable
PreBind	2			N/A	71.0ms	crossconnect.common.ReadTestTable
PreBind	3			N/A	71.0ms	crossconnect.common.ReadTestTable
PreBind	4			N/A	71.0ms	crossconnect.common.ReadTestTable
Main	1	1	Pass	3032.7ms	3028.6ms	crossconnect.common.LabMain
Main	2	1	Pass	3032.7ms	3028.6ms	crossconnect.common.LabMain
Main	3	1	Pass	3032.7ms	3028.6ms	crossconnect.common.LabMain
Main	4	1	Pass	3032.7ms	3028.6ms	crossconnect.common.LabMain
Main.BasicDigitalFlow	1			N/A	3016.9ms	crossconnect.basicDigital.FunctionalTe...
Main.BasicDigitalFlow	2			N/A	3016.9ms	crossconnect.basicDigital.FunctionalTe...
Main.BasicDigitalFlow	3			N/A	3016.9ms	crossconnect.basicDigital.FunctionalTe...
Main.BasicDigitalFlow	4			N/A	3016.9ms	crossconnect.basicDigital.FunctionalTe...
Main.NewSubFlow	1			N/A	11.1ms	crossconnect.setupFilesLab.SetupFilesLab
Main.NewSubFlow	2			N/A	11.1ms	crossconnect.setupFilesLab.SetupFilesLab
Main.NewSubFlow	3			N/A	11.1ms	crossconnect.setupFilesLab.SetupFilesLab
Main.NewSubFlow	4			N/A	11.1ms	crossconnect.setupFilesLab.SetupFilesLab

Result View with SetupFile subflow

Note	Use **Ctrl Shift F** to format your code properly.

Note	Use **Ctrl /** to toggle comment/uncomment for highlighted code block.

Lab 4 Summary

What you have learnt

- A testflow can call another (sub)flow and, as a consequence, the subflow can be called multiple times even by different parent flows.
- You should always remember that adding a subflow (and also test suite) to a test flow requires two steps:
  1. Add the definition of the testflow (or test suite) in the "setup" part.
  2. Add the place in the flow when to execute the testflow (or test suite) in the "execute" part.

Lab 5: Operating Sequence

Learning objective

An operating sequence is an arrangement of calls of patterns, actions, and transaction sequences to be executed that apply to one or more signals or signal groups. The arrangement can be serial, parallel, or a combination of both.

Getting started

This lab describes how to work with an Operating Sequence.

In this exercise you will learn how to:

- Add a new test suite to execute an operating sequence.
- Enhance the test suite setup so that it runs on additional signals.
- Add a pattern to the operating sequence and import corresponding specification files to the setup.

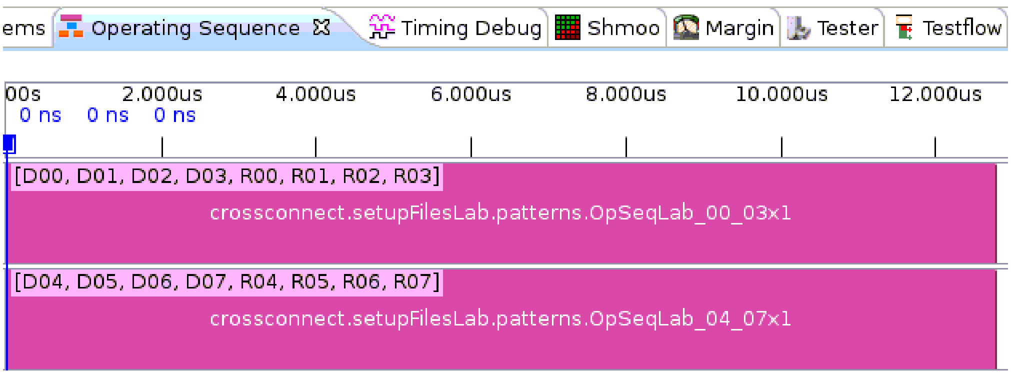
Make sure SmarTest is running and you completed the lab 4 "Testflow".

Task 1: Add a New Test Suite OpSeqLab

1. Open the testflow SetupFilesLab.flow (use F3 in the test program file). You may want to maximize the testflow view for this task.
2. Copy (using Copy and Paste) the existing test suite `ExampleTest`.
3. Change the test suite name to `OpSeqLab`.
4. Change the setup of the test suite so that it executes an operating sequence with a different specification name. Set the test method input parameter `testSignals` that defines which signals will return results:

```
suite OpSeqLab calls testMethods.acLib.FunctionalTest {
    measurement.specification =
 setupRef(crossconnect.setupFilesLab.mainSpecs.OpSeqLab);
    measurement.operatingSequence = setupRef(crossconnect.setupFilesLab.OpSeqLab);
    testSignals = "gR00_R07";
}
```

5. In the execute part add the statement `OpSeqLab.execute()`.
6. Save the testflow file and verify the testflow structure in the **Testflow view** and in the **Flow Chart**.
7. Execute the test program `BasicLab.prog` from the `Flow Chart` and check if it runs without errors.
8. Open the operating sequence file `OpSeqLab.seq` using F3 from the subflow. Open the corresponding pattern (using F3 from the operating sequence file).
9. Check the contents:
    a. Called patterns.
    b. What signals are used to execute the pattern?
    c. How the patterns are executed? (Parallel? Serial?)
    d. Open the main spec file `OpSeqLab.spec`
    e. Check the contents.
    f. Make sure the signals defined in the level and timing spec are exactly the same as in the called patterns.
10. Start a debug session: From the **Testflow View** start the test program in Debug mode. If needed, confirm the dialog to switch to the debug perspective. Use Expand to see all testflows and test suites. In debug mode, right-click on the test suite and select **Execute**. Alternatively you can use the Debug icons of the **Testflow View**.
11. Check the executed operating sequence in the **Operating Sequence View**.



Operating Sequence with 2 parallel groups

12. Check the property page of the **Operating Sequence view**.
13. While on one of the pattern rows of the **Operating Sequence View**, Right Click and select **Open with Pattern Editor** to navigate from the **Operating Sequence View** to the **Pattern Editor**.
14. From the **Operating Sequence View**, right click and select **Show in Measurement View** and verify that the settings shown in the Measurement View are correctly evaluated from the values and equations given in the specification files
15. Use the button to terminate the debug session.

Task 2: Add a Pattern Call to an Operating Sequence

1. Open operating sequence file `OpSeqLab.seq`.
2. Add a pattern call `OpSeqLab_08_15x1.pat` to be executed in parallel as shown below.

```
sequence OpSeqLab {
    parallel parGrp1 {
        sequential seq1_1 {
            patternCall crossconnect.setupFilesLab.patterns.OpSeqLab_00_03x1;
        }
        sequential seq2_1 {
            patternCall crossconnect.setupFilesLab.patterns.OpSeqLab_04_07x1;
        }
        sequential seq3_1 {
            patternCall crossconnect.setupFilesLab.patterns.OpSeqLab_08_15x1;
        }
    }
}
```

3. Save the file.
4. Check which signals are used in the new pattern.

Task 3: Import Additional Specification Files

1. Open the main spec file `OpSeqLab.spec`
2. Add statements to import level and timing specifications import for the 08-15 signals D08, .., D15 and R08, .., R15.

```
import crossconnect.setupFilesLab.levels.OpSeqLabLevel_08_15;
import crossconnect.setupFilesLab.timings.OpSeqLabTiming_08_15x1;
```

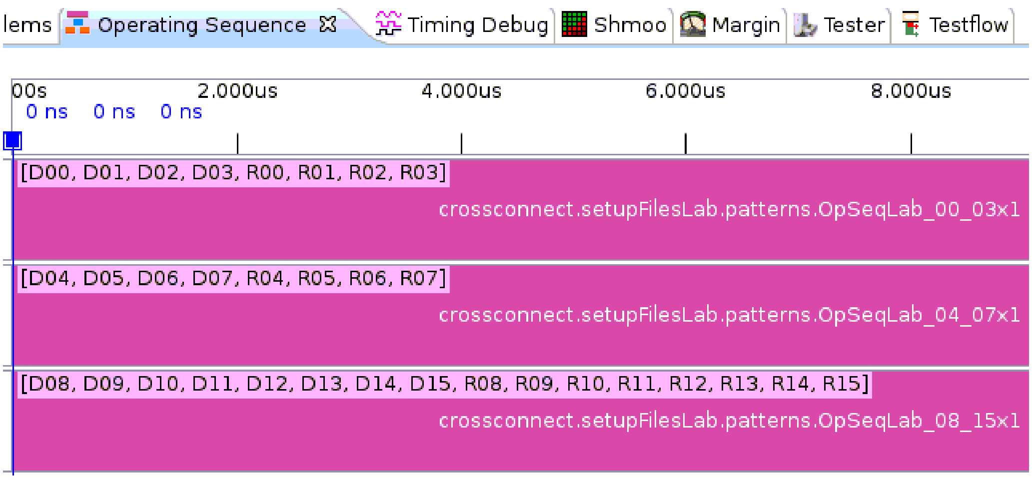
3. Add the group gD08_D15 + gR08_R15 to the setup group to configure Pass/Fail results:

```
setup digInOut gD00_D03 + gR00_R03 + gD04_D07 + gR04_R07 + gD08_D15 + gR08_R15
```

4. Execute the testflow and check if it runs properly.
5. Start a debug execution: From the **Flow Chart** start the test program in Debug mode using the debug icon in the Flow Chart.
6. Use Expand to see all testflows and test suites. In debug mode, right-click on the test suite and select **Execute**.
7. Check with the Timing Debug view and the operating sequence view:

   a. If the added patterns for the signals D08, .., D15 and R08, .., R15 are included..

   b. If the patterns for the signals D08, .., D15 and R08, .., R15 are not shown, right click in the view and select **Reset Layout**.



Operating Sequence with 3 parallel groups

8. Use the  button to terminate the debug session.

Lab 5 Summary

What you have learnt

- How to add a new test suite to a test flow.
- How to modify a test suite setup so that it runs on additional signals:
    1. You have to setup the additional signals in the specification.
    2. You have to enhance the operating sequence to define what should run on the additional signals.
    3. How to debug from the Flow Chart.

Lab 6: Timing Set/Level Set

Learning objective

The measurement specification file contains the measurement specification information and has the file name extension .spec. The specification file must be saved in the source folder of a project. In this exercise you will modify timing and level settings given in specification files. You will create an additional specification file by copying an existing one.

A specification file contains information such as:

- Declaration of the variables used in the level or timing setups.
- Definition of signal aliases or signal groups.
- Definition of waveforms.
- Setups of the instruments used in the measurement.
- Instrument settings and properties.
- Wavetable definition.
- Level and timing sets.
- Actions definitions.
- Setups of protocols for protocol-aware testing.

Getting started

Make sure SmarTest is running and you have completed the lab 5 "Operating Sequence".

Task 1: Make use of the Default Wavetable

In this task, you will learn how to use the default wavetable in SmarTest 8 for the timing setup of a new test suite.

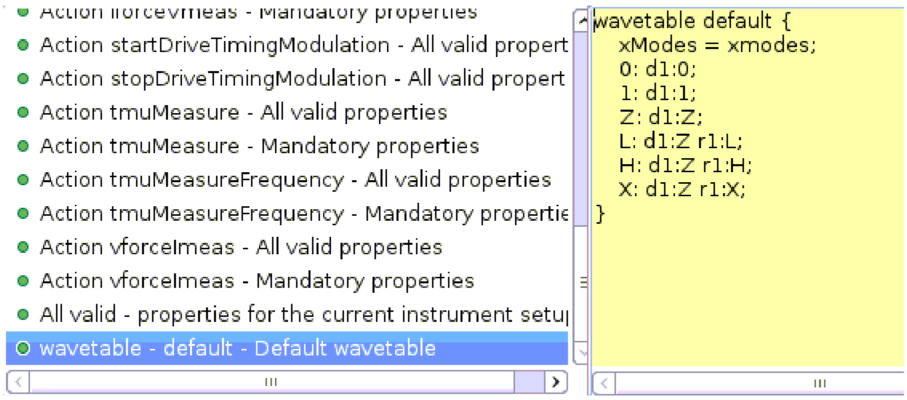
1. Open testflow `SetupFilesLab.flow` from the LabMain.flow.
2. Add a test suite `SpecFileLab` as follows:

```
suite SpecFileLab calls testMethods.acLib.FunctionalTest {
            measurement.specification =
 setupRef(crossconnect.setupFilesLab.mainSpecs.SpecFileLab);
            measurement.operatingSequence =
 setupRef(crossconnect.setupFilesLab.SpecFileLab);
        }
```

3. Add a corresponding execute statement to the execution section of the testflow.

```
SpecFileLab.execute ();
```

4. Save the file.
5. Open the specification file SpecFileLab.spec (using F3). Use F3 to access the timing file (see import section).
6. Use the content assist to add the default wavetable to all setup sections of `SpecFileLabTiming`. Set the xModes property to 1.



Adding Default Wavetable

7. Execute the testflow and check if it passes by checking the **Site Result view**.

Task 2: Change Timing Period

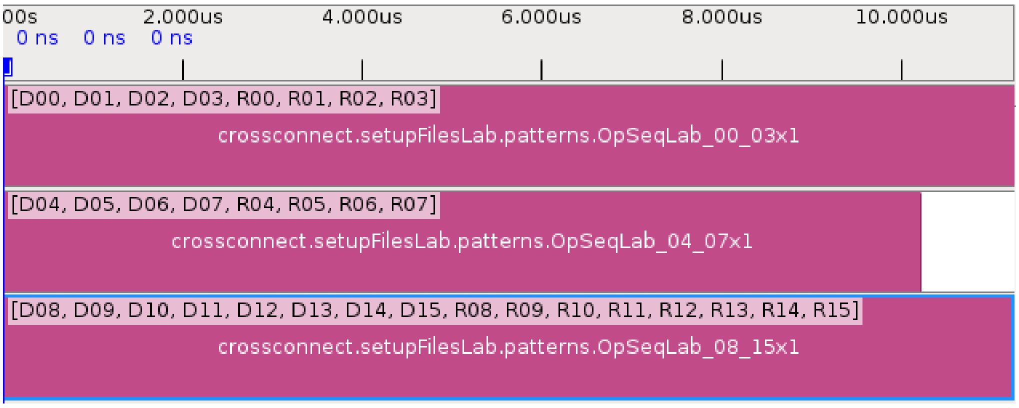
In this task, you will modify the device period in the specification file, so that three groups of signals with different periods are specified. Then you will run an operating sequence that calls a dedicated pattern for each signal group.

1. Use the navigation F3 from the testflow to open the specification file `crossconnect.setupFilesLab.mainSpecs.SpecFileLab`
2. In the import section, open the file `SpecFileLabTiming`
3. Change the period entries as shown below.

```
import crossconnect.common.SignalGroups;
spec SpecFileLabTiming {
    set timingSet_1;
    var Time per;
    var Time t_drive;
    var Time t_exp;
    setup digInOut gD00_D03 + gR00_R03 {
        set timing timingSet_1 {
            period = per;
            d1 = t_drive;
            r1 = t_exp;
        }
         wavetable default {
            xModes = 1;
            0 : d1 : 0;
            1 : d1 : 1;
            Z : d1 : Z;
            L : d1 : Z r1 : L;
            H : d1 : Z r1 : H;
            X : d1 : Z r1 : X;
        }
              }
    setup digInOut gD04_D07 + gR04_R07 {
        set timing timingSet_1 {
            period = per * 0.8; // Lab: change period here
            d1 = t_drive;
            r1 = t_exp;
        }
         wavetable default {
            xModes = 1;
            0 : d1 : 0;
            1 : d1 : 1;
            Z : d1 : Z;
            L : d1 : Z r1 : L;
            H : d1 : Z r1 : H;
            X : d1 : Z r1 : X;
        }
            }
    setup digInOut gD08_D15 + gR08_R15 {
        set timing timingSet_1 {
            period = per * 1.2; // Lab: change period here
            d1 = t_drive;
            r1 = t_exp;
        }
         wavetable default {
            xModes = 1;
            0 : d1 : 0;
            1 : d1 : 1;
            Z : d1 : Z;
            L : d1 : Z r1 : L;
            H : d1 : Z r1 : H;
            X : d1 : Z r1 : X;
        }
      }
}
```

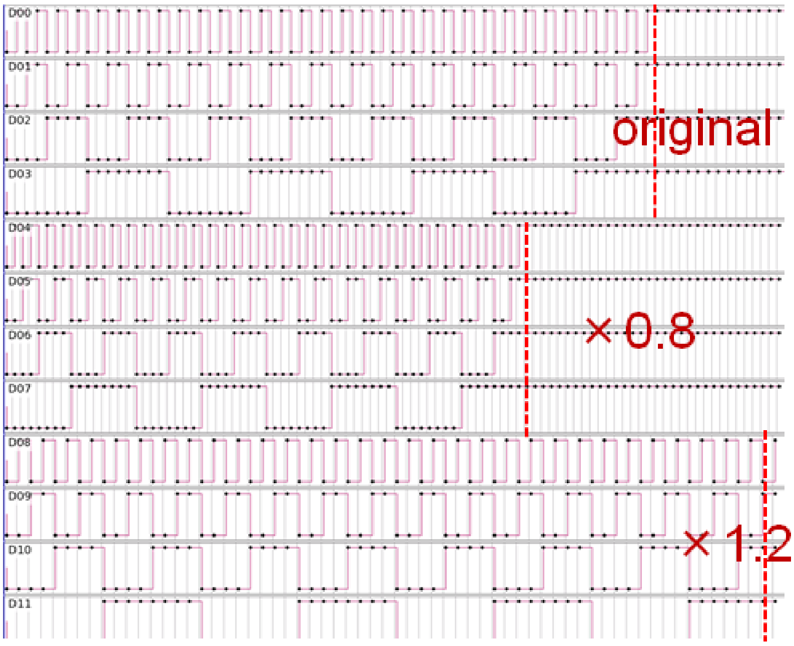
4. Save the file.

5. From the **Testflow View** or from the **Flow Chart** start Debug for the test program and execute the test suite `SpecFileLab`.
6. Use **Expand All** to see all testflows and test suites.
7. If prompted to acknowledge the change of Perspective switch dialog, confirm it.
8. Verify the following items in the **Measurement View** and its tabs: **Instrument, Level, Timing, Result**.
9. Check that the **Operating Sequence View** shows three lanes.
10. Confirm the names of the executed patterns in the Operating Sequence View.



Operating sequence view showing all pattern with different period as lanes

11. Check the pattern(waveform) using the **Timing Debug View**. Make sure the period for of each pattern is reflected in the **Property View** of the Timing Debug view: 200 ns, 160 ns, 240 ns.
12. Terminate the debug mode when completed.



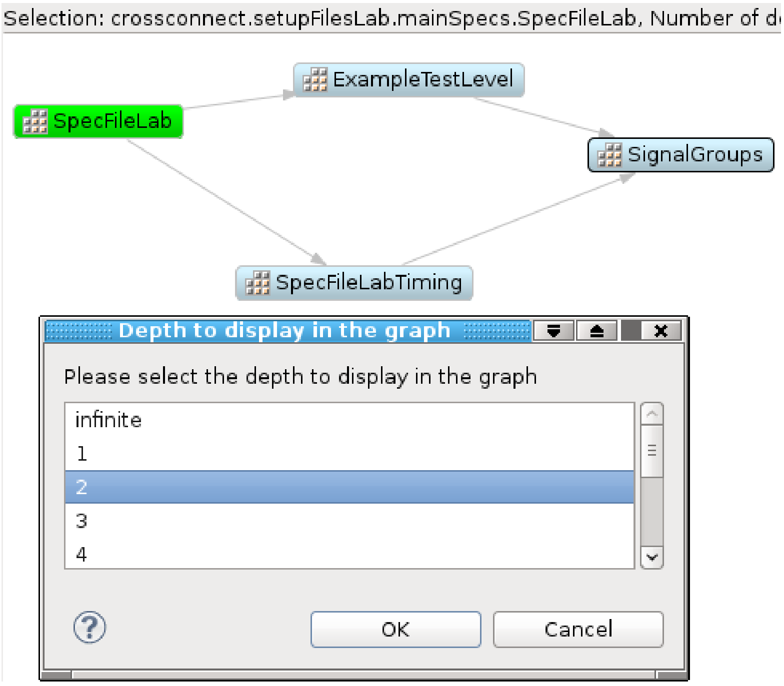
Period of specidied patterns

Note	The Cycle numbers are shown in the Timing Debug View. Right Click on a Signal and select **Show Cursor Label**.

Task 3: Graphically Visualize Dependencies of the Level Setup

In this task you will learn, how to use the Setup Dependency Graph to graphically visualize the references between specification files (and other files like pattern files, operating sequence files, testflow files and the test program file).

1. Make sure you are in the **Setup Perspective**.
2. Use the navigation F3 from the `testflow to` open the specification file `crossconnect.setupFilesLab.mainSpecs.SpecFileLab`
3. Check what variables are defined for level setting.
4. Open the imported level specification file. (Use **F3** key)
5. Note how the specified variables are used for the level set.
6. Check dependencies of imports in the **Setup Dependency Graph** with depth=2. To open the **Setup Dependency Graph**, use the **Window > Show View > other** and Search for the **Setup Dependency Graph**. Alternatively, you can use the context menu of a setup file and select **Show In...**
7. To setup the depth, right click the icon **Set the depth of the graph** and select the corresponding level.
8. Modify settings of the tool such that you see also the test suites that use the specification file `crossconnect.setupFilesLab.mainSpecs.SpecFileLab`.



Setup Dependency Graph

Task 4: Create a New Specification File

In this task, you will modify the level settings of the device in the specification file. In the next task you will execute the test and in task 6 you will verify the modified level settings using the debugging tools.

1. Copy the file `ExampleTest.spec` located in setupFilesLab > mainSpecs > ExampleTest.spec
2. Paste it with a new name of `SpecFileMaxVcc.spec.`
3. Use quick-fix (Ctrl-1) to correct the spec name.
4. Change the value of the level set related variables
   a. vcc: 1.4 V > 1.8 V
   b. IV_Swing: 1.4 V >1.8 V
   c. OV_Swing: 0.4 V > 0.5 V
5. Save the file.

Task 5: Add another Test Suite that uses Maximal Level Settings

1. Open the testflow `SetupFilesLab.flow`.
2. Add a test suite `SpecFileMaxVcc` in the `setup{}` section.
   - Copy the existing test suite `ExampleTest`.
   - Change the test suite name to `SpecFileMaxVcc`.
   - Change specification name to `SpecFileMaxVcc`.

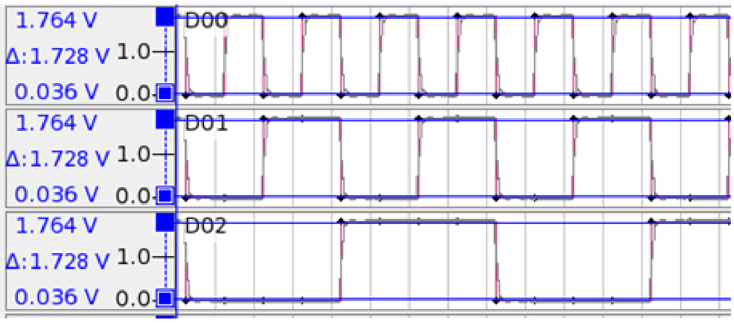
   when done, your new test suite will looks like:

```
        suite SpecFileMaxVcc calls testMethods.acLib.FunctionalTest {
            measurement.specification =
setupRef(crossconnect.setupFilesLab.mainSpecs.SpecFileMaxVcc);
            measurement.pattern =
setupRef(crossconnect.setupFilesLab.patterns.Counting_00_15x1);
        }
```

3. Add the test suite `SpecFileMaxVcc` to the execute{} section.
4. Execute the testflow in debug mode and check if it runs without failure.

Task 6: Verify Level Settings Using the Timing Debug View and Measurement View

1. Start a debug execution and execute the `SpecFileLabMaxVcc`. (Debug the testflow `NewSubFlow`, right click on test suite SpecFileLabMaxVcc and hit execute).
2. The Debug perspective opens.
3. Open **Timing Debug view**. (Test suite: `SpecFileMaxVcc`)
4. Run Scope.
5. Use the Cursor and timer markers and verify that the level is correctly shown. The property values are shown in the Property pane. In offline mode, the waveform of the scope may not correctly show the level.
6. Press Resume button in Debug tab (or press F8).



Scope Levelset

7. Use the Property page of the Timing debug view to verify voltage low, voltage high and range.
8. Open the **Measurement View** and open the tab **Specification Variables**. Ensure that the values are set as expected.
9. Terminate the debug session.

Lab 6 Summary

What you have learnt

- How to make use of the default wavetable.
- How to visualize the dependencies between the various setup files.
- How to start a debug mode, execute a single test suite and see different periods of patterns in the Timing Debug View.
- How to setup different clock periods for different patterns/groups of signals.
- How to utilize existing setup files to setup a new test suite with slightly different settings.

Lab 7: Timing Debug

Learning objective

The `Timing Debug View` displays drive and receive data, actions, and compare data in signal rows or signal groups.

Getting started

Make sure SmarTest is running and you finished lab 6 "Timing Sets/Level Sets".

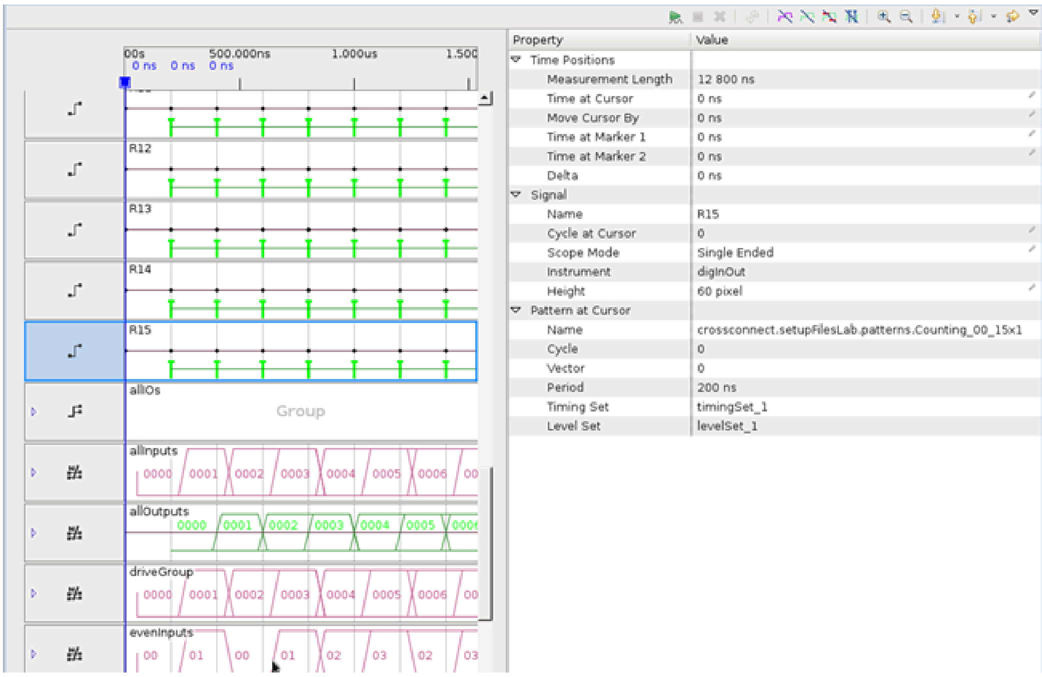
Task 1: Prepare the Testflow for Timing Debug View

In this task, you will start a debug session from the Testflow view and observe waveforms in the Timing Debug View.

1. Open testflow file `SetupFilesLab.flow` (use F3 in LabMain.flow).
2. In the **Testflow View**, Right click the subflow `NewSubFlow` and select **Expand**.
3. Right click the `NewSubflow` and select **Debug**.
4. Right click the test suite `ExampleTest` and select **Execute**. Confirm it opens the Debug Perspective.
5. Keep this status for the following steps. This is needed to use the **Timing Debug View** and other debugging tools.

Task 2: Working with the Timing Debug View

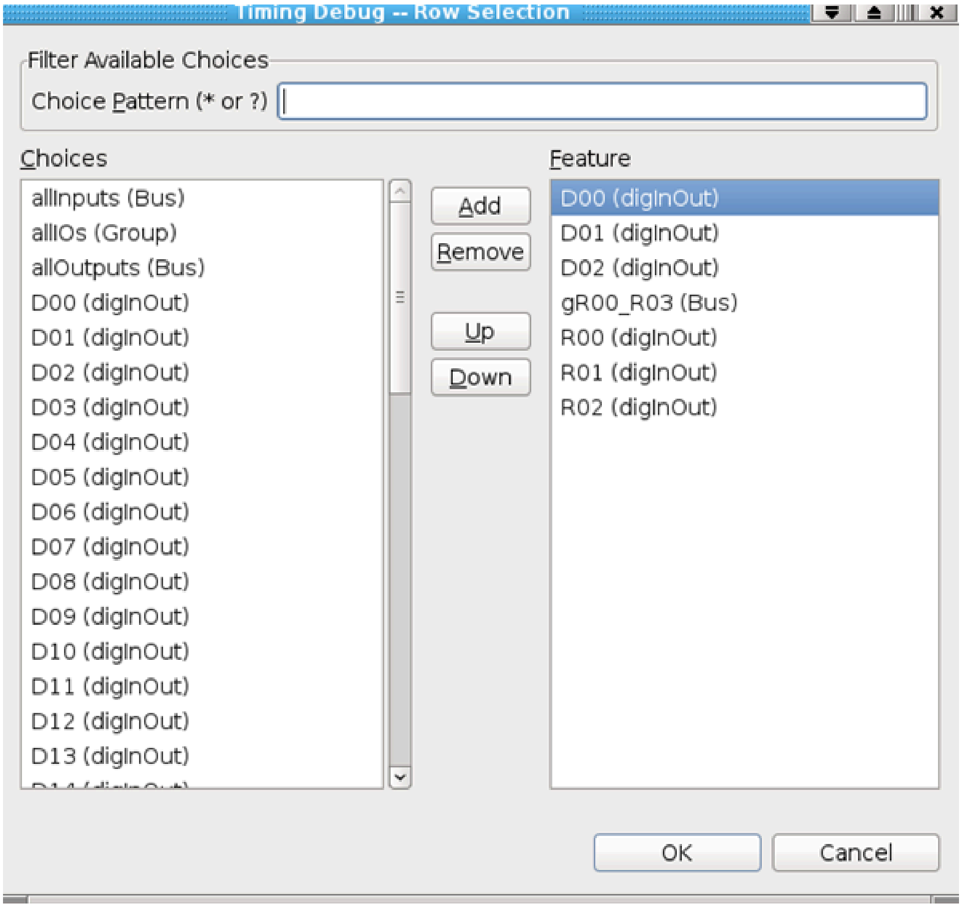
1. Open the **Timing Debug View**.
2. Observe the representation of signals in the **Timing Debug View:** Drive signals and Expect signals.



Signal representation in Timing Debug View

3. Try Zoom In and Zoom Out. Zoom Icons or `CTRL-<mouse scroll>`.
4. Use **Maximize** and **Restore** to change the size of the view. See TDC# 246076 for details info about the meaning of the icons.
5. Click a signal and note its properties in the **Property View**.
6. Switch between the different sites in the **Site Result View** and observe the Timing Debug view. All Debug tools are multi site aware.

7. Right Click and open the **Configure Layout** GUI to select the corresponding signals. Layouts help to focus on signals of interest.



Layout in Timing Debug view

8. Run a Scope and see the waveforms.
9. Hide Drive or receive signals.
10. Note the representation of signal groups. Unfold or expand signal group to single signals and back.
11. Right Click and select **Show Cursor Label** to see the Cycles Number and Level values of specific signals.
12. Move Cursor and see the corresponding cycle number of a selected signal.
13. Terminate the debug execution.

Lab 7 Summary

What you have learnt

- How to work with the Timing Debug View.

Lab 8: X-Mode

Learning objective

An x-mode is an operating mode of a tester in which a tester cycle contains multiple device cycles. X-modes are specified by the number of device cycles in each tester cycle. For example, in the X4 mode there are four device cycles in a tester cycle.

Getting started

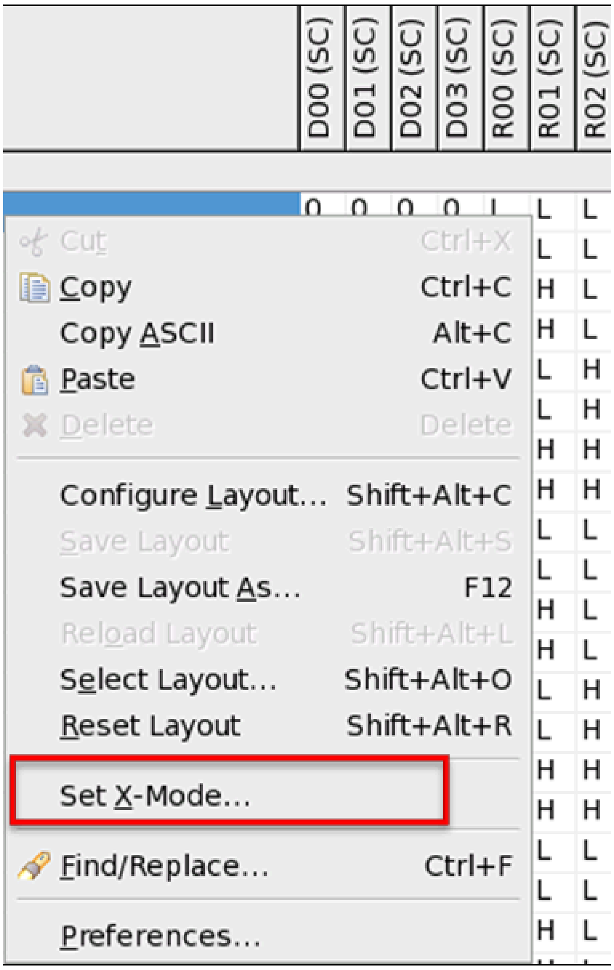
Make sure SmarTest is running and you finished lab "Timing Debug".

Task 1: Setting a Higher x-Mode for a Pattern

You will change the `x-mode` of a pattern from x1 mode to x2 mode. Then you will modify the corresponding `operation sequence` and `specification file` to match the `x2-mode` and execute the test. You will use the test suite `SpecFileLab` from the testflow `SetupFilesLab`.

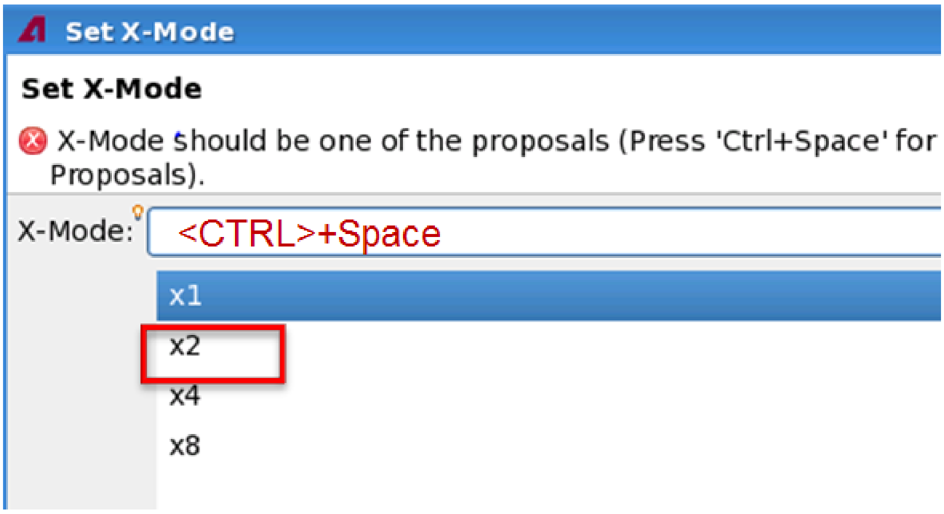
Procedure

1. Copy the pattern file using the `Package Explorer`.
   a) Select the pattern `OpSeqLab_00_03x1.pat`.
   b) Right click and select **Copy** then **Paste**.
   c) Specify the name of the target pattern `XModeLab_00_03x2.pat`.
2. Open the pattern file with the default editor, i.e. Pattern Debug, by double-clicking on the file in the `Package Explorer`.
3. Right click on the Pattern Editor showing the pattern and select **Set X-mode....**



Setting x-mode

4. Choose **x2** mode and save the file (`Ctrl Space`).



Select available x-mode

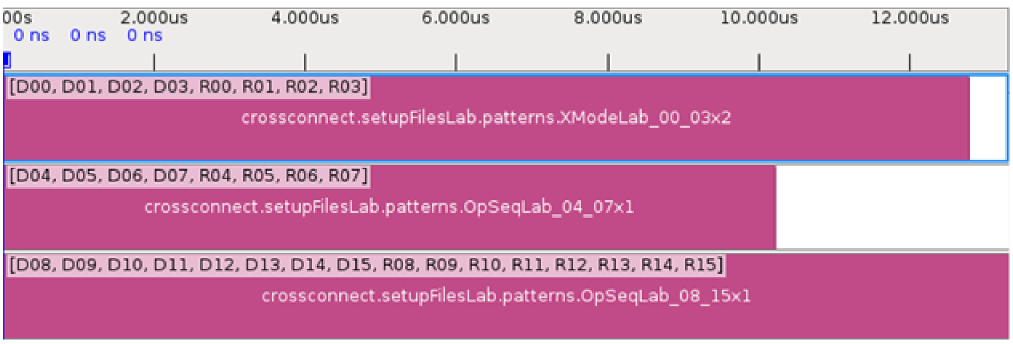
5. Open the operating sequence file `SpecFileLab.seq`. (Testflow: SetupFilesLab > Test Suite: SpecFileLab)
6. Replace the pattern to call (as shown below)

```
crossconnect.setupFilesLab.patterns.OpSeqLab_00_03x1 >
crossconnect.setupFilesLab.patterns.XModeLab_00_03x2
```

7. Save the file.
8. Open timing file `SpecFileLabTiming.spec`, set `xModes=2` for the signals digInOut gD00_D03 + gR00_R03 as shown below.

```
setup digInOut gD00_D03 + gR00_R03 {
        set timing timingSet_1 {
            period = per;
            d1 = t_drive;
            r1 = t_exp;
        }
         wavetable default {
            xModes = 2;
            0 : d1 : 0;
            1 : d1 : 1;
            Z : d1 : Z;
            L : d1 : Z r1 : L;
            H : d1 : Z r1 : H;
            X : d1 : Z r1 : X;
        }
```

9. From the **Testflow View**, execute the testflow in debug mode. Check the `Operating Sequence view`.



Operating Sequence with x2 Setting

10. Check the pattern using `Timing Debug view`. Make sure the waveform is same as that of the X1 mode in the previous lab.
11. Open the `Tester view` and select the **digInOut** tab at the bottom. Check the values in the columns **Period** and **Sequencer period** for various signals.
12. Open the `Measurement view`, select the tab **digInOut** at the bottom and the tab **Timing** at the top. Check the device cycle period of the signals.
13. Terminate the debug session.

Lab 8 Summary

What you have learnt

- How to set an x-Mode higher than 1 in a pattern
- How to change wavetables in specification files, such that a certain higher x-Mode is supported.

Lab 9: Pattern Debug

Learning objective

The pattern debugger can be used in debug mode for the last executed measurement. The pattern debugger can be used to:

- Display a pattern of your loaded setup.
- Debug the vectors, instructions, and anchored actions that the pattern contains.
- Display the pass/fail results and other measurement information.

This lab is about checking a failing pattern using the following debug tools

- Result view.
- Measurement view.
- Error Map.
- Timing Debug view.
- Pattern Debugger.

Getting started

Make sure SmarTest is running and you finished the lab "Timing Debug".

Task 1: Prepare the Operating Sequence

The pattern to be executed is typically contained in an operating sequence. In this task, you will execute an operating sequence that contains a failing pattern. Further, you will use the Pattern debugger to identify and fix the failures.

Procedure

1. Open the operating sequence `OpSeqLab.seq` from the testflow `SetupFilesLab` and test suite `OpSeqLab`.
2. Add the failing pattern `DebugLab_Fail_04_07x1_Fail` to the sequencial group `seq2_1`.

```
sequence OpSeqLab {
    parallel parGrp1 {
        sequential seq1_1 {
            patternCall crossconnect.setupFilesLab.patterns.OpSeqLab_00_03x1;
        }
        sequential seq2_1 {
            patternCall crossconnect.setupFilesLab.patterns.OpSeqLab_04_07x1;
            patternCall
 crossconnect.setupFilesLab.patterns.DebugLab_Fail_04_07x1_Fail;
        }
        sequential seq2_3 {
            patternCall crossconnect.setupFilesLab.patterns.OpSeqLab_08_15x1;
        }
    }
}
```

3. Execute the testflow in the Testflow View.
4. Check the Site **Result view**.

Task 2: Debug a Failing Pattern (ONLINE)

You will use the navigation capabilities of the debug tools to identify and fix failures in a pattern. To move from one debug tool to another use the Show In...feature when applicable.

Procedure

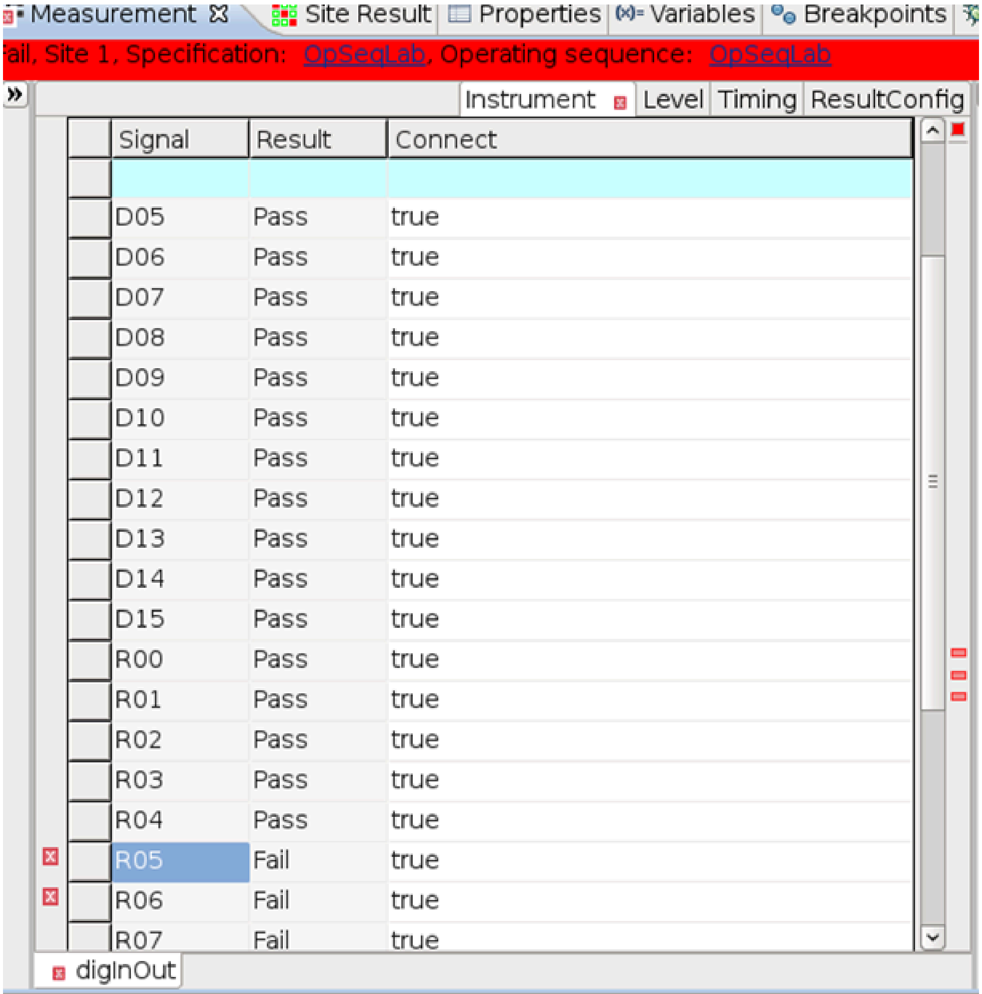
1. Open the `Result view` and check the failing pattern using the `Testsuite tab`.
2. Locate the fail annotations.

PreBind.exportTestTable	3	Pass	36.5ms	36.5ms
PreBind.exportTestTable	4	Pass	36.5ms	36.5ms
Main.NewSubFlow.ExampleTest	1	Pass	4.0ms	3.7ms
Main.NewSubFlow.ExampleTest	2	Pass	4.0ms	3.7ms
Main.NewSubFlow.ExampleTest	3	Pass	4.0ms	3.7ms
Main.NewSubFlow.ExampleTest	4	Pass	4.0ms	3.7ms
Main.NewSubFlow.OpSeqLab	1	Fail	5.0ms	3.1ms
Main.NewSubFlow.OpSeqLab	2	Fail	5.0ms	3.1ms
Main.NewSubFlow.OpSeqLab	3	Fail	5.0ms	3.1ms
Main.NewSubFlow.OpSeqLab	4	Fail	5.0ms	3.1ms
Main.NewSubFlow.SpecFileLab	1	Pass	6.3ms	6.0ms
Main.NewSubFlow.SpecFileLab	2	Pass	6.3ms	6.0ms
Main.NewSubFlow.SpecFileLab	3	Pass	6.3ms	6.0ms

Result view with failing tests

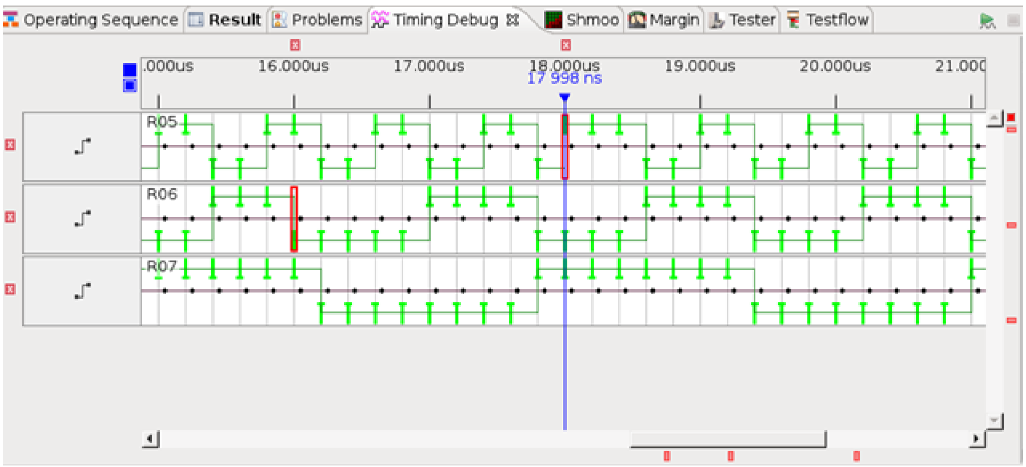
3. In the Result View, Right Click on a failing Test suite and select **Open Testflow Editor (F3)**.
4. Start a debug session (from the Testflow view) for the `SetupFileLab` flow and debug the `OpSeqLab` test suite.

5. Check what signals are failing using **Measurement view**.



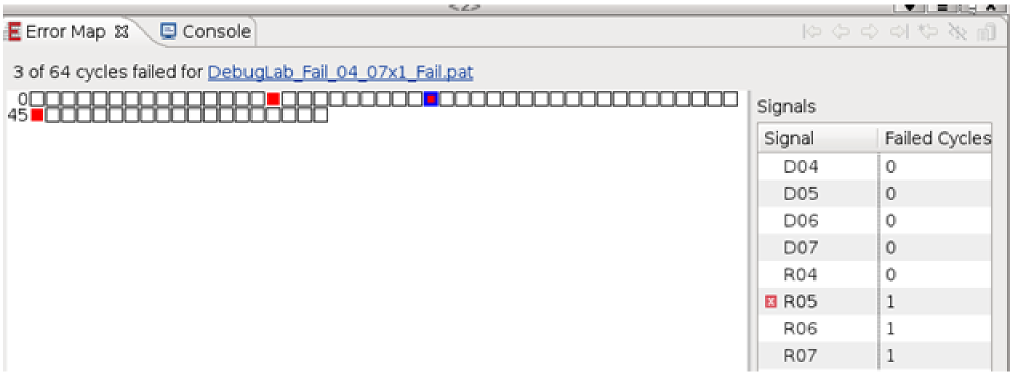
Measurement view showing failing signals

6. Select failing signals and navigate via **Right Click and Show In..**. to the **Timing Debug view**. Check the information (signal name, failing cycle number) shown in the `Property view.`



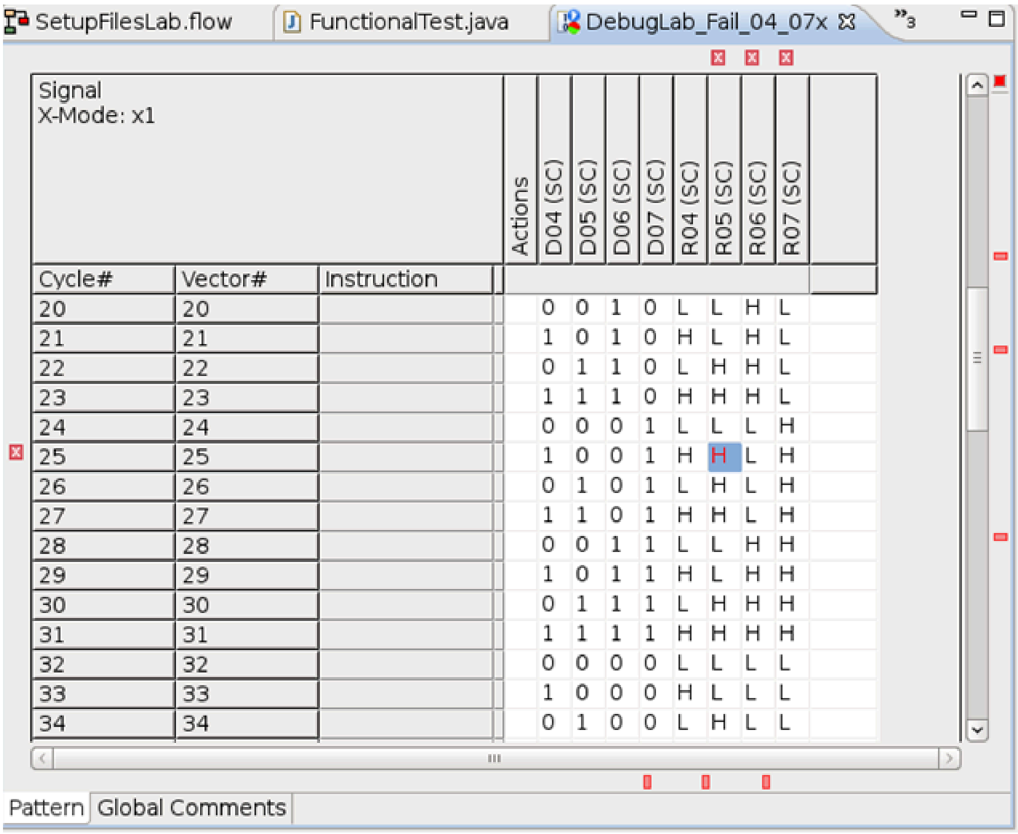
Timing Debug View with fails

7. Use the **Show In...**.feature to navigate from the Timing Debug view to the Error Map. Hover on a failing point and check the **Failed Signals** indication in the property page (right site) of the Error Map.



Error map showing fails

8. Access the **Pattern Editor** from the `Timing debug view`. Right click on the failing point and Select **Open Pattern Editor** or press **F3**.



Navigate from Timing debug to Pattern debug

9. Search failing signal location using the fail annotation surrounding the pattern and fix these (change L>H or H>L as needed). Stay in the debug mode and save the pattern file.

10. Open the `Measurement View` and press to execute the measurement again with the modified pattern.

11. Check if the measurement is still failing using the Debug tools.

12. Terminate the debug execution.

Lab 9 Summary

What you have learnt

- How to use various debug tools to easily analyze the root cause a failing pattern.
- How to fix it and validate the fix while still in the debug mode.

Lab 10: DC Measurement

Learning objective

In this lab you will anchor actions to specific vectors in a pattern. The actions will be executed when the vectors are reached during the execution of the pattern. Furthermore, you will implement operating sequences, that call patterns and DC actions sequentially or in parallel or both.

Getting started

Make sure SmarTest is running and and you are in the SmarTest Work Center.

Task 1: Pattern Based DC Measurement

In this task, you will anchor DC actions within a pattern execution.

Procedure

1. From the test program `BasicLab`, open the testflow `LabMain.flow`
2. Add a testflow `NewDcFlow` calling the test subflow `crossconnect.dcLab.DcLab`

```
flow NewDcFlow calls crossconnect.dcLab.DcLab {
            }
```

3. Add the execute statement to the execute section of the testflow

```
NewDcFlow.execute();
```

4. Open the `DcLab.flow` from the LabMain flow and use the navigation key F3 to open the file `DigInOutDC.spec` that defines the specification of the test suite `DcInPattern`.
5. In the specification file `DigInOutDC.spec`, you will declare and implement two DC actions: `digInOutVfIm and digInOutIfVm`. Insert the actions declaration right after the variable declarations in the specification file as follows:

```
 // add action declarations here
    action digInOutVfIm;
    action digInOutIfVm;
```

6. In the setup of the instrument digInOut associated with the signal D00, specify an action of the type `vforceImeas` and set it up to force a voltage of 1.0 V. Specify the action type and the action name you previously declared.
7. Specify the waitTime property and assign the value 1.5 ms to it.
8. Specify the property irange and assign the value 0.1 uA to it.
9. Specify not to use the high accuracy mode (Board ADC).
10. Use the content assist and the code below as reference.

```
action vforceImeas digInOutVfIm {
            forceValue = 1.0 V;
            waitTime = 1.5 ms;
            irange = 0.1uA;
            highAccuracy = false;
        }
```

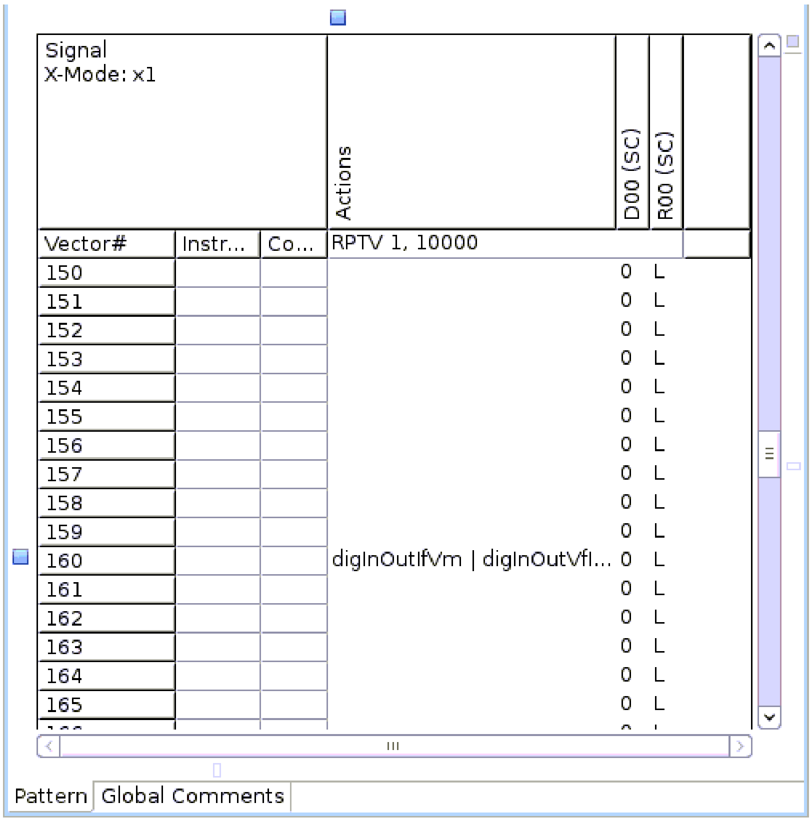
11. Define a DC measure action for the signal `R00`. Use the following steps:
    a) In the setup of the instrument `digInOut` associated with the signal R00, specify an action of the type `iforceVmeas` and define the action properties as given in the following steps.
    b) Use the action name `digInOutIfVm` that you have declared previously.
    c) Specify the properties `limits.high, limits.low, forceValue, waitTime, vclampHigh, vclampLow` and assign the appropriate values to them as shown the code below.
    d) Specify not to use the high accuracy mode (Board ADC).

```
action iforceVmeas digInOutIfVm {
            limits.high = 1.1V;
            limits.low = 0.99V;
            forceValue = 0.01uA;
            waitTime = 1.5ms;
            vclampHigh = 1.5V;
            vclampLow = 0.0V;
            highAccuracy = false;
        }
```

12. Add actions anchors to Pattern

a) From the testflow, use F3 to open the pattern file `DcTrigger.pat`. Note that to open a pattern file in the pattern editor, the test program must be activated.

b) Use the content assist to add the action anchor `digInOutVfIm and digInOutIfVm` at vector 160 and save the pattern. The name of the action must be the same as previously defined in the specification file.



Anchors for Measurement Actions in Pattern

13. Modify the Java test method as shown below. The java files are located in `/src/testMethods/dcLib` Before this, the pattern file should be saved otherwise the anchor will be lost!

a) Rename `Exec1VoltageMeasDigInOut.java` to `DCLab.java`. Right Click the file in the `Package Explorer` and select **Refactor > Rename**. Do not add .java extension.

b) Open `DCLab.java`.

c) Change the default value of signalsMeas (line 20):

```
@In public String signalsMeas = "R00"; // Change default value
```

d) Save modified file.

14. Modify the testflow

a) Open the `DCLab.flow`.

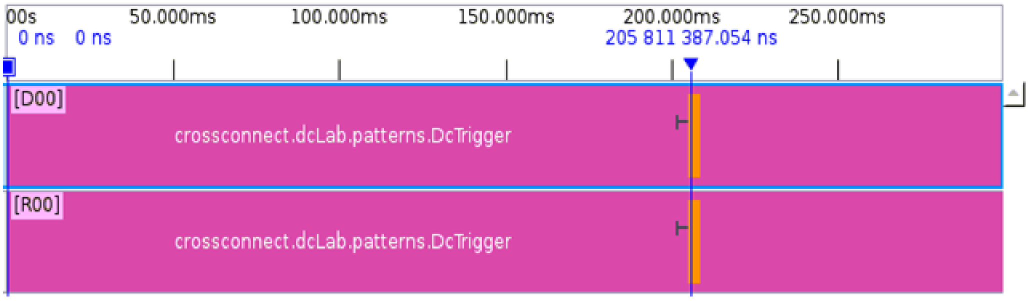
b) Make sure that the test method name to be called in the Test suite is `DCLab` (automatically renamed when the java class file name was renamed at step 6).

c) Execute the testflow `NewDcFlow` from the Testflow View (Right Click on the testflow and Select **Run**).

d) Verify the results in the **Signal** tab of the **Result view**.

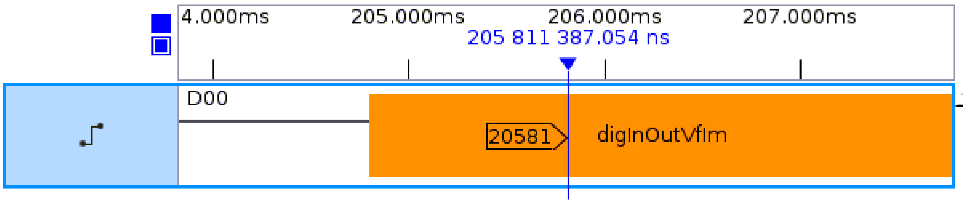
e) From the **Testflow View** or from the **Flow Chart**, start a debug session of the DCLab.flow and execute the DcViLab test suite in debug mode.

f) Open the Operating Sequence View to visualize the representation of the DC measurement within the pattern. Move the cursor to the location of the DC measurement.



DC in Pattern Timing Operating Sequence

g) From the Operating Sequence View, Right Click and select **Show in > Timing Debug** to navigate to the Timing Debug View and visualize the DC measurement again.



DC in Pattern Timing Debug

15. Terminate the debug execution.

Task 2: DC Measurement via Sequencing Group

You will create a DC measurement based on Voltage Force and Voltage Measure between cross-connected pins using an operating sequence with dcVI configured channels.

About this task

You will declare and specify the force voltage and measure voltage actions in a specification file and add the actions to measure the voltage to an operating sequence. The setup of such a measurement involves:

- Specification files: Actions and instruments properties, timing and levels for patterns.
- Pattern files: Instructions, anchors and vector data.
- Operating sequence files: Calls of patterns and actions.
- Java test method: Execution, result and datalog.

Procedure

1. Use the navigation capabilities to access the testflow `DCLab` and the test suite `DcViLab`.
2. Remove the comments for the Test suite DcViLab (in setup and execute section).
3. Define the actions for the signals `D00, D02 and D03` which are associated with the dcVI instrument in the specification file.
   a) Open the spec file `DcViLab.spec`.
   b) Declare the following voltage force and current measurement actions for the signals D00,D02 and D03

```
dcVIVfIm1;
dcVIVfIm2;
dcVIVfIm3;
```

   c) Declare the following current force and voltage measurement actions for the signals R00,R02 and R03

```
dcVIIfVm1;
dcVIIfVm2;
dcVIIfVm3;
```

   d) Locate the setup block and define the actions as shown in the code below:

```
setup dcVI D00 + D02 + D03 {
        // add actions here
        action vforceImeas dcVIVfIm1 {
            forceValue = 1.0V;
            waitTime = 1.5ms;
            irange = 0.1uA;
            highAccuracy = false;
        }
        action vforceImeas dcVIVfIm2 {
            forceValue = 0.9V;
            waitTime = 1.5ms;
            irange = 0.1uA;
            highAccuracy = false;
        }
        action vforceImeas dcVIVfIm3 {
            forceValue = 2.5V;
            waitTime = 1.5ms;
            irange = 0.1uA;
            highAccuracy = false;
        }
    }

    setup dcVI R00 + R02 + R03 {
        // add actions here
        action iforceVmeas dcVIIfVm1 {
            forceValue = 0.01uA;
```

```
            waitTime = 1.5ms;
            vclampHigh = 1.5V;
            vclampLow = 0.0V;
            highAccuracy = false;
        }
        action iforceVmeas dcVIIfVm2 {
            forceValue = 0.01uA;
            waitTime = 1.5ms;
            vclampHigh = 1.5V;
            vclampLow = 0.0V;
            highAccuracy = false;
        }
        action iforceVmeas dcVIIfVm3 {
            forceValue = 0.01uA;
            waitTime = 1.5ms;
            vclampHigh = 2.9V;
            vclampLow = 0.0V;
            highAccuracy = false;
        }
    }
```

4. Add the actions to the operating sequence

a) Modify `DcViLab.seq` to incorporate all IFVM actions sequentially as shown below.

```
sequence DcViLab {
    sequential seqGrp1 {
        actionCall dcVIIfVm1;
        actionCall dcVIIfVm2;
        actionCall dcVIIfVm3;
    }
}
```

b) Verify that DcVILab.flow references the correct files

5. Execute the Test Program from the testflow view and verify the results in the `Result view`.

Task 3: Sequencing DC Actions and Patterns Calls

In this task, you will setup timing and levels for patterns and combine DC measurements and pattern execution based on cross-connected signals using an operating sequence.

Procedure

1. Modify the specification file DcViLab.spec to incorporate digInOut signals for the signals D01 and R01 as shown in the code below. digInOut signals need timing and level definition in the specification file.

```
    setup digInOut D01+R01 {
        wavetable default {
            xModes = 1;
            0 : d1 : 0;
            1 : d1 : 1;
            Z : d1 : Z;
            L : d1 : Z r1 : L;
            H : d1 : Z r1 : H;
            X : d1 : Z r1 : X;
        }
        set timing timingSet_1 { // Timing Set
            period = per;
            d1 = t_drive;
            r1 = per / 2;
        }
        connect = true; set level levelSet_1 { // Level Set
            vil = vcc / 2 - IV_Swing / 2;
            vih = vcc / 2 + IV_Swing / 2;
            vol = vcc / 2 - OV_Swing / 2;
            voh = vcc / 2 + OV_Swing / 2;
        }
    }
```

2. Modify the operating sequence DcViLab.seq to incorporate interleaved (sequential) pattern executions as shown below.

```
sequence DcViLab {
    sequential seqGrp1 {
        // Add action and pattern calls here
      patternCall crossconnect.dcLab.patterns.DcViLab01;
        actionCall dcVIIfVm1;
      patternCall crossconnect.dcLab.patterns.DcViLab02;
        actionCall dcVIIfVm2;
      patternCall crossconnect.dcLab.patterns.DcViLab03;
        actionCall dcVIIfVm3;
    }
}
```

3. Activate the test program and execute it from the Testflow View. Verify the results in the `Result View`.

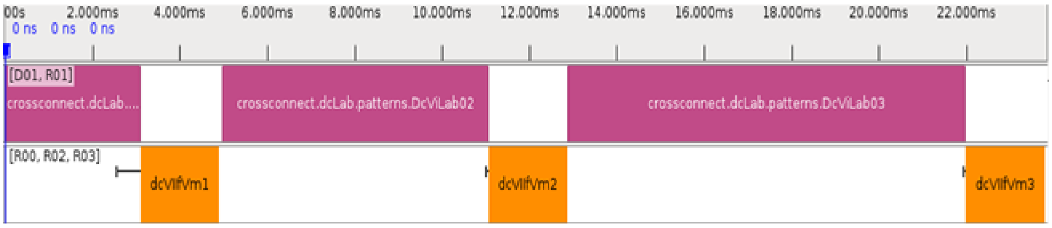
Task 4: Sequential DC Action Calls in the Operating Sequence View

You will execute a test and visualize the representation of the sequences in the `Operating Sequence view`. Make sure you start debugging from the Testflow view or from the Flow Chart.

Procedure

1. Start a debug session for the testflow `DcLab.flow`.
2. Check the `Operating Sequence view` and use the navigation (Show In....) to see also the results in the `Timing Debug view`.

Note	If you do not see the correct operating sequence, right click on it and select Reset Layout.



Operating Sequence with pattern and DC Actions

3. Terminate the debug session.

Task 5: Parallel DC Actions Calls in the Operating Sequence View

You will execute a test and visualize the parallel execution of DC actions and pattern calls `Timing Debug View`.

Procedure

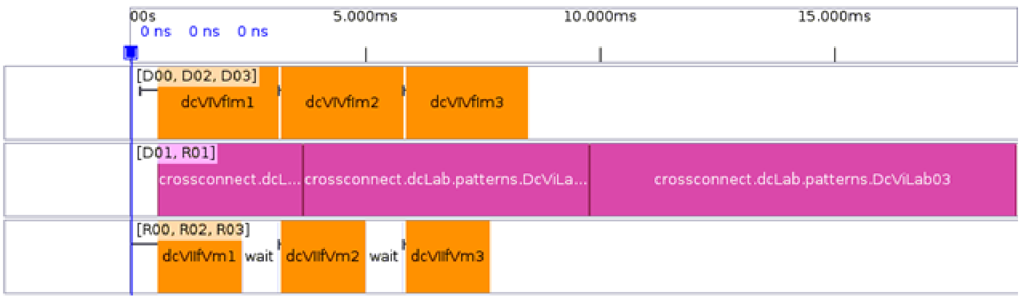
1. Open the flow DcLab.flow and add a test suite of the newly created operating sequence DcViLabParallel (copy the existing one and modify it as shown below)

```
suite DcViLabParallel calls testMethods.dcLib.Exec3VoltageMeasDcVi {
            measurement.specification = setupRef(crossconnect.dcLab.DcViLab);
            measurement.operatingSequence =
 setupRef(crossconnect.dcLab.opSeqs.DcViLabParallel);
            // Parameters passed to method
            signalsMeas = "R00 + R02 + R03";
        }
```

2. Create a parallel group to call the DC actions and patterns in parallel.
3. Add calls of DC actions to sequential group `seqGrp1, seqGrp2` and add calls of pattern in `seqGrp3.`

```
 DcViLabParallel {
    parallel parGrp1 {
        sequential seqGrp1 {
            // Add action calls here
            actionCall dcVIIfVm1;
            wait 850us;
            actionCall dcVIIfVm2;
            wait 850us;
            actionCall dcVIIfVm3;
        }
        sequential seqGrp2 {
            // Add action calls here
            actionCall dcVIVfIm1;
            actionCall dcVIVfIm2;
            actionCall dcVIVfIm3;
        }
        sequential seqGrp3 {
            // Add pattern calls here
            patternCall crossconnect.dcLab.patterns.DcViLab01;
            patternCall crossconnect.dcLab.patterns.DcViLab02;
            patternCall crossconnect.dcLab.patterns.DcViLab03;
        }
    }
}
```

4. Add an `execute()` statement to the execute section of the testflow.
5. Activate the test program and execute the testflow `DcLab.flow` from the Testflow View. Verify the results in the `Result View`.
6. Start a debug session to check the `Operating Sequence View` and verify the signals in the `Timing Debug View`.



Parallel Group in Operating sequence view

7. Terminate the program.

Task 6: Sequential and Parallel Groups in the Timing Debug view

In this task, you will build a more complex operating sequence with sequential and parallel groups.

Procedure

1. Open the flow DCLab.flow.
2. Add a test suite DcViLabComplex for the execution of the newly created operating sequence DcViLabComplex (copy the existing one and modify it).

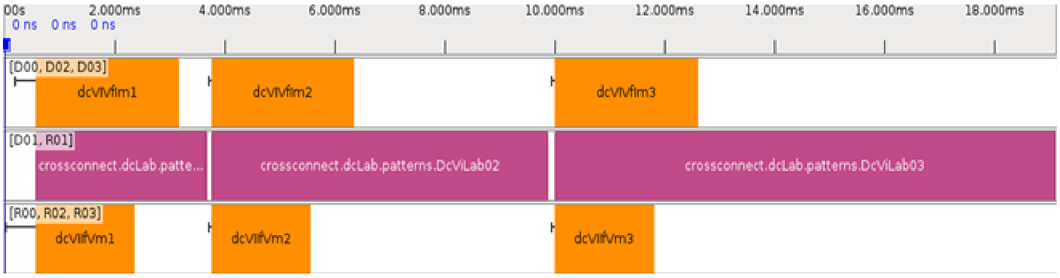
```
suite DcViLabComplex calls testMethods.dcLib.Exec3VoltageMeasDcVi {
          measurement.specification = setupRef(crossconnect.dcLab.DcViLab);
        measurement.operatingSequence =
 setupRef(crossconnect.dcLab.opSeqs.DcViLabComplex);
            signalsMeas = "R00 + R02 + R03";
        }
```

3. Locate and open the operating sequence `DcVILabComplex.seq.`
4. Create 3 parallel groups in a sequential group as shown in the code below.

```
sequence DcViLabComplex {
    sequential seqGrp1 {
        parallel parGrp1 {
            sequential seqGrp11 {
            // Add action call here
                actionCall dcVIIfVm1;
            }
            sequential seqGrp12 {
            // Add action call here
                actionCall dcVIVfIm1;
            }
            sequential seqGrp13 {
            // Add pattern call here
                patternCall crossconnect.dcLab.patterns.DcViLab01;
            }
        }
        parallel parGrp2 {
            sequential seqGrp21 {
            // Add action call here
                actionCall dcVIIfVm2;
            }
            sequential seqGrp22 {
            // Add action call here
                actionCall dcVIVfIm2;
            }
            sequential seqGrp23 {
            // Add pattern call here
                patternCall crossconnect.dcLab.patterns.DcViLab02;
            }
        }
        parallel parGrp3 {
            sequential seqGrp31 {
            // Add action call here
                actionCall dcVIIfVm3;
            }
            sequential seqGrp32 {
            // Add action call here
                actionCall dcVIVfIm3;
            }
            sequential seqGrp33 {
            // Add pattern call here
                patternCall crossconnect.dcLab.patterns.DcViLab03;
            }
        }
    }
}
```

5. Add an `execute()` command to the testflow.

6. Activate the test program and execute the testflow from the **Testflow View**. Verify the results in the `Result View`.
7. Start a debug session and verify the representation in the **Operating Sequence View**.
8. In the **Operating Sequence View**, right click the dcVIVfIm2 action and select **Set Stop Point**.
9. Right Click in the **Operating Sequence View** and select **Re-run to Stop Point**.



Operating sequence with combo pattern and DC Actions

Lab 10 Summary

What you have learnt.

- How to add anchors in pattern that call actions.
- How to implement operating sequences with parallel groups, sequential groups and the combination of both.
- How to set stop point in the operating sequence.

Lab 11: Using the Test Method Library

About this task

Learning objective

Know how to set up test suites that utilize test methods of the SmarTest standard test method library.

Getting started

Ensure SmarTest is running and the lab device is loaded.

Task 1: Add a functional test

In this task, you will add a functional test that uses a test method of an Advantest test method library.

About this task

You can use your own created test methods or the library test methods.

Procedure

1. Open the `LabMain.flow` from the test program file.
2. Open the testflow `SetupFilesLab.flow` called by the LabMain.flow.
3. Add a test suite `ExampleTmlTest`. Instead of completely typing the fully qualified name of the called test method, use content assist (CTRL+space).

```
suite ExampleTmlTest calls com.advantest.itee.tml.actml.FunctionalTest {
            measurement.specification =
 setupRef(crossconnect.setupFilesLab.mainSpecs.ExampleTest);
            measurement.pattern =
 setupRef(crossconnect.setupFilesLab.patterns.Counting_00_15x1);
        }
```

4. Add a corresponding execute statement to the execution section of the testflow.

```
ExampleTmlTest.execute ();
```

5. Save the file.
6. To review the used test method of the library in SmarTest, press F3 while the cursor is at the called test method of `ExampleTmlTest`.
7. Execute the test program and check if it runs the newly inserted test suite runs without failure.

Task 2 : Adding a continuity test

In this task, you will add a continuity test to your test program, that will be executed with every test program run.

Before you begin

Ensure your test program is loaded.

About this task

A continuity test verifies electrical short/open circuits which is useful to check the DUT signal pins for:

- DUT external pin-to-internal circuitry continuity.
- DUT external pin-to-test head channel continuity.
- DUT external pin-to-pin short circuits.

The first test performed in a testflow is typically a continuity test. It senses the presence of the internal ESD protection diodes by performing a current force voltage measure action. Doing this verifies that the DUT signal pins are properly connected to the DUT internal circuitry and to the test head channels.

Procedure

1. In the test program file, define a `PowerUp` auxiliary flow. If the `PowerUp` auxiliary flow is defined, SmarTest executes the `PowerUp` auxiliary flow always before starting the actual test on a DUT. Therefore, it is highly recommended to always explicitly define the PowerUp auxiliary flow.

```
testflow PowerUp {
        flow = crossconnect.auxiliaryFlows.PowerUp;
  }
```

2. Open the called flow with **F3**. The `PowerUp` auxiliary flow calls the flow for continuity test.
3. Insert a new test suite in the `Continuity.flow` that calls the test method Continuity of the test method library in SmarTest with the following properties:
   a) Signals to be tested: `I00 and I01`.
   b) Small current forced: `12 mA`.
   c) Wait time for settling of the test setup: `1 ms`.

```
suite ContinuityTest calls com.advantest.itee.tml.dctml.Continuity {
            dpsSignals = "VDD1 + VDD2";
            specParameters = setupRef(crossconnect.auxiliaryFlows.ContinuityNames);
            signalGroup[IoSignals1] = {
                signals = "I00 + I01";
                forceCurrent = 0.012;
                settlingTime = 1E-3;
            };
        }
```

4. Add an execute statement to the execution section of the testflow such that the new test suite is executed between `PreConditionsContinuity` and `ResetConditionsContinuity`.
5. Remove comments of the `if … else …` statement.
6. Execute the test program and check if it runs the newly inserted test suite properly.
7. Check the measured results of the continuity test. The limits of the test have been defined in the test table which is introduced later.
8. Now assume, that other IO signals should be tested under different conditions. Add a slightly different test setup for the signals `I02` and `I03` which should be tested with a current `13mA`.

```
suite ContinuityTest calls com.advantest.itee.tml.dctml.Continuity {
            dpsSignals = "VDD1 + VDD2";
            specParameters = setupRef(crossconnect.auxiliaryFlows.ContinuityNames);
            signalGroup[IoSignals1] = {
```

```
                signals = "I00 + I01";
                forceCurrent = 0.012;
                settlingTime = 1E-3;
            };
            signalGroup[IoSignals2] = {
                signals = "I02 + I03";
                forceCurrent = 0.013;
                settlingTime = 1E-3;
            };
        }
```

9. Execute the test program and check if it runs the newly inserted test suite properly.

10. Check the measured results of the continuity test

Lab 11 Summary

What you have learnt

- How to create a test suite that calls a test method provided by the SmarTest library.
- How to add and modify an auxiliary flow that executes a continuity test.

Lab 12: Test Method Creation

A test method is needed in each test suite to execute one or multiple measurements.

Learning objective

Create a new test method and make first changes in Java to enhance it slightly.

Getting started

Make sure SmarTest is running and the lab device is loaded.

Task 1: Creating a New Test Method

In this task, you will implement a test method that runs a measurement on the tester hardware. You will then evaluate the result and perform datalogging with an optional print out of the result into the console.

Before you begin

Access the labsCrossConnect device in the Package Explorer of the SmarTest Work Center.

About this task

In this task, you will create a test method from scratch.

Procedure

1. In the **Package Explorer**, right click `src.testMethods.acLib` and select from the pop-up menu **New > Test Method**.
2. In the window **Create a New Test Method** set as name `NewLabTest`.
3. Use the **Add Measurement** button to add an interface to the tester, that allows to configure the tester, to execute tests and to retrieve results from the test head cards. Keep the default name `measurement`.
4. Use the **Add Test Descriptor** button to add an interface for datalogging. Keep the default name **testDescriptor** and type `Functional`.
5. Use the **Add Parameter** button to add an input parameter for your new test method. Name it `printResult` of type `Boolean`.
6. Check the box **Setup and update method stubs** and then press **Finish** to generate a test method that contains placeholders for all essential parts of a test method.
7. Review the generated test method code in `src.testMethods.acLib.NewLabTest.java`.
8. In the execute section, add the command `measurement.execute();` to execute a measurement. Use `Ctrl Space` (Content Assist) whenever possible to enter your code

```
@Override
public void execute() {
   measurement.execute(); // added
```

9. In the following line, add a command to protect the basic results of the measurement, so that these are not overwritten by test results of the next test suites:

```
IMeasurementResult measurementResult = measurement.preserveResult();
```

10. In the following line, add a command to release the tester resources for the next test:

```
    releaseTester();
```

11. In the following line, add a command to log the result of the executed measurement:

```
    testDescriptor.evaluate(measurementResult);
```

12. In the following line, add a line to dump the result to console if the input parameter is true:

```
if(printResult){ println("Test has passed: " + measurementResult.hasPassed());}
```

13. Save the test method file.

Task 2: Adding your Test Method to a Test Suite

In this task, you will add the test method to a test suite within a testflow. You will then execute the testflow and check the results of the execution.

Before you begin

Make sure you are in the SmarTest Work Center and access the method NewLabTest.java.

About this task

A test method is needed in each test suite to execute one or multiple measurements.

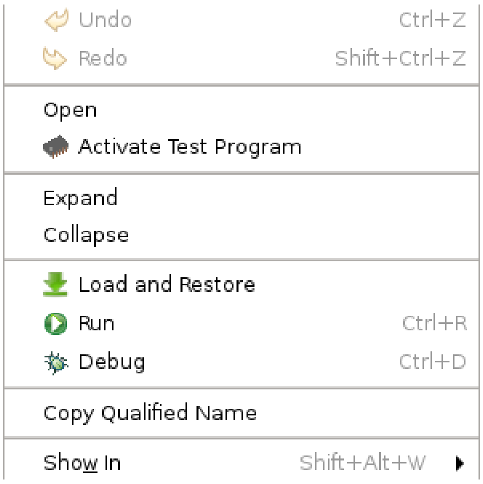
Procedure

1. Open the `LabMain.flow` from the Test Program file.
2. Add a subflow calling statement for LabMain.flow in the `setup{}` section, and add subflow execution statement in the `execute{}` section.

```
flow LabMain {
    setup {
        flow BasicDigitalFlow calls crossconnect.basicDigital.FunctionalTests {
        }
        flow NewSubFlow calls crossconnect.setupFilesLab.SetupFilesLab {
        }
        flow NewDcFlow calls crossconnect.dcLab.DcLab {
        }
        flow NewTestMethodFlow calls crossconnect.testMethodLab.TestMethodLab {
        }
    }
    execute {
        BasicDigitalFlow.execute();
        NewSubFlow.execute();
        NewDCFlow.execute();
        NewTestMethodFlow.execute();
            }
}
```

3. Save the file.
4. Use **F3** to open the testflow `TestMethodLab`.
5. Modify the testflow `TestMethodLab` such that the test suite `DcViMeasurements` calls your new test method.
6. In the setup part of the testflow, set the input parameter `printResult` for the test suite: `printResult = true;`
7. Run the test program. Check the **Console window** for the output `Test has passed!`.

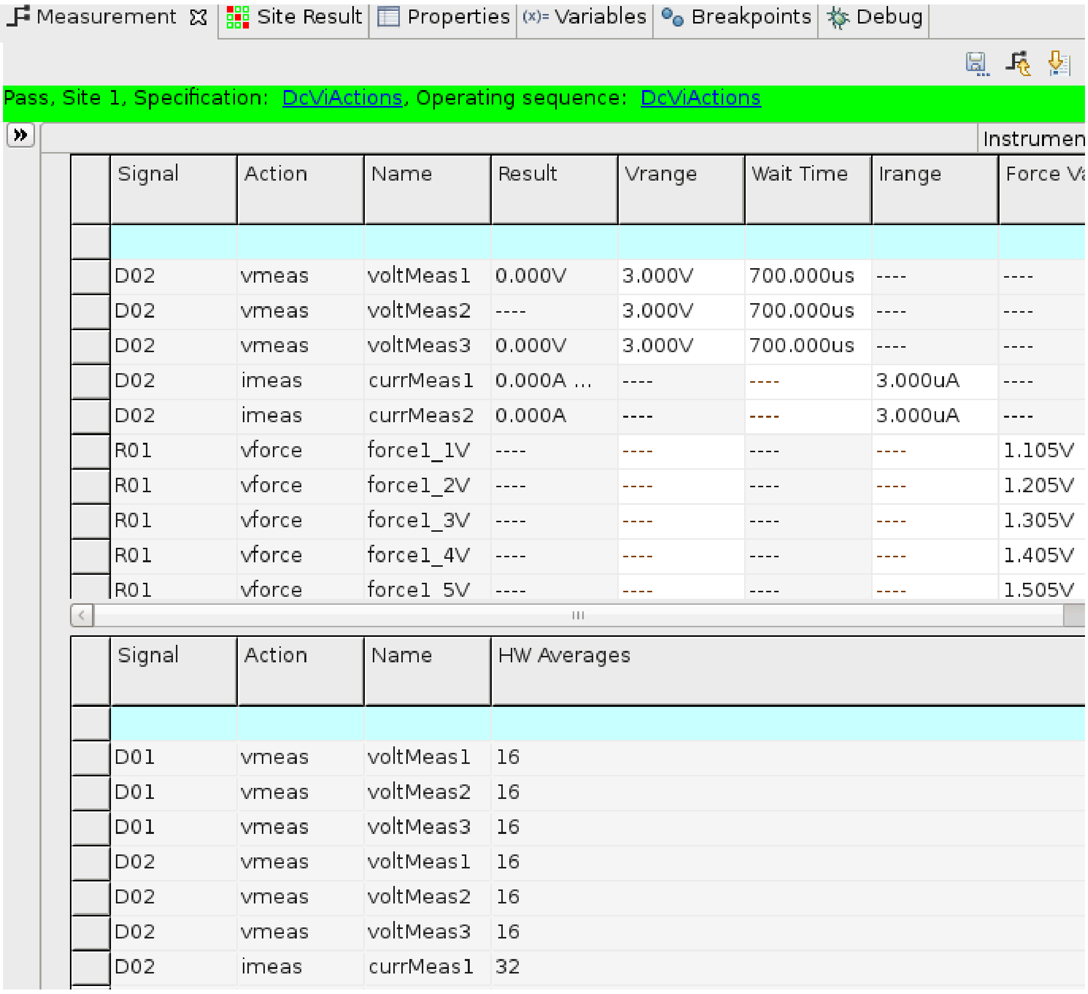
8. From the **Testflow View** start the test program in `Debug` mode.



Open a Debug Session

9. Use **Expand** to see all testflows and test suites.
10. In debug mode, right-click on the test suite `DcViMeasurements` of the newly added testflow `NewTestMethodFlow` and select `Execute`.
11. Compare the setup in the operating sequence file of the test suite with its diagram shown in the **Operating Sequence View**.
12. If online, check the measured results of the various DC measurements of the operating sequence. To do this, select in the **Measurement View** at the bottom the tab for the instrument **dcVI** and at the top the buttom for **Actions**. Then expand the results window.

Results



Signal	Action	Name	Result	Vrange	Wait Time	Irange	Force Va
D02	vmeas	voltMeas1	0.000V	3.000V	700.000us	----	----
D02	vmeas	voltMeas2	----	3.000V	700.000us	----	----
D02	vmeas	voltMeas3	0.000V	3.000V	700.000us	----	----
D02	imeas	currMeas1	0.000A ...	----	----	3.000uA	----
D02	imeas	currMeas2	0.000A	----	----	3.000uA	----
R01	vforce	force1_1V	----	----	----	----	1.105V
R01	vforce	force1_2V	----	----	----	----	1.205V
R01	vforce	force1_3V	----	----	----	----	1.305V
R01	vforce	force1_4V	----	----	----	----	1.405V
R01	vforce	force1_5V	----	----	----	----	1.505V

Signal	Action	Name	HW Averages
D01	vmeas	voltMeas1	16
D01	vmeas	voltMeas2	16
D01	vmeas	voltMeas3	16
D02	vmeas	voltMeas1	16
D02	vmeas	voltMeas2	16
D02	vmeas	voltMeas3	16
D02	imeas	currMeas1	32

Measurement and Operating Sequence view of TM result

Lab 12 Summary

What you have learnt

- How to create a new test method.
- The essential parts of a typical test method.
- How to modify a test suite so that it can call a certain test method.
- How to start a debug mode and execute a single test suite.

Lab 13: Modifying Test Setups in Test Methods

You can use your Java code to modify your test setups.

Learning objective

Read and modify test setups in a test method using the Instrument API.

Getting started

Make sure SmarTest is running and the lab device is loaded and the Lab "Test Method Creation" has been completed.

Task 1: Overwrite Test Setup in Test Methods

In this task, you will add an additional input parameter to the test method to overwrite the test setup and check the results of the execution of the modified test methods.

Before you begin

Access the **SmarTest Work Center** and terminate the previous debug session.

About this task

Parameters of the specification files can be modified in the test method.

Procedure

1. Open the new test method `NewLabTest`. To find it, start from the Test Program file `BasicLab.prog` to the main flow `LabMain.flow`, to the testflow `TestMethodLab`, to the test suite `DcViMeasurements` that calls the test method `NewLabTest`.
2. In the test method, add below the existing input parameter another input parameter `newForceValue` of type `double`.

```
public Boolean printResult;
public double newForceValue; // newly added
```

3. Check again the specification file of the corresponding test suite: The signal `R01` is setup as a `dcVI` instrument that executes multiple actions that force a voltage. One action is named `force1_5V` and forces ~1.5V.
4. Now add in the update part of the test method a line that uses the Instrument API to force a new voltage value for this action `force1_5V` and the new value is set by the new input parameter and will overwrite the settings in the specification file.

```
public void update() {
   measurement.dcVI("R01").vforce("force1_5V").setForceValue(newForceValue); //
 newly added
}
```

5. Match the keywords of the syntax used in the spec file and of Instrument API call in the test method.
6. In the testflow `TestMethodLab`, duplicate the setup and the execution of test suite `DcViMeasurements` and rename the duplicate: `DcViMeasurements3V`.
7. In the setup part of the test suite `DcViMeasurements3V` set the new input parameter to 3 Volts:

```
suite DcViMeasurements3V calls testMethods.acLib.NewLabTest {
   measurement.operatingSequence =
 setupRef(crossconnect.testMethodLab.DcViActions);
   measurement.specification = setupRef(crossconnect.testMethodLab.DcViActions);
   printResult = true;
   newForceValue = 3.0; // set new input parameter of the test method
}
```

8. Use debug mode and use the **Measurement View** to check, which values are forced for the action `force1_5V` for the test suites `DcViMeasurements` and `DcViMeasurements3V`.

Note	The setup of two different test suites is completely separated in the tester (only exception: patterns), so the new value for the voltage force of `DcViMeasurements3V` does not affect the test suite.

Note	In Java the variable type `double` has the default value 0.

9. Now use a different variable type for the input parameter of the test method: `Double`. Change the source code accordingly. Ensure you really write `Double` (not double).

10. In Java the variable type `Double` has the default value `null`. What happens if you execute the test suites? To handle this change the `update` part of the test method needs to be evaluated again.

```
public void update() {
   if (newForceValue != null) {
      measurement.dcVI("R01").vforce("force1_5V").setForceValue(newForceValue);
   }
}
```

11. Use debug mode and use the **Measurement View** to check, which values are forced for the action `force1_5V` for the test suites DcViMeasurements and DcViMeasurements3V. Compare to the behavior of step 8.
12. Set a default value for the input parameter newForceValue:

```
public Double newForceValue = 2.0;
```

13. Compare the result in the **Measurement View**

Task 2: Set Site Specific Values in Test Setups

In this task, you will overwrite just for one site the test setup. Check the results of the execution of the modified test methods. Write out a message, that the setup has been changed.

Before you begin

Access your project in the SmarTest Work Center.

About this task

Setting site specific values.

Procedure

1. Add in the update part of the test method NewLabTest a line that uses the Instrument API to force a new voltage value for the action `force1_5V` only for site 2. The new value is set by the new input parameter and will overwrite the settings in the specification file.

```
  // measurement.dcVI("R01").vforce("force1_5V").setForceValue(newForceValue); //
 commented out, all sites
  measurement.dcVI("R01").vforce("force1_5V").setForceValue(2, newForceValue); //
 newly added, site 2 only
```

2. Use debug mode and use the **Measurement View** to check, which values are forced for the action `force1_5V` for the test suite `DcViMeasurements3V`. To see different results for different sites, open the **Site Result View**. Click on the site for that you want to see the values in the **Measurement View**.
3. Use the built-in `message` function to write out to the Console view a notification, that the setup has been changed. The setup part and the update part of the test method should look like this:

```
  messageLogLevel = 8; // newly added
```

```
public void update() {
  // measurement.dcVI("R01").vforce("force1_5V").setForceValue(newForceValue); //
 commented out, all sites
  if (newForceValue != null) {
measurement.dcVI("R01").vforce("force1_5V").setForceValue(2, newForceValue); //
 site 2 only
  message(4, "Updated forced value for action 'force1_5V' to value " +
 newForceValue + "V!"); // newly added
 }
}
```

4. Execute the testflow `TestMethodLab` and check the output of the **Console view.**
5. Use debug mode and use the **Measurement View** to check, which values are forced for the action `force1_5V` for the test suite `DcViMeasurements3V`.
6. `messageLogLevel` is a variable of every test method, that allows to individually set the level of detail of information that the test method dumps to the **Console View**. Play around with the setting of this variable. What happens if the value is <4 or >=10?
7. Add a message in the execute part of the test method:

```
public void execute() {
   message(6, "No update in the execute part - it would cost test time for each
 execution!"); // newly added
```

8. Use a `Run Configuration` to execute the test suites multiple times or modify the execute part of the testflow `TestMethodLab` as shown below.

```
for (counter : 1 .. 4) {
       DcViMeasurements.execute();
       DcViMeasurements3V.execute();
```

```
    }
```

9. Execute the testflow with different values for `messageLogLevel`: 0, 4, 6, 10 and check the output in the **Console View.**
10. If applicable, comment out the loop command in the execute part of the testflow file `TestMethodLab`.

Task 3: Read and Modify Test Setups

In this task, you will read values from the test setup, modify these and write them back. Then you can check the results of the execution of the modified test methods.

Before you begin

Access your test method projects in the SmarTest Work Center.

About this task

Access and modify test method parameters.

Procedure

1. In the test method NewLabTest, add below the existing input parameters another input parameter `addForceValue` of type `Double`.

```
public Boolean printResult;
public Double newForceValue = 2.0; // recently added
public Double addForceValue; // newly added
```

2. In the update part of the test method add lines that uses the Instrument API to read out the forced voltage value for the `action force1_4V` as it is set in the specification file.

```
   measurement.dcVI("R01").vforce("force1_4V").getForceValue();
```

3. The value must be stored in a variable. Select the name `oldValue`.

```
   oldValue = measurement.dcVI("R01").vforce("force1_4V").getForceValue();
```

4. This causes an error because the type of `oldValue` is not defined. To fix it, hover the mouse over `oldValue` or put the cursor at its name and press CTRL-1. So-called `quick fixes` are proposed, select to create a local variable, so the the result is:

```
   MultiSiteDouble oldValue =
 measurement.dcVI("R01").vforce("force1_4V").getForceValue();
```

5. Add the following lines that print out `oldValue` and that exactly writes back its values:

```
   MultiSiteDouble oldValue =
 measurement.dcVI("R01").vforce("force1_4V").getForceValue();
   message(4, "Value of 'oldValue': " + oldValue);
   measurement.dcVI("R01").vforce("force1_4V").setForceValue(oldValue);
```

6. Run the test suite and check how the values of `oldValue` are shown in the **Console View**.
7. Now the value of `addForceValue` should be added to the old values and then written to the setup. In order to accomplish that, put a "." behind `oldValue` in the third line and press `CTRL-SPACE`. SmarTest offers a list of available functions for the variable type of `oldValue`, which is `MultiSiteDouble`. Select `add(MultiSiteDouble aa)` and then modify the test method code as shown below:

```
if (addForceValue != null) {
   MultiSiteDouble oldValue =
 measurement.dcVI("R01").vforce("force1_4V").getForceValue();
   message(4, "Value of 'oldValue': " + oldValue);

 measurement.dcVI("R01").vforce("force1_4V").setForceValue(oldValue.add(addForceValue));
}
```

Note	Java does not allow to overload operators like "+", "-", "*", "/" etc. Therefore, functions like "add", "subtract", "multiply", "divide" etc exist.

8. In the testflow `TestMethodLab`, in the setup part of the test suite `DcViMeasurements3V` set the new input parameter *addForceValue* to 1 Volt.
9. Execute the test suite `DcViMeasurements3V` and check the forced values for the action `force1_4V`.
10. Stop Debug mode. Start Debug mode again. Check that always 1 V is added after re-executing this test.
11. Use **Load and Restore** for the testflow TestMethodLab to initialize the voltage with the value given in the specification file. Verify the voltage value. Note that **Load and Restore** are accessible only from the Setup Perspective.
12. In the update part of the test method, comment out the line that sets the forced value for the action *force_1_4V* at signal `R01` as preparation for the following labs.

Lab 13 Summary

What you have learnt

- How to add (input) parameters.
- How to check if a value was assigned to an input parameter.
- How to define a default value for an input parameter.
- How to read and change instrument setups in a test method.
- How to provide site specific values for instrument setups in SmarTest 8.
- How to use "message" to write out to the Console view from the test method.
- How to call a test suite multiple times from a testflow using a loop. Differences between primitive data types (like "double") and other data types (like "Double") in Java.

Lab 14: Retrieve Test Results

Retrieving test results involves executing measurements, accessing result accessors and test descriptors.

Learning objective

Accessing the results of the DC measurements to judge, if the test passed or failed. Put the results into data log.

Getting started

Make sure SmarTest is running, the lab device is loaded and the Lab Test Setup Modifications has been completed.

Task 1: Retrieve Results (ONLINE)

In this task, you will retrieve the current and voltage values that are measured by corresponding actions.

Before you begin

Access your test method from the previous lab.

About this task

Usage of test descriptors.

Procedure

1. Open your test method `NewLabTest.java` - starting from the Test Program file to the main flow, to the testflow `TestMethodLab`, to the test suite `DcViMeasurements` that calls the test method.
2. Results are available as soon as the measurement has been executed, i.e. the tester has physically performed a measurement. To generate a handle for the results measured at signals *D01* and *D02*, add the following line in the test method - after the line with the command to protect the basic results of the measurement:

```
measuredResultsHandle = measurement.dcVI("D01+D02").preserveResults();
```

3. Use a `quick fix` as described in Lab 13 Task 3 to find the correct type for the local variable *measuredResultsHandle*, which is a handle that provides access to the measured values. Furthermore, this command protects the memory storing the results of the signals `D01` and `D02`, so that the memory content is not overwritten by the results of the next measurements. Now the next test suite can start to use the tester resources, therefore the next line is `releaseTester();` which is already in the test method.
4. The following command uploads all measured voltages of the *vmeas* actions from the test head cards to the workstation. Insert it after `releaseTester();`

```
Map<String, MultiSiteDoubleArray> measuredVoltages =
 measuredResultsHandle.vmeas("").getVoltage();
```

5. *measuredVoltages* is a map, for each signal for that a `vmeas` action was performed, it stores the result. The signal names (of type "String") are the `keys` of the map that "open the door" to access the measured voltages. Voltage values are stored as type `Double` for multiple `vmeas` actions at multiple sites. Therefore, the voltages are stored in the data type `MultiSiteDoubleArray`.
6. Add the following line to see the content of the variable *measuredVoltages* and:

```
Map<String, MultiSiteDoubleArray> measuredVoltages =
 measuredResultsHandle.vmeas("").getVoltage();
message(4, "Measured voltages: " + measuredVoltages); // dump measured voltages to
 console
```

7. Compare the values dumped to the **Console View** with the values shown in the **Measurement View**.
8. Now add two more lines to upload measured current values and to dump these to the **Console View**:

```
Map<String, MultiSiteDoubleArray> measuredVoltages =
 measuredResultsHandle.vmeas("").getVoltage();
message(4, "Measured voltages: " + measuredVoltages); // dump measured voltages to
 console
Map<String, MultiSiteDoubleArray> measuredCurrents =
 measuredResultsHandle.imeas("").getCurrent();
message(4, "Measured currents: " + measuredCurrents); // dump measured currents to
 console
```

Task 2: Result Accessors (ONLINE/Optional)

In this task, you will learn more ways to process the voltage values that are measured by corresponding actions.

Before you begin

Access your test method from the previous lab.

About this task

Accessing measurement results.

Procedure

1. In the following continue to add lines to the execute part of the test method after the line:

```
releaseTester();
```

2. Check that only the measured voltages of actions *voltMeas1* are dumped:

```
Map<String, MultiSiteDoubleArray> measuredVoltagesMeas1 =
                          measuredResultsHandle.vmeas("voltMeas1").getVoltage();
message(4, "Measured voltages of action 'voltMeas1': " + measuredVoltagesMeas1);
```

3. Check that only the measured voltages at signal "D01" are dumped:

```
MultiSiteDoubleArray measuredVoltagesD01 =
 measuredResultsHandle.vmeas("").getVoltage("D01");
message(4, "Measured voltages at signal D01: " + measuredVoltagesD01);
```

4. Check that only the measured voltages at signal "D01" of actions *voltMeas1* are dumped:

```
MultiSiteDoubleArray measuredVoltagesMeas1D01 =
 measuredResultsHandle.vmeas("voltMeas1").getVoltage("D01");
message(4, "Measured voltages of action 'voltMeas1' at signal D01: " +
 measuredVoltagesMeas1D01);
```

5. Check that only the measured voltages at site 2 and at signal "D01" of actions *voltMeas1* are dumped:

```
double[] measuredVoltagesMeas1D01Site2 = measuredVoltagesMeas1D01.get(2);
// arrays of type double[] are not implicitly converted to a string -> no simple
 print out
message(4, "Measured voltages of action 'voltMeas1' at signal D01 at site 2: ");
for (double value : measuredVoltagesMeas1D01Site2) {
    message(4, "Value:" + value);
 // loop over all values in measuredVoltagesMeas1D01Site2
 // variables of type "double" are implicitly converted to a string
}
```

6. Check that only the measured voltages at signal "D01" of the first action *voltMeas1* are dumped:

```
MultiSiteDouble measuredVoltagesFirstMeas1D01 =
 measuredVoltagesMeas1D01.getElement(0);
message(4, "Measured voltages of the first action 'voltMeas1' at signal D01: " +
                                              measuredVoltagesFirstMeas1D01);
```

7. Check that only the measured voltage at site 2 and at signal "D01" of the first action *voltMeas1* is dumped:

```
double measuredVoltagesFirstMeas1D01Site2 = measuredVoltagesMeas1D01.getElement(2,
 0);
message(4, "Measured voltage of the first action 'voltMeas1' at signal D01 at site
 2: " +
```

```
 measuredVoltagesFirstMeas1D01Site2);
```

8. Often calculations and other post processing of the results must be performed. To do this, the data stored in the result map (for example in 'measuredVoltages') must be accessed. In order to see, how this is done, add the following lines to dump all data to the console:

```
/* loop over all entries in the map 'measuredVoltages' */
for (Entry<String, MultiSiteDoubleArray> entry : measuredVoltages.entrySet()) {

   /* get signal name, i.e. key of map entry */
   String signal = entry.getKey();

   /* print the signal name */
   message(6, "Signal \"" + signal + "\"");

   /* retrieve the measured voltages of all actions on all sites for the specified
 signal */
   MultiSiteDoubleArray measuredVoltagesOfSignal = entry.getValue();

   /* loop over the number of actions */
   for (int actionCount = 0; actionCount < measuredVoltagesOfSignal.length();
 actionCount++) {

      /* print the action number */
      message(6, "   \"vmeas\" Action number \""+ (actionCount + 1) + "\"");

      /* retrieve the measured voltages of the specified action */
      MultiSiteDouble measuredVoltagesOfAction =
 measuredVoltagesOfSignal.getElement(actionCount);

      /* loop over all active sites */
      for (int site : context.getActiveSites()) {

         /* assemble and print a formatted message */
         String output = String.format("      Site \"%d\" measured %3.6fV.", site,

 measuredVoltagesOfAction.get(site));
         message(6, output);
      }
   }
}
```

9. Modify the code above so that, at the end the test method dumps the average value of voltages per signal, if `messageLogLevel` is 8 or higher.

Task 3: Pass/Fail Decision and Result View/Optional

In this task, you will use test descriptors to let SmarTest make a pass/fail decision. Then the result is automatically written to the datalog.

Before you begin

Access your lab retrieving test results.

About this task

Usage of test descriptors to send data to datalog.

Procedure

1. SmarTest uses for pass/fail decisions on test results objects of type `test descriptor`. Three different types of test descriptors are available: functional `IFunctionalTestDescriptor`, parametric `IParametricTestDescriptor` and scan `IScanTestDescriptor`. So far the functional test descriptor has been used. After executing the testflow `TestMethodLab`, check for its test suites, what pass/fail results have been datalogged in the Result View. Use the tab **Test**.
2. In the following, a test method should judge the measured voltages. As a first step, access the **Package Explorer** and use right click to copy the test method file `NewLabTest.java`.
3. Then use mouse right click and Paste a copy of the test method file. Select for the copy the name `NewLabTestParametric`.
4. In the setup part of the testflow `TestMethodLab` make sure that the test suite *DcViMeasurements3V* uses now the new test method `NewLabTestParametric`:

```
suite DcViMeasurements3V calls testMethods.acLib.NewLabTestParametric {
   measurement.operatingSequence =
 setupRef(crossconnect.testMethodLab.DcViActions);
   measurement.specification = setupRef(crossconnect.testMethodLab.DcViActions);
   printResult = true;
   newForceValue = 3.0;
   addForceValue = 1.0;
}
```

5. In the new test method *NewLabTestParametric*, the functional test descriptor is no longer needed. Instead, a parametric test descriptor will be used to judge the measured voltages. Modify the corresponding part of the new test method *NewLabTestParametric* as follows:

```
    public IMeasurement measurement;
    public IParametricTestDescriptor testDescriptor; // newly added
    //    public IFunctionalTestDescriptor testDescriptor;
```

6. The editor shows an error after modifying the type of the test descriptor. The reason is: While functional test descriptors work with Boolean values, parametric test descriptors expect for pass/fail decisions parametric values and not just a simple Boolean value. Comment out the corresponding line:

```
// testDescriptor.evaluate(measurementResult);
```

7. Now let SmarTest make a pass/fail decision for the measured voltage values at signal "D01" of the first action *voltMeas1*: Whether or not the values are in the limits given in the testflow. For that, add the following two lines after the line that has been commented out:

```
// testDescriptor.evaluate(measurementResult);
testDescriptor.setTestText("Check voltages of the first action 'voltMeas1' at
 signal D01");
```

```
testDescriptor.evaluate(measuredVoltagesFirstMeas1D01);
```

In the test table, which is not subject of this lab task, limits have been chosen for the parametric test descriptor in a way, such that a fail is produced if the voltage values are below 0.9V (for example, if the test is run offline) or if they exceed 3.1V, which is more than the maximum voltage forced.

8. After executing the testflow `TestMethodLab`, check again for its test suites, what pass/fail results have been datalogged in the Result View. Use the tab **Test**.
9. Now let SmarTest make a pass/fail decision for the measured voltage values at signal "D01" of the second execution of the action *voltMeas1* by selecting these results out of the results of all actions *voltMeas1* at signal "D01". For that, add the following two lines:

```
testDescriptor.setTestText("Check voltages of the second action 'voltMeas1' at
 signal D01");
testDescriptor.evaluate(measuredVoltagesMeas1D01, 1);
```

Note	The multiple executions of the same action are numbered starting with "0". Therefore, to access the results of the second action, the number given for the `evaluate` function must be "1".

10. After executing the testflow `TestMethodLab`, check again for its test suites, what pass/fail results have been datalogged in the **Result View**. Use the tab **Test**.
11. Also check the tab **Signal** of the **Result View**. There are no results from the test suite `DcVIMeasurements3V`, because in the test method code so far no call of `testDescriptor.evaluate(<data>)` was used with data that contained names of signals: The data was either of type `MultiSiteDouble` for `measuredVoltagesFirstMeas1D01` or `MultiSiteDoubleArray` for `measuredVoltagesMeas1D01`.
12. Now let SmarTest make a pass/fail decision for the measured voltage values at all signals that execute the action *voltMeas1*. In order to do that, data of type `Map<String, MultiSiteDoubleArray>` will be used, which contains signal names. Furthermore, the values of the first execution of the action *voltMeas1* are selected. In order to do so, add the following two lines:

```
testDescriptor.setTestText("Check voltages of the first action 'voltMeas1'");
testDescriptor.evaluate(measuredVoltagesMeas1, 0);
```

13. After executing the testflow `TestMethodLab`, check again for its test suites, what pass/fail results have been datalogged in the **Result View**. Note that in the tab Test no value is shown ("N/A") because this test has two results: one for D1 and one for D2. Right-click in the test tab, select **Preferences** and in **Visible colums** check the box **Mean Pin Value** to see the average value of the two signals.

Lab 14 Summary

What you have learnt

- How to protect the memory in test head cards, so that results stored in there are not overwritten.
- How to upload test results from the test head channels to the workstation and your test method.
- How to make sure that you only upload these results you need for a pass/fail decision - in order to minimize communication between test head cards and workstation and reduce run time. Various data types of variables that contain results and how the content of the results relates to the data type.
- How to make a pass/fail decision on parametric values.
- How to datalog parametric results.
- How to use the Result View to check results written to datalog.

Lab 15: Datalogging

You will use Run Configurations to control the datalogging. You will learn how datalog profiles influence the test results.

Task 1: Verify Datalog Setting

Verify settings for datalog in the related testtable.

Before you begin

Make sure SmarTest is running and your test program is imported in the `SmarTest Work Center`.

About this task

Setting formatter and generate datalog in Engineering mode.

Procedure

1. Access the test program file `BasicLab.prog` in the **Package Explorer**.
2. Open the testable, BasicLabTestTable.ods
3. Open the STDF_Config worksheet and verify the stdf Version.

Task 2: Using Run Configurations

You can specify, save and reuse customized configurations for running test programs or testflows.

Before you begin

Make sure SmarTest is running and verify the datalog setting in the testtable.

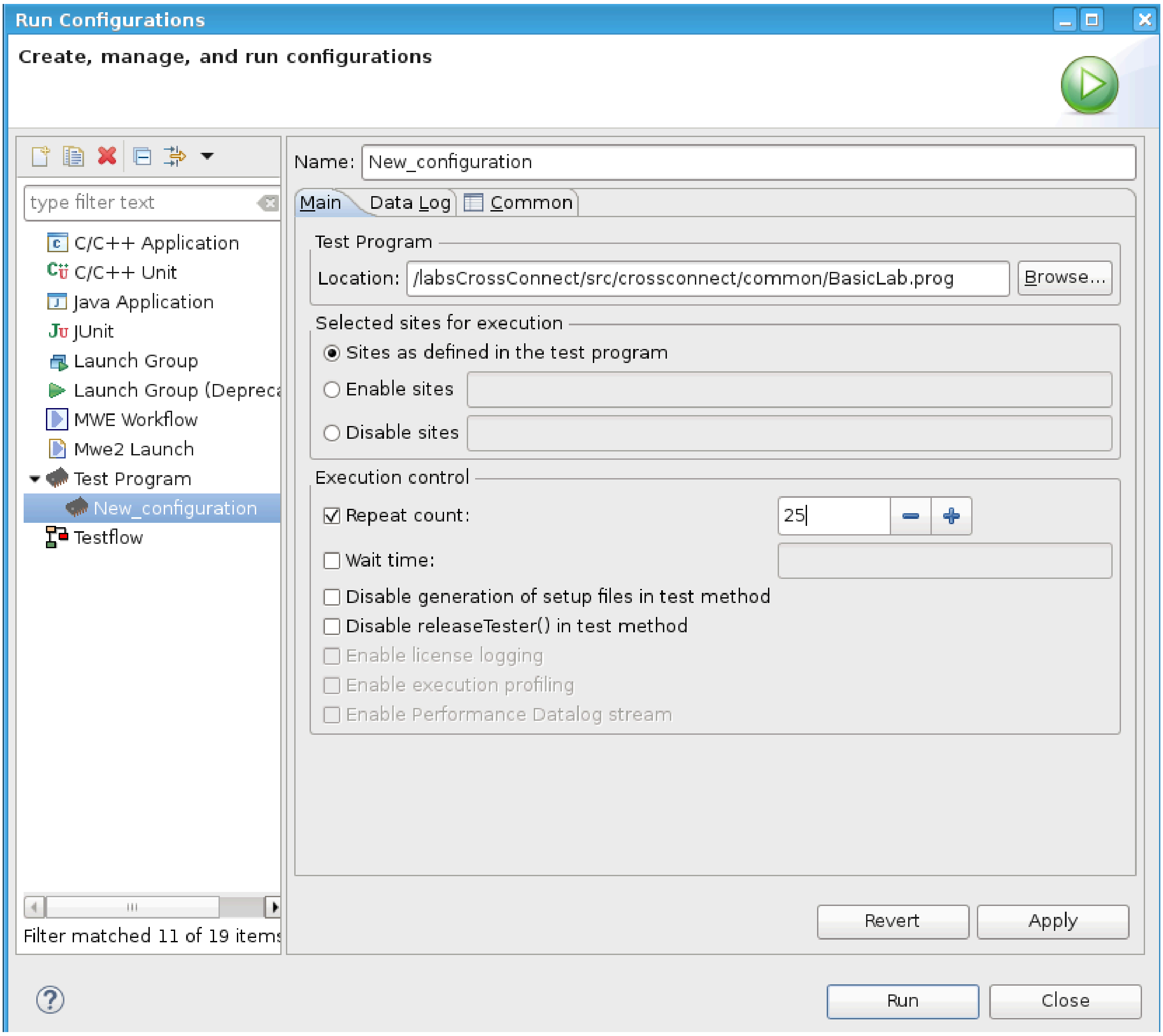
About this task

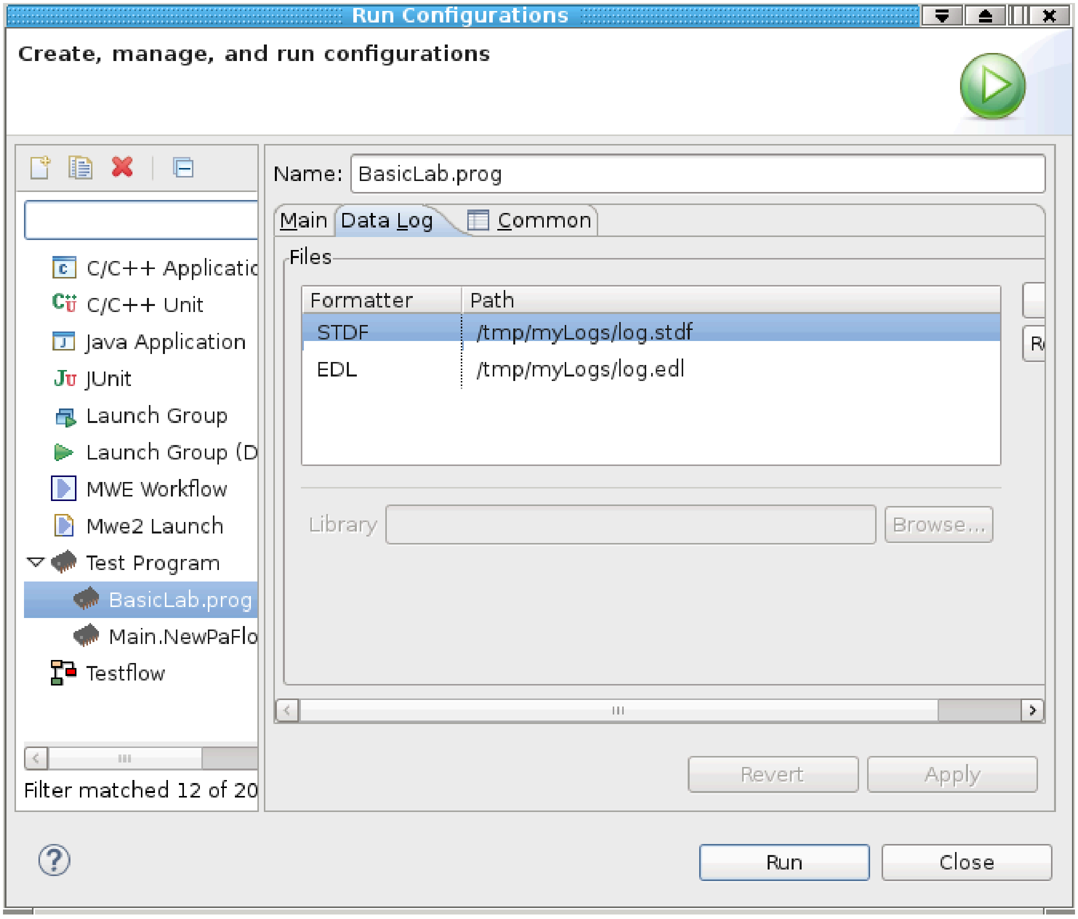
Using `Run configurations` to execute testflows and send results to the datalog.

Procedure

1. In the **Main Tool bar** and select **Run > Run Configurations…**
2. Select the **New** icon and enter a name for the configuration.
3. Use the pictures below as reference for setting up the configuration.
4. Set the Repeat count to 25.
5. Open the **Data Log** tab and use the **Add** button to enter the path and the format for the data log. Make sure the path is available on the workstation (you may have to create the directory).
6. Click **Apply** and **Run** to execute the Test Program according to your settings.
7. Use the pictures below as reference.

8. Go to the `System Console` or a file browser and verify that the EDL and STDF file has been generated.




Run Configurations
Create, manage, and run configurations
C/C++ Applicatio
C/C++ Unit
Java Application
JUnit
Launch Group
Launch Group (D
MWE Workflow
Mwe2 Launch
Test Program
BasicLab.prog
Main.NewPaFlo
Testflow
Filter matched 12 of 20
Name: BasicLab.prog
Main
Data Log
Common
Files
Formatter
Path
STDF
/tmp/myLogs/log.stdf
EDL
/tmp/myLogs/log.edl
Library
Browse...
Revert
Apply
Run
Close

Task 3: Avoid Overwriting of old Datalog Files

Use system variables to make datalog file unique.

Procedure

1. In the Test Program file, introduce a new variable "Name" as follows:

```
var Name = "labCrossconnect";
```

2. In the Run Configurations, set the path for the datalog files as follows (edl or stdf).

```
home/${ENV.USER}/datalog/${TP.Name}_${ENV.TIMESTAMP}.edl
```

```
/home/${ENV.USER}/datalog/${TP.Name}_${ENV.TIMESTAMP}.stdf
```

3. Execute the test program mutiple times and verify that each run generates a unique log file in the subfolder `datalog` of your home directory.

Task 4: Analyzing Datalog Results

Use the Analysis Perspective to visualize datalogs (EDL and STDF).

Before you begin

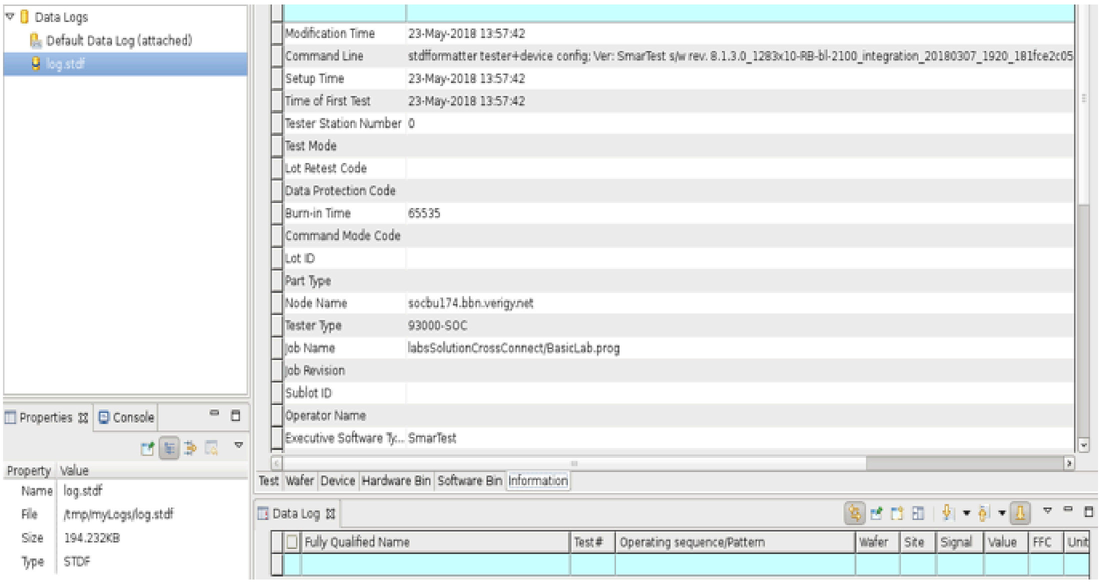
Access to the previously generated datalog files.

About this task

Using the Analysis Perspective, you can open, load and test result data.

Procedure

1. In the **SmarTest Work Center**, select **Window > Perspective > Open Perspective** and select **Analysis**.
2. In the **Datalog Explorer** of the **Analysis Perspective**, right click and select **Load** (browse to select your EDL file).
3. Right click and select **Open** to view the content of the EDL file in the **Analysis View**.
4. Use the different views of the **Analysis Perspective** to verify the content of your EDL file.
5. If you have loaded several data logs in the `Data Log Explorer`, you can use the **Link with Editor** icon to link a datalog with its content.
6. Use the navigation capability of the **Analysis View** to jump to the **Testflow View**, **Result view** etc.
7. Load an `STDF` datalog the same way to see its content.
8. Use the following picture as reference.



Lab 15: Summary

What you learnt

- How to use **Run Configurations** to setup datalogging.
- How to ensure that each datalog file is unique.

Lab 16: Protocol Aware Setup Files

Working with Protocol aware.

Learning objective

Generate setups for tests based on protocol aware software in order to know what files and file contents are needed for protocol aware tests in SmarTest 8.

The lab is based on an artificial protocol that is very simple and easy to understand, so that you need only to focus on the setup for protocol aware tests in SmarTest 8. The protocol definition is based on six signals, the so-called "protocol signal roles":

- "Clk": clock.
- "Data0", "Data1", "Data2" and "Data3": four data signals.
- "Ack": reserved for acknowledgement..

Getting started

Make sure SmarTest is running, the lab device is loaded and Labs 1-15 have been successfully completed.

Task 1: Add a Sequence of Protocol Transactions

In this task, you will define a new protocol transaction sequence that is added to an existing setup based on protocol aware software. Then you will enhance an operating sequence so that it calls the new transaction sequence. Finally you will execute a test suite calling this operating sequence so that the new transaction sequence is part of the execution.

About this task

Creating and executing a test based on a protocol transaction.

Before you begin

Make sure SmarTest is running, the lab device is loaded, and the Labs 1-15 have been sucessfully completed.

Procedure

1. From the test program `BasicLab`, open the main testflow LabMain.flow
2. Add a testflow NewPaFlow calling the test subflow `crossconnect.paLab.ProtocolAwareLab`

   ```
   flow NewPaFlow calls crossconnect.paLab.ProtocolAwareLab {
               }
   ```

3. Add the execute statement to the execute section of the testflow

   ```
   NewPaFlow.execute();
   ```

4. Open the `ProtocolAwareLab.flow` from the `LabMain flow`: Use the navigation key **F3** to open the file `ProtocolAware.spec` that defines the specification of the test suite `PaCounterTest`.
5. Review the lines in the specification file, that define for the DUT the protocol interface "LSB4". The definition includes:
   a) The fully qualified name of the file `Lsb4.prot`, that defines the protocol. This definition lists the six protocol signal roles used in the protocol and describes, how basic protocol transactions (for example, read and write) are translated to digital values to be driven or expected .
   b) The mapping of the six protocol signal roles (given in the protocol definition file) to signal names that are set up for the test program (given in the `DUT board description file`).
6. Review the lines that declare and setup the protocol transaction sequence "ts0to15".
   a) An instance of a transaction sequence definition must be declared with a name.
   b) A transaction sequence, that is instantiated, has to use a certain protocol interface (here LSB4), that defines the protocol used and the signals involved
   c) To instantiate a transaction sequence, a reference to the definition of the transaction sequence must be given. Transaction sequences are defined in transaction sequence definition files. For the example in `ProtocolAware.spec`, the definition of the transaction file is named "Write0to15". "Write0to15" is defined in the file `src/crossconnect/paLab/ReadWrite.trseq`. The path of the file and the name of the transaction sequence definition results in the fully qualified name `crossconnect.paLab.ReadWrite.Write0to15`
   d) To instantiate a transaction sequence, you have to provide a name which is declared.
7. Review at the bottom of the specification file the used wavetable, that defines waveforms that are mandatory: "0", "1", "L", "H", "X", "C". Additionally, the waveform "P" is defined (and used in the protocol definition file Lsb4.prot). It describes a clock pulse.
8. Use the key **F3** at the lines of the instantiation of "ts0to15" to open in the editor the transaction sequence definition file `ReadWrite.trseq`.
9. Define a new transaction sequence: Add the definition of the transaction sequence "Read15to0", that executes the transactions "read(15)", "read(14)",…,"read(1)", "read(0)".

10. Instantiate a transaction sequence of type "Read15to0" in order to use it: Use the yellow arrow pointing to the left to go back to the specification file and edit it. Declare and instantiate after "ts0to15" another transaction sequence "ts15to0" . Use the newly defined "Read15to0" of "ReadWrite.trseq".
11. Call a transaction sequence in the operating sequence which will be executed during test: Go back to the testflow file `ProtocolAwareLab.flow` and open the operating sequence file of test suite `PaCounterTest`. Add to the parallel group `parGroup1` a sequential `exList12` that calls the new transaction sequence "ts15to0".
12. Run the lab test program and check the **Result View**, that the new flow and the new test suite are executed.
13. In debug mode, use the **Operating Sequence View**, to check if the newly added transaction sequence is executed. Unfold in the **Timing Debug View** the special protocol row. When unfolded, it shows the mapping from protocol signal roles to actual signals defined in the **DUT Board description file**.
14. If online, modify the value of one "read()" transaction of the definition of the transaction sequence "Read15to0" to inject a fail. Then run and debug the new test suite. Use the debug tools to find the modified transaction.
15. Undo the modification of "Read15to0".

Task 2: Define a Parametrized Sequence of Protocol Transactions

About this task

In this task, you will add definitions of sequences of protocol transaction that use input parameters to set the data written or expected. Then you will enhance an operating sequence so that it calls one of the new transaction sequences. Finally you will execute a test suite calling this operating sequence.

Procedure

1. Starting from the test program `BasicLab`, navigate through multiple files: Open the main testflow `LabMain.flow`, which calls the testflow `NewPaFlow`. This testflow defines the test suite `PaCounterTest`, that uses the specification file `ProtocolAware.spec`. In the specification file is a reference to the transaction sequence definition file `ReadWrite.trseq`.
2. Define new transaction sequences that are parametrized. Add the definitions of the transaction sequences "WriteOneVector" and "ReadOneVector with the input parameter "data". The value of "data" is written to the DUT or expected to be read from the DUT.

```
transactSeqDef WriteOneVector(UnsignedLong IN data) {
        write(data);
    }

    transactSeqDef ReadOneVector(UnsignedLong IN data) {
        read(data);
    }
```

3. Instantiate a transaction sequence in order to use it: Go back to the specification file `ProtocolAware.spec`. In the specification file a variable "SpecData" of type "UnsignedLong" is defined. The value of this variable serves as parameter of a new transaction sequence instantiating "WriteOneVector":

```
transactSeq writeData;
    setup protocolInterface LSB4 {
        transactSeq crossconnect.paLab.ReadWrite.WriteOneVector writeData{data =
 SpecData; }
    }
```

   The usage of the specification variables for parametrized transaction sequences is not mandatory. One could also implement …{data = 42;}. However, the usage of the specification variables for parametrized transaction sequences has benefits, because during test program execution, a test method can modify the value of a specification variable. For example, if the test suite PaCounterTest would call a test method that changes the site common value or some site specific values of "SpecData" at run time, then automatically the transaction sequence will be translated and downloaded before executing the test for the next time. Therefore, the next execution of the transaction sequence "writeData" is correctly based on the changed values.
4. Go back to the testflow file `ProtocolAwareLab.flow` and open the operating sequence file of test suite `PaCounterTest` to add at the end the parallel group "parGroup2" calling "writeData".
5. Run the lab test program and use in debug mode the **Operating Sequence View** to check, if the newly added transaction sequence is executed.
6. Change the four least significant bits of the specification variable "SpecData" and after running again, check in the timing debug view, if the corresponding data in the executed transaction sequence has changed as well.

Task 3: Enhance the Definition of a Protocol

Understand how to work with Protocol Aware.

About this task

In this task, you will change the definition of a protocol in order to implement an additional check for an acknowledge bit from the DUT.

Before you begin

Make sure you completed task 2 of the current lab.

Procedure

1. Starting from the test program BasicLab, navigate again to the test suite `PaCounterTest` and its specification file `ProtocolAware.spec`. In the specification file is a reference to the protocol definition file "Lsb4.prot", open this file for editing.
2. In the file "Lsb4.prot" search for the definition of the transaction "write(UnsignedLong IN Data)".
3. Check, how the syntax of the protocol definition file describes the translation of the protocol transactions per protocol signal role.
4. Now change the protocol definition in that way, that the transaction "write(..)" expects the "Ack" to be driven to "1" by the DUT. In order to change this for the protocol signal role "Ack", set for the field "ExpectAck" the state character from "X" to "H".
5. Run the lab test program and use in debug mode the **Timing Debug View** to check, if the modified transaction "write(..)" expects a valid "Ack" value.

Lab 16 Summary

What you have learnt

1. The four different file types required for a protocol aware test in SmarTest 8.
2. The difference between defining a transaction sequence and its instantiation.
3. The way, how dynamically the data of transaction sequences can be modified at execution time to write or expect device specific data in transactions.

Lab 17: Setup Generation in Test Methods

About this task

Generate a setup for a test in a test method using the `Device Setup API`. Use a `parameter group` as an input parameter.

Before you begin

Make sure SmarTest is running, the lab device is loaded, the Lab `Test Results` has been done and the two training presentations on setup generation in test methods and on parameters in test methods are known.

Task 1: Implement a Generic Test Method that Bursts Patterns

In this task, you will enhance an already existing skeleton. This will be done by re-using source code shown in the two training presentations on setup generation in test methods and on parameters in test methods.

Before you begin

Access and `activate` your test program.

About this task

- Add definition of the parameters of an `input parameter group` for the patterns to be bursted.
- Add retrieval of result.
- Add creation of the measurement setup with specification file and operating sequence.

Procedure

1. Use the **Package Explorer** to open in the editor the test method `BurstGroup.java` in the folder `src/testMethods/acLib`. In the following steps this test method will be enhanced.
2. Add the definition of the parameters of the input parameter group as shown in the training presentation on parameters in test methods with the three variables:
   a) the path to the pattern file: `path`
   b) the functional test descriptor to data log fails of the pattern: `patternFtd`
   c) the optional time to wait after the pattern has been executed: `waitTime`

```
public String path = "";
public IFunctionalTestDescriptor patternFtd;
public Double waitTime = 0.0;
```

3. Open in the folder `src/testMethods/acLib` the test method `ProgrammaticTestSetup.java`, which has been introduced in the training presentation on setup generation in test methods. From this test method, copy over the following parts into the `setup()` part of the test method `BurstGroup.java`:
   a) creation of the device setup API instance
   b) setup of digInOut signals given by input parameter `signalsInPattern`
   c) setup of dcVI signals given by input parameter `signalsToMeasure`
   d) import of the specification file given in the input parameter `specFileToImport`
   e) at the end the setting to make the generated setups valid for the measurement
4. Copy the function to define a wavetable from **ProgrammaticTestSetup.java** to `BurstGroup.java`.
5. Add source code to generate the burst of patterns in the operating sequence as shown below (see also the training presentations on parameters in test methods).

```
// Build operating sequence
        deviceSetup.sequentialBegin();
        {
            for (PatternSetting singleGroup : patternGroup.values()) {
                deviceSetup.patternCall(singleGroup.path);
                if (singleGroup.waitTime > 0) {
                    deviceSetup.waitCall(singleGroup.waitTime);
                }
              //  deviceSetup.actionCall(dcVISetup.vmeas().setAverages(16));
            }
        }
        deviceSetup.sequentialEnd();
```

6. In the `update()` part of the test method add the setting to store call pass/fail results if the `loglevel` of the test descriptor `mainFtd` is larger than 30. For help check `ProgrammaticTestSetup.java`.

```
// Enable the call pass/fail when log level is 30 and higher.
        if (mainFtd.getLogLevel() >= 30) {
```



```
measurement.digInOut(signalsInPattern).result().callPassFail().setEnabled(true);
       }
       // Set loglevel of test per pattern to the loglevel of the main test
descriptor
       for (PatternSetting singleGroup : patternGroup.values()) {
           singleGroup.patternFtd.setLogLevel(mainFtd.getLogLevel());
       }
```

7. In the `execute()` part of `BurstGroup.java` uncomment lines to preserve results. After `releaseTester()`, add source code to check if the corresponding result handle contains pass/fail results. If results exist
   a) store results per pattern in the variable `patternPassFailResults` using the result handle for pass/fail results and the function `getPatternPassFail()`
   b) evaluate results as shown in the training presentation on parameters in test methods.

```

         // next test suite can start and use tester resources
         releaseTester();

         if (callPassFailResults != null) {
             IResultPatternPassFail[] patternPassFailResults =
  callPassFailResults.getPatternPassFail();
             int patternCounter = 0;
             for (PatternSetting singleGroup : patternGroup.values()) {
                 singleGroup.patternFtd.setTestText("ResultOfGroup" +
  singleGroup.getId());

  singleGroup.patternFtd.evaluate(patternPassFailResults[patternCounter].getSignalPassFail());
                 patternCounter++;
             }
         }

         // for all sites: check global pass/fail and send results to datalog
         mainFtd.evaluate(measurementResults);

         // set output parameter: pass/fail result of execution
         results = mainFtd.getPassFail();
```

8. In the testflow `TestMethodLab.flow`, that was already used in the lab 12 "Test Method Creation", add in the `setup` part the new test suite `Burst`. as shown below

```
 suite Burst calls testMethods.acLib.BurstGroup {
           specFileToImport = setupRef(crossconnect.common.SignalGroups);
           patternGroup[Pat1] = {
               path = setupRef(crossconnect.basicDigital.patterns.FunctionalA);
               waitTime = 10e-6;
           };
           patternGroup[Pat2] = {
               path = setupRef(crossconnect.basicDigital.patterns.FunctionalB);
               waitTime = 5e-6;
           };
       }
```

9. Add in the `execute` part of the testflow the execution of the new test suite `Burst`.
10. Execute the testflow `NewTestMethodFlow` and check for the new test suite the following:
    a) The generated specification and operating sequence files in folder `src/dsa_gen`.
    b) The test results per pattern in the **Results View**.

Task 2: Add Actions to Measure Voltages after each Pattern Execution

In this task, you will enhance the test method BurstGroup.java such that after each pattern execution (and its wait time) an action to measure voltages is performed and the results are datalogged. The datalogging requires additional parametric test descriptors depending on the number of bursted patterns. Therefore, the additional test descriptor is part of the parameter group: If an additional pattern should be added to the burst, then an additional parameter group needs to be defined and automatically an additional test descriptor will be instantiated.

Procedure

1. Add actions of type `vmeas` after each pattern execution (and its wait time): These actions measure the voltage levels for signals given by the input parameter **signalsToMeasure**. They must be placed in the `setup()` part of the test method, in particular in the loop over the parameter group that is used to build the operating sequence. Insert the following line after the `waitCalls`:

```
deviceSetup.actionCall(dcVISetup.vmeas().setAverages(16));
```

2. Preserve the results of the action by inserting the following line in the `execute()` part of the test method:

```
IDcVIResults dcResults = measurement.dcVI(signalsToMeasure).preserveResults();
```

3. To be able to test and datalog the results of the voltage measure actions, for each action (i.e. per pattern) an additional `test descriptor` is needed. This is accomplished by adding a test descriptor to the definition of the parameter group `PatternSetting`:

```
/** Dedicated parametric test descriptor for the actions */
public IParametricTestDescriptor actionPtd;
```

4. To evaluate the results of voltage measure actions, insert the following line after evaluating the pass/fail results of the patterns:

```
singleGroup.actionPtd.evaluate(dcResults.vmeas("").getVoltage(), patternCounter);
```

5. Execute the testflow `TestMethodLab` and check, if for the test suite `Burst` the voltage results of the added actions are shown in the **Result View**.
6. Set limits for the test descriptors `patternFtd` and `actionPtd` in the table **Tests** of the Test Table file `src/crossconnect/common/BasicLabTestTable.ods`. Use the commented test table `/tmp/BasicLabTestTable.ods` or the test names of the **Test** tab of the **Result View** to get help to correctly specify the fully qualified name of the tests in the Test Table.

Lab 17 Summary

What you have learnt

- How to implement a test method that generates setup files in the "setup()" part of the test method.
- How to find the generated setup files.
- How to use parameter groups to provide an arbitrary number of (groups of) arguments then calling a test suite from the test flow.
- How to generate an arbitrary number of test descriptors in the test method.

Lab 18: Debugging Measurement Setups

About this task

Learn how to debug a setup by executing a measurement, changing its setup and executing it again while staying in the debug mode.

Before you begin

Make sure SmarTest is running, the lab device is loaded the setup generation in test methods and on parameters in test methods are known.

Task 1: Modifications of Static Setups defined in SSF Setup Files

Modifications of static setups defined in SmarTest Sestup Format files.

Before you begin

Make sure SmarTest is running, the lab device is loaded, the `Lab Setup Generation in Test Methods` has been done.

About this task

In the following it is shown how to work with modified static setups of a measurement while in the debug mode. As an example, the action property `average` is changed for a voltage measurement action of the `DcViMeasurements` test suite. `Average` specifies the number of voltage measurements to execute for an averaged result value. You can start your debug session from the **Testflow View** or from the **Flow Chart**.

Procedure

1. From the **Testflow View**, Right click the Test Program and Select **Debug** to start a debug session.
2. Right click and use **Expand** to see all testflows and test suites.
3. In debug mode, right-click on the test suite `DcViMeasurements` of the testflow `NewTestMethodFlow` and select **Execute**.
4. Check the generated operating sequence in the **Operating Sequence View**. If needed, select in the right-click menu **Reset Layout** to see the actions of all signals used in the operating sequence. Note, for the various actions of signal `D01`, the execution time is the same.
5. Open the **Measurement View** and select the tab **Action** on the upper right side. Check in the top right corner, if the button **Hide HW Default Value** is not selected: You should be able to see all action properties. Select the tab **dcVI** at the botton and the tab **Action** on the upper right side. Set for signal `D01` and its `vmeas` action `voltMeas3` the action property `Averages` from 1 to 128.

Temporary modifications

6. Execute again the measurement, i.e. `measurement.execute()`
   a) Execute it directly using the green **play** button which is inside a gear-wheel in the **Measurement View**.
   b) Or execute again the measurement as part of the `execute()` part of the test method which additionally retrieves and evaluates results and writes results to datalog and **Result View**: Use `Execute` in the **Testflow View** as described above in step 3.
7. Check the generated operating sequence in the **Operating Sequence View**. Note, the time required to execute the `voltMeas3` action is now much larger because the number of voltage measurements for an averaged result value is now 128.
8. Terminate the debug session ("stop" button with red square). Start debug mode again and execute again the test suite `DcViMeasurements` as described above.
9. Check the generated operating sequence in the **Operating Sequence View**. Note, the time required to execute the `voltMeas3` action is still large.
10. Terminate the debug session. Run **Load and Restore** for the whole test program of the main flow and start debug mode again. Execute again the test suite `DcViMeasurements` as described above.
11. Check the generated operating sequence in the **Operating Sequence View**. Note, the time required to execute the `voltMeas3` action is short again, because the changed value of the action property `Averages` in the **Measurement View** is lost after executing **Load and Restore.**

Permanent modification

12. Permanent modification of the measurement setup defined in static files:
    a) While still in debug mode, open the **Measurement View** and select the tab **Action** on the upper right side. For signal `D01`, right-click into the cell of its `vmeas` action `voltMeas3` and select from the pop-up menu **Open Specification Editor**. This opens the corresponding specification file in the editor

showing the definition of the action `voltMeas3`. Add a line `averages = 128;` in the definition to set the action property `Averages` from 1 to 128.

13. Save the specification file.

14. Execute again the measurement using one of the following methods:

a) Execute it using the green **play** button which is inside a gear-wheel in the **Measurement View**

b) Use `Execute` in the **Testflow View** as described above in step 3

15. Check the generated operating sequence in the **Operating Sequence View**. Note, again the time required to execute the `voltMeas3` action is now much larger because the number of voltage measurements for an averaged result value is 128.

16. Terminate the debug session (**stop** button with red square). Start debug mode again and execute again the test suite `DcViMeasurements` as described above.

17. Check the generated operating sequence in the **Operating Sequence View**.

18. Note, the time required to execute the `voltMeas3` action is still long, because changing the specification file and saving it to hard disc is permanent.

19. Execute **Load and Restore** Start in Debug mode again and execute again the test suite DcVIMeasurement as describe above. Repeat the steps 17 and 18.

20. Terminate the debug session.

Task 2: Modifications of Dynamic Setups generated in Test Methods

In the following it is shown how to work with modified dynamic setups of a measurement while in the debug mode.

Before you begin

Make sure SmarTest is running, the lab device is loaded, the Lab `Setup Generation in Test Methods` has been done.

About this task

As examples, an action property and the wait times of the operating sequence of the `Burst`" test suites are changed.

Procedure

1. Start debug mode as described for task 1, right-click on the test suite `Burst` of the testflow `NewTestMethodFlow` and select **Execute**.
2. Check the generated operating sequence in the **Operating Sequence View**. If needed, select in the right-click menu **Reset Layout** to see the actions of all signals used in the operating sequence. Note, for the various actions of signal `R04`, the execution time is the same.
3. Temporary modification of the measurement setup generated dynamically in test methods - **Measurement View**:
   a) Open the **Measurement View** and select the tab **dcVI** at the bottom and the tab **Action** on the upper right side.
   b) Set for signal `R04` and its `vmeas` action `_action_2` the action property `Averages` from 16 to 128.
   c) Execute again the measurement and check the generated operating sequence in the **Operating Sequence View**. The time required to execute the `_action_2` action is now much larger because the number of voltage measurements for an averaged result value is now 128.
4. Terminate the debug session (**stop** button with red square). **Load and Restore...**. Start debug mode again and execute again the test suite `Burst` as described above. Open the **Operating Sequence View**. The time required to execute the `_action_2` action is short again, because the changed value of the action property `Averages` in the **Measurement View** is lost after execution **Load and Restore...**.
5. Temporary modification of the measurement setup generated dynamically in test methods - generated specification file:
   a) While still in debug mode, open the **Measurement View** and select the tab **dcVI** at the bottom and the tab **Action** on the upper right side.
   b) For signal `R01`, right-click into the cell of its `vmeas` action `_action_2` and select from the pop-up menu **Open Specification Editor** to open the corresponding specification file in the editor. Note, this file was generated from the test method `BurstGroup.java` using the `Device Setup API`.
   c) Set `averages = 128;`, save the specification file, execute again the **Measurement View** and check the generated operating sequence in the **Operating Sequence View**: The time required to execute the `_action_2` action is now much larger.
6. While still in the debug mode, right-click on the test suite `Burst` in the **Testflow View** and select now **Rebind and Execute**: This executes the `setup()` part (which generates the specification file) the `update()` part and also the `execute()` part of the test method.
7. Check the generated operating sequence in the **Operating Sequence View**: The time required to execute the `_action_2` action is now short again, because the specification file, that was changed, was generated again from the test method. Check in the specification file and in the **Measurement View**, that the action property `averages` is again 16.
8. Try out the following: In debug mode change again the specification file, terminate the debug session and restarting the debug mode: It will also generate the specification files again, overwriting the changes.

9. Permanent modification of the measurement setup generated dynamically in test methods:
   a) If needed, start debug mode. Execute the test suite `Burst` again as described above and open the **Operating Sequence Viewer**. Note that the wait time (with settings like 5us and 10us) is very short compared to the time needed for `vmeas` actions.
10. Open the corresponding specification file or operating sequence file from the **Measurement View**. Scroll up to the head of the specification file. The head consists of comments, that for example give instructions how to open the test method file that generated the specification file using `CTRL+ALT+R` keys. Follow these instructions.
11. In the test method `BurstGroup.java` change the line that inserts the call for waiting in the operating sequence such that the wait time is multiplied by 120:

```
if (singleGroup.waitTime > 0) {
    deviceSetup.waitCall(singleGroup.waitTime * 120);
}
```

12. Execute the measurement from the **Measurement View** and check the generated operating sequence in the **Operating Sequence View**: The wait times have not changed. The reason is, that the modification of the test method affected the `setup()` part, but execution from the Measurement View only executes `measurement.execute()`.
13. Execute the measurement from the **Testflow View** selecting **Execute** and check the generated operating sequence in the **Operating Sequence View**: The wait times have not changed. The reason is, that the modification of the test method affected the `setup()`, but using `Execute` in the **Testflow View** only executes the `execute()` part of the test method.
14. Execute the measurement from the **Testflow View** selecting **Load and Restore** and check the generated operating sequence in the **Operating Sequence View**: The wait times are now larger, the first wait time is almost as long as the time needed for the actions. The reason is, using **Load and Restore** in the **Testflow View** executes the `setup()` part, which has been modified to increase the wait time, the `update()` part and also the `execute()` part of the test method.
15. Terminate the debug session (**stop** button with red square).
16. Start debug mode again and execute again the test suite `Burst` as described above.
17. Check the generated operating sequence in the **Operating Sequence View**.
18. Terminate the debug session (**stop** button with red square).
19. Run **Load and Restore...**and repeat steps 16 and 17.
20. Note, the wait times are large, because changing the test method file and saving it to hard disc is permanent.
21. Terminate the debug session.

Lab 18 Summary

What you have learnt

- How to change the setup of a test suite while in debug mode.
- How to execute the measurement with changes again while staying in debug mode.
- Which modification is permanent and which modification is temporary and gone when leaving the debug mode.

Lab 19: Characterization/Shmoo

Learning objective

Shmoo over Measurement with the `Shmoo View` or shmoo in flow with the `Shmoo Analysis View` can be used to characterize devices.

The interactive way allows you to manually set up shmoo parameters through the SmarTest user interface, click to run it and get shmoo results plotted on-the-fly, while the automated way lets you set up shmoo tests in code, and then have them executed and logged like standard test suites.

Getting started

Make sure SmarTest is running and you have finished lab " Pattern Debug". The Shmoo View is one of the views in SmarTest, available in the Debug perspective.

Task 1: Setting up Shmoo over Measurement (ONLINE)

In this task, you will execute Shmoo over a measurement.

Procedure

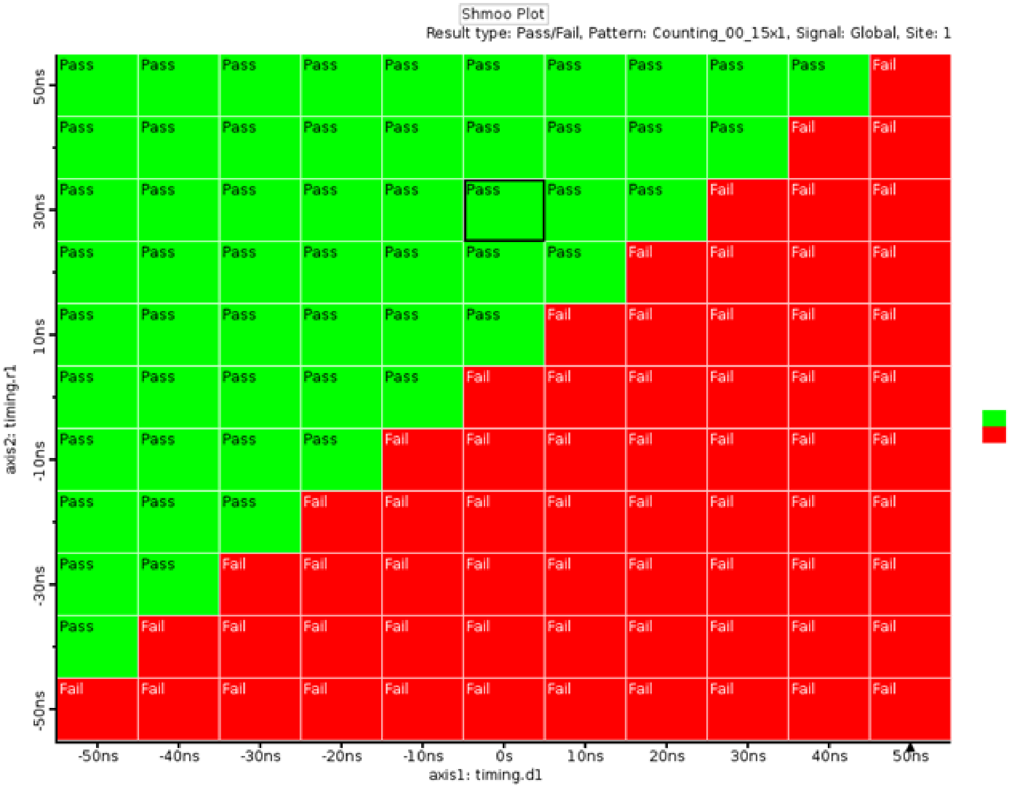
1. Open the `SetupFilesLab.flow`.
2. In the Testflow view, start a debug session for the subflow `NewSubFlow`. Confirm the dialog to open the Debug Perspective.
3. Expand the `NewSubFlow` in the Testflow view to view its test suites.
4. Access and execute the test suite `ExampleTest`.
5. Open the **Shmoo view**.
6. Setup your shmoo parameters as shown in the following picture. Add output signals to the field `resultSignal`.

Setup	Value
Characterization Method	Shmoo
executionOrder	Horizontal
FFC/ErrorCount	OFF
perSignalResult	OFF
perLabelResult	OFF
periodDeviation	
axis1	ON
setupSignal	
resourceType	instrument property
resourceName	timing.d1
format	range
start	-50.000ns
stop	50.000ns
step	#10
scale	linear
axis2	ON
setupSignal	
resourceType	instrument property
resourceName	timing.r1
format	range
start	-50.000ns
stop	50.000ns
step	#10
scale	linear
resultSignal	

Shmoo properties

7. Run Fast Shmoo .

8. Run Shmoo and verify the result.



Shmoo over measurement run

9. (Optional) change parameters, set the FFC/Error count to ON and try other type of shmoo.

Task 2: Analyzing with Shmoo View (ONLINE).

You will use the Shmoo view to perform some typical device characterization tasks.

Overlay shmoo results

1. In the Shmoo view, select a rectangular region around the PASS/FAIL transition.
2. Right click and select **Magnify > 5x** to increase the granularity of the results. If you are not happy with the selected region, right click and select **Back** to undo the Magnify selection.
3. Select site 1 in the **Site Result view** to focus on site one. The shmoo tool is multi site aware, that is, it keeps the result of all active sites and displays the result of the site in focus. All site results are available after execution of a shmoo measurement.
4. Right click in the Shmoo view and select **Add to Overlay**. An **Overlay** tab is added at the bottom and contains the result of site 1.
5. Click site 2 in the **Site result view** to set the focus on site 2.
6. Make sure the **Shmoo** tab is active, right click in the Shmoo view and select **Add to Overlay**. Now, the overlay contents site 1 and site 2 (overlaid).

Sites Shmoo in overlay

7. Perform the same task with the other sites.
8. Note the Pass Fail transition area and the jitter over the sites.
9. Experiment with the overlay features.
10. When you are done, right click the overlay tab and select **Clear Overlay**.

Task 3: Add a shmoo setup to a testflow

You will add a shmoo setup to a testflow.

Procedure

1. While in debug mode with the shmoo open, press **Copy Setup Into Clipboard** in the shmoo view.
2. Copy the content in the clipboard into the setup part of the testflow `SetupFilesLab.flow` to insert a shmoo setup to the testflow.
3. Add the `execute()` part of the testflow file `shmooOverExampleTest.execute().`
4. Terminate the debug session.
5. Save the testflow file and run the test program again.
6. In the **Result view**, tab **Test**, search for the results of test suite `shmooOverExampleTest`: Use the filter for the column **Fully Qualified Name** and type `shmoo` in the light blue cell below the head line.
7. Right click on **ShmooOverExampleTest** and select Show in and then **Shmoo Analysis.**
8. Verify the results.

Task 4: Link Shmoo with Measurement View

You will use the Shmoo result of site 1 and link it with the measurement view.

1. In the Testflow view, start a debug session for the subflow `NewSubFlow.`
2. Access and execute the test suite `ExampleTest`.
3. Open the Shmoo view and run the Shmoo.
4. Select one cell in the shmoo and see the axis values (x and y).
5. Right click in the cell and select **Apply to Instrument**. This will modify the values displayed in the **Measurement view**. Now you could open the **Timing Debug view** and perform a scope measurement based on the actual value (coming from the shmoo tool) or execute from the **Measurement view**.
6. Terminate the debug execution.

Lab 19 Summary

What you have learnt

- How to configure and run shmoo.
- How to analyze results of a shmoo execution.
- How to transfer a shmoo setup to a testflow.

Quick reference cards

SmarTest provides several keyboard shortcuts for standard procedures like switching the perspective or calling for context sensitive information.

Note	These keyboard shortcuts reflect the default SmarTest configuration as it is shipped. Because users can change shortcuts, the default shortcuts may be different from the shortcuts on a user configured system. For details on how to apply the default shortcut scheme see *Adapting keyboard shortcuts (key bindings)* on page 142.

For your convenience you may want to print out the following shortcut cards: *Key bindings.pdf*

Editor key bindings

Working with editors

Save current editor	Ctrl S
Undo last action	Ctrl Z
Go to last edited position in editor	Ctrl Q
Delete current line in editor	Ctrl D
Minimize / Maximize current editor	Ctrl M
Go to line	Ctrl L
Format source code	Ctrl Shift F
Adjust import statements	Ctrl Shift O

Debugging key bindings

Debugging

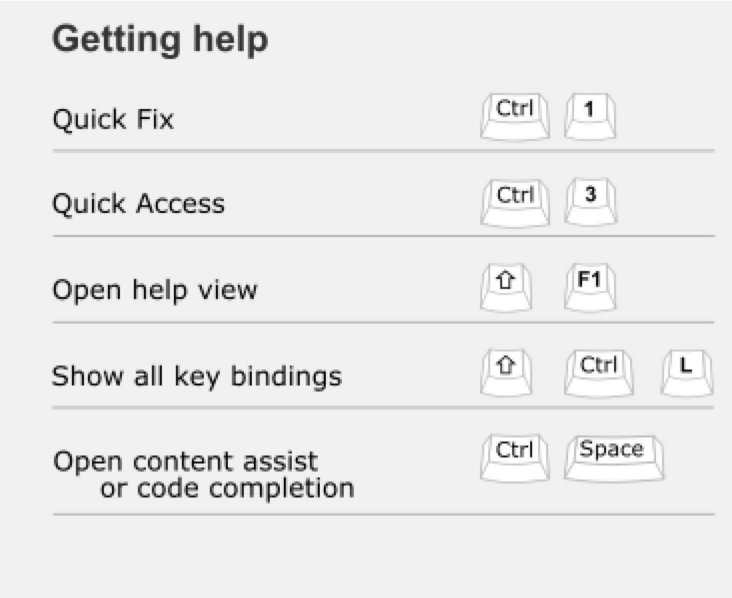
Step into	F5
Step over	F6
Step return	F7
Resume	F8
Terminate	Ctrl F2

Navigation key bindings

Navigating easily

Open editor	F3
Open resource	⇧ Ctrl R
Quick class outline	Ctrl O
Switch between editors	Alt → / ←

Help key bindings



Related tasks

Adapting keyboard shortcuts (key bindings) on page 142
This topic describes how to adapt keyboard shortcuts using the preferences page *Keys*.

Related reference

Keyboard shortcuts (key bindings) - context related on page 139
These keyboard shortcuts reflect the default SmarTest configuration as it is shipped. Because users can change shortcuts, the default shortcuts may be different from the shortcuts on a user configured system.

Useful keyboard shortcuts (key bindings)

SmarTest provides several keyboard shortcuts for standard procedures like switching the perspective or calling for context sensitive information.

Note	These keyboard shortcuts reflect the default SmarTest configuration as it is shipped. Because users can change shortcuts (see *Adapting keyboard shortcuts (key bindings)* on page 142), the default shortcuts may be different from the shortcuts on a user configured system. For details on how to apply the default shortcut scheme.

For your convenience you may want to print out the following shortcuts table:

Key-binding.pdf

General

Shortcut	Description
F3	Open the assigned editor according to the current focus (active selection) Examples: • When, on the Test Program Explorer, a testflow is selected F3 opens it with the default editor for this testflows (flow sequence or flow data editor). • When the focus is on a pattern in the patten explorer F3 opens the pattern editor for this pattern.
Shift+**F1**	Open the dynamic help view
Ctrl+**3**	Open the Quick Access window to quickly access views, commands, preference pages, and others.
Ctrl+**Shift**+**L**	Shows the yellow `key assist` view that lists all available key bindings valid for the active dialog or window

Working with editors

Shortcut	Description
ALT+left / right arrow	Switch between editors according to the navigation history The Eclipse Workcenter keeps a navigation history for the editors. You can activate or even open the editors you used during your session.
Ctrl+**Space**	Start content assist
Ctrl+**J**	Incremental search
Ctrl+**L**	Go to line number
Ctrl+**W**	Close the active editor or dialog view
Ctrl+**Z**	Undo the last action
Ctrl+**.** (period)	Next annotation

Shortcut	Description
Ctrl+**,** (comma)	Previous annotation
Ctrl+**Shift**+**E**	Open the window Switch to Editor to activate an editor or close one or more editors from the list
Ctrl+**Shift**+**R**	Open resource: Opens any file in the workspace

For more information, see *Keyboard shortcuts - context related*.

For all available shortcuts, on the preferences window (on the menu: **Window** > **Preferences**) see **General** > **Keys** (*Adapting keyboard shortcuts (key bindings)* on page 142).

Related reference

Keyboard shortcuts (key bindings) - context related on page 139
These keyboard shortcuts reflect the default SmarTest configuration as it is shipped. Because users can change shortcuts, the default shortcuts may be different from the shortcuts on a user configured system.

Keyboard shortcuts (key bindings) - context related

These keyboard shortcuts reflect the default SmarTest configuration as it is shipped. Because users can change shortcuts, the default shortcuts may be different from the shortcuts on a user configured system.

- For general keyboard shortcuts (`key assist`: **Ctrl**+**Shift**+**L**), see *Useful keyboard shortcuts*.
- To change keyboard shortcuts on the *Key* preferences page.

For your convenience you may want to print out the following shortcuts table:

Setup editor

Command	Shortcut
Insert row after current	**Insert**
Delete selected cells	**Delete** or **BACKSPACE**
Delete selected rows	**Delete** or **BACKSPACE**
Context sensitive help	**Ctrl**+**F1**
Extend selection to the left	**Shift**+**Arrow Left**
Extend selection to the right	**Shift**+**Arrow Right**
Extend selection one up	**Shift**+**Arrow Up**
Extend selection one down	**Shift**+**Arrow Down**
Go to cell to the right	**Arrow right** or **Tab**
Go to cell to the left	**Arrow left or Shift+Tab**
Go to cell one down	**Arrow Down** or **Enter**
Go to cell one up	**Arrow Up** or **Shift+Enter**
Go one page up	**Page Up**
Go one page down	**Page Down**
Go to first line	**Ctrl**+**Home**
Go to last line	**Ctrl**+**End**
Go to first cell in row	**Home**
Go to last cell in row	**End**
Collapse current tree node	**Ctrl**+**Arrow Left**
Expand current tree node	**Ctrl**+**Arrow Right**
Comment selected lines (adds // before the code)	**Ctrl**+**7**
Table key bindings	
Move a row up within Level	**Alt**+**Arrow Up**

Table key bindings	
Move a row to top level	**Alt**+**Arrow Left**
Move a row down within level	**Alt**+**Arrow Down**
Move a row down to a subtree	**Alt**+**Arrow Right**
Switch value between site common mode and site specific mode	**F6**

Common cell editor key bindings	
Edit current selected cell and remove value	start typing or **Space**
Edit current selected cell and keep value	**F2**
Exit from edit mode	**Esc or Tab or Enter**

Compound cell editor (combo + text) key bindings	
Move between combo and text	**Tab**
Exit from edit mode	**Esc** or **Enter**

Java editor

Command	**Shortcut**
Add Block Comment	**Shift**+**Ctrl**+**/**
Add Include	**Shift**+**Ctrl**+**N**
Comment/Uncomment	**Ctrl**+**/**
Find Declaration	**Ctrl**+**G**
Find Reference	**Shift**+**Ctrl**+**G**
Format	**Shift**+**Ctrl**+**F**
Go to Matching Bracket	**Shift**+**Ctrl**+**P**
Go to Next Member	**Shift**+**Ctrl**+**Down**
Go to Previous Member	**Shift**+**Ctrl**+**Up**
Indent Line	**Ctrl**+**I**

Debugging

Command	**Shortcut**
Resume	**F8**
Run to Line	**Ctrl**+**R**
Step Into	**F5**

Command	Shortcut
Step Over	**F6**
Step Return	**F7**
Terminate	**Ctrl**+**F2**

Pattern editor

Shortcut key	Function
Ctrl+Spacebar	Activates the control assist.
Ctrl+Spcebar-Spacebar	Activates the content assist and opens the drop-down menu.
Ctrl+Backspace	Activates the content assist and deletes the current entry.
Ctrl+Backspace-Backspace	Activates the content assist, deletes the current entry and shows the complete drop-down menu.
Shift+Left-Mouse button	Selects a range in a table. To select a range in a table, click the left upper cell, press Shift and click the right lower cell.
ESC	Leaves the content assist without changing the value.

Related reference

Useful keyboard shortcuts (key bindings) on page 137
SmarTest provides several keyboard shortcuts for standard procedures like switching the perspective or calling for context sensitive information.

Related information

Setup editor

Pattern editor

Setup perspective

Device Debug perspective

Adapting keyboard shortcuts (key bindings)

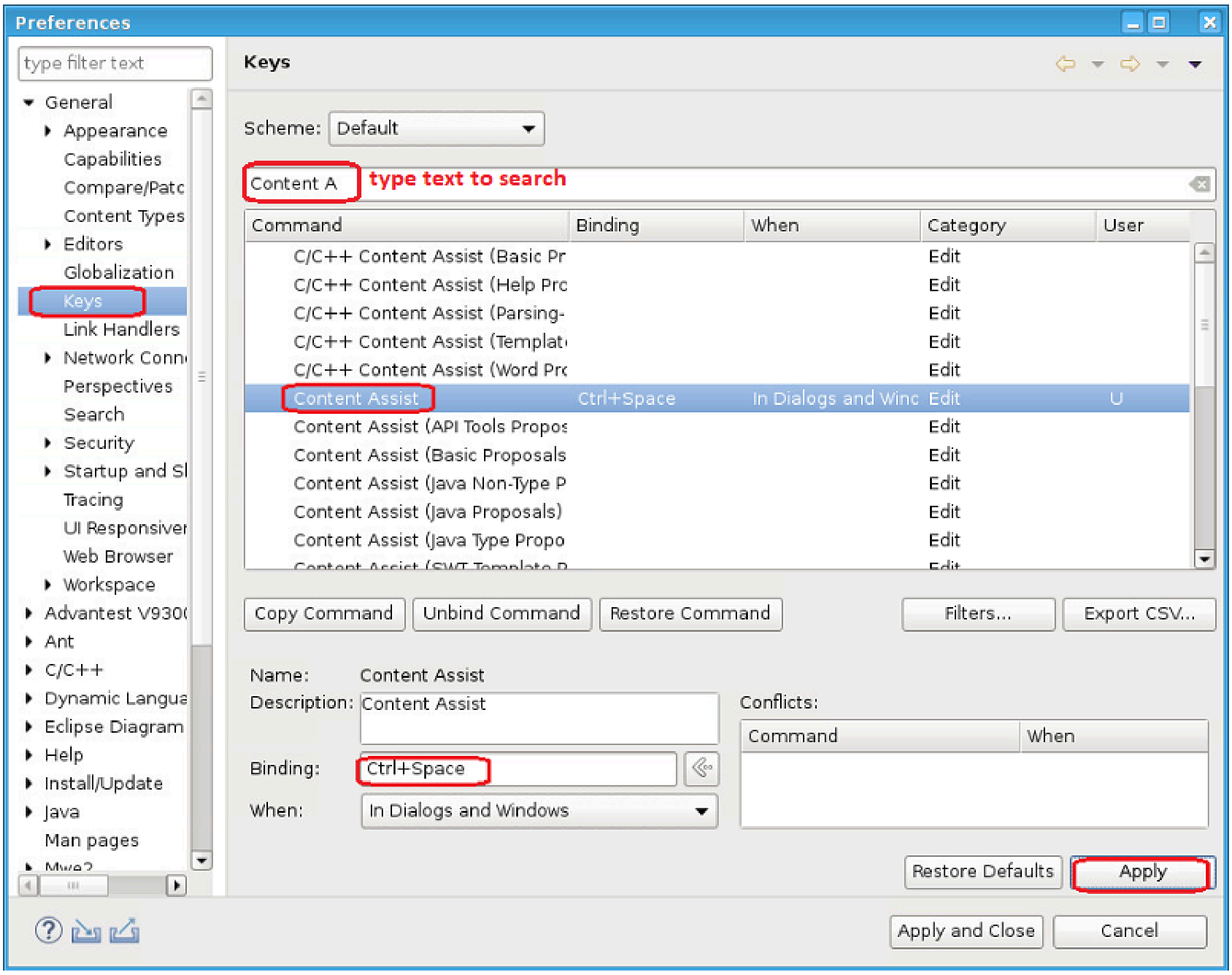
This topic describes how to adapt keyboard shortcuts using the preferences page *Keys*.

About this task

Advantest recommends to use the default keyboard shortcuts which are documented at *Useful keyboard shortcuts* (`key assist`) and *Keyboard shortcuts - context related*.

Procedure

1. On the menu click **Window** > **Preferences**.
2. In the Preferences window click **General** > **Keys**.



Preferences page Key

3. Find out the target key from the key list or use the search bar by typing the key text.

   For example, type **Content** to search for **Content Assist**.
4. Click on the target key and adapt the keyboard shortcut by changing the content of **Binding**.
5. Click **Apply** to apply the change.

Related reference

Useful keyboard shortcuts (key bindings) on page 137
SmarTest provides several keyboard shortcuts for standard procedures like switching the perspective or calling for context sensitive information.

Keyboard shortcuts (key bindings) - context related on page 139

These keyboard shortcuts reflect the default SmarTest configuration as it is shipped. Because users can change shortcuts, the default shortcuts may be different from the shortcuts on a user configured system.