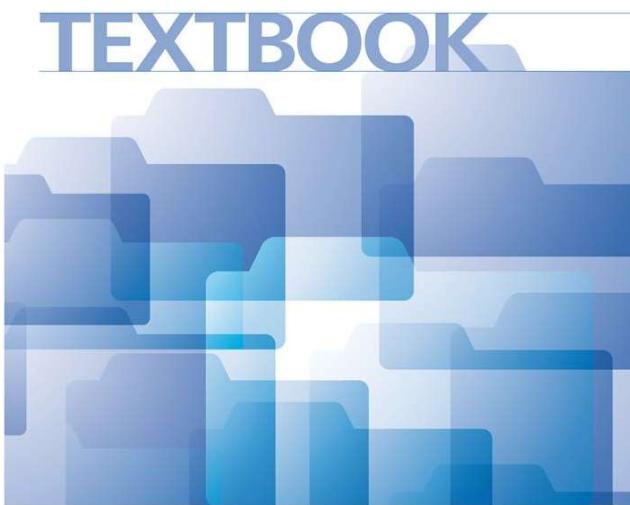


**V93000**  
**SmarTest 8 Basic User Training**

Version 8.2.5

January 2020





# Basic User Training

## Agenda for SmarTest 8.2.5

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION



## Legal Notices

### Notices

All rights reserved. All text and figures included in this publication are the exclusive property of Advantest Corporation. Reproduction of this publication in any manner without the written permission of Advantest Corporation is prohibited. Information in this document is subject to change without notice.

### 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. BY ACCESSING AND OTHERWISE USING THIS TRAINING DOCUMENTATION, YOU ACKNOWLEDGE AND AGREE THAT YOUR USE AND ACCESS HERETO IS GOVERNED BY THE ADVANTEST TRAINING TERMS AND CONDITIONS ("TRAINING TERMS"), WHICH ARE HEREBY INCORPORATED HEREIN BY REFERENCE.

EXCEPT AS OTHERWISE EXPRESSLY SET FORTH HEREIN, ADVANTEST MAKES NO WARRANTY WHATSOEVER WITH RESPECT TO ANY TRAINING COURSE OR TRAINING DOCUMENTATION, INCLUDING SAMPLE SOURCE CODE, PROVIDED IN CONNECTION WITH ANY TRAINING OR ANY TRAINING DOCUMENTATION INCLUDING ANY: (A) WARRANTY OF MERCHANTABILITY; (B) WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE; OR (C) WARRANTY AGAINST INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF A THIRD PARTY WHETHER WRITTEN OR ORAL, EXPRESS OR IMPLIED BY LAW, COURSE OF DEALING, COURSE OF PERFORMANCE, USAGE OF TRADE OR OTHERWISE. ADVANTEST'S LIABILITY RELATING TO THE TRAINING AND THIS TRAINING DOCUMENTATION IS LIMITED IN ACCORDANCE WITH THE TRAINING TERMS. 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.

### 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.

Published on 30 January 2020.



January 2020

All Rights Reserved - ADVANTEST CORPORATION



# Agenda - Monday

Morning	<b>Introduction/Agenda</b> <b>Lecture 01:</b> Introduction, Concepts and Software overview <b>Lecture 02:</b> SmarTest Projects <b>Lab 01:</b> Preparation and Prerequisites <b>Lecture 03:</b> Test Program File
Afternoon	<b>Lecture 04:</b> DUT Board Description File <b>Lab 02:</b> DUT Board Description <b>Lecture 05:</b> Building Blocks of Test Programs <b>Lab 03:</b> Test Program <b>Lecture 06:</b> Testflow and Test Suites <b>Lab 04:</b> Testflow

# Agenda - Tuesday

Morning	<b>Lecture 07:</b> Operating Sequence <b>Lab 05:</b> Operating Sequence <b>Lecture 08:</b> Hardware Overview* <b>Lecture 09:</b> Instruments <b>Lecture 10:</b> Basics Level and Timing* <b>Lecture 11:</b> Specification Files - Digital Setups <b>Lab 06:</b> Timing Sets and Level Sets
Afternoon	<b>Lecture 12:</b> Specification Files - Multi Domains <b>Lab 07:</b> Timing Debug <b>Lecture 13:</b> Patterns <b>Lab 08+09:</b> X-Mode + Pattern Debug <b>Lecture 14:</b> Actions, Patterns and Transactions <b>Lab 10:</b> DC Measurement <b>Lecture 15:</b> Technical Documentation Center *not needed for experienced V93000 users

# Agenda - Wednesday

Morning	<b>Lecture 16:</b> Testflow File <b>Lab 11:</b> Test Method Library* <b>Lecture 17:</b> Test Methods – Introduction <b>Lab 12:</b> Test Method Creation <b>Lecture 18:</b> Test Methods – Basics <b>Lecture 19:</b> Test Methods – Multi Site Types <b>Lab 13:</b> Test Setup Modifications
Afternoon	<b>Lecture 20:</b> Test Methods – Result Access <b>Lab 14:</b> Retrieving Test Results <b>Lecture 21:</b> Datalogging <b>Lab 15:</b> Datalogging <b>Lecture 22:</b> Binning

\*optional

# Agenda - Thursday

Morning	<b>Lecture 23:</b> Usage of Test Tables <b>Lecture 24:</b> Content of Test Tables <b>Lecture 25:</b> Test Program Execution and Debug <b>Lecture 26:</b> Internal Steps of Test Program Execution <b>Lecture 27:</b> Protocol Aware – Introduction <b>Lab 16:</b> Protocol Aware Setup Files*
Afternoon	<b>Lecture 28:</b> Test Methods – Programmatic Setup Generation* <b>Lab 17:</b> Setup Generation in Test Methods* <b>Lecture 29:</b> Test Methods – Parameters* <b>Lecture 30:</b> Debug Tools <b>Lab 18:</b> Debugging Measurement Setups

\*optional

# Agenda - Friday

Morning	<p><b>Lecture 31: Characterization Tools</b> <b>Lab 19: Characterization/Shmoo*</b> <b>Lecture 32: Recommended Setup Structure</b> <b>Lecture 33: Utility Lines</b> <b>Lecture 34: Licensing</b> <b>Lecture 35: TCCT*</b></p>
Afternoon	<p><b>Lecture 36: Test Program Migration Framework*</b> <b>Lecture 37: Conversion of STIL files*</b> <b>Open Questions</b> <b>Feedback</b></p>

\*optional



January 2020

All Rights Reserved - ADVANTEST CORPORATION



# Introduction & Software Overview

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest Introduction & Software Overview - 1

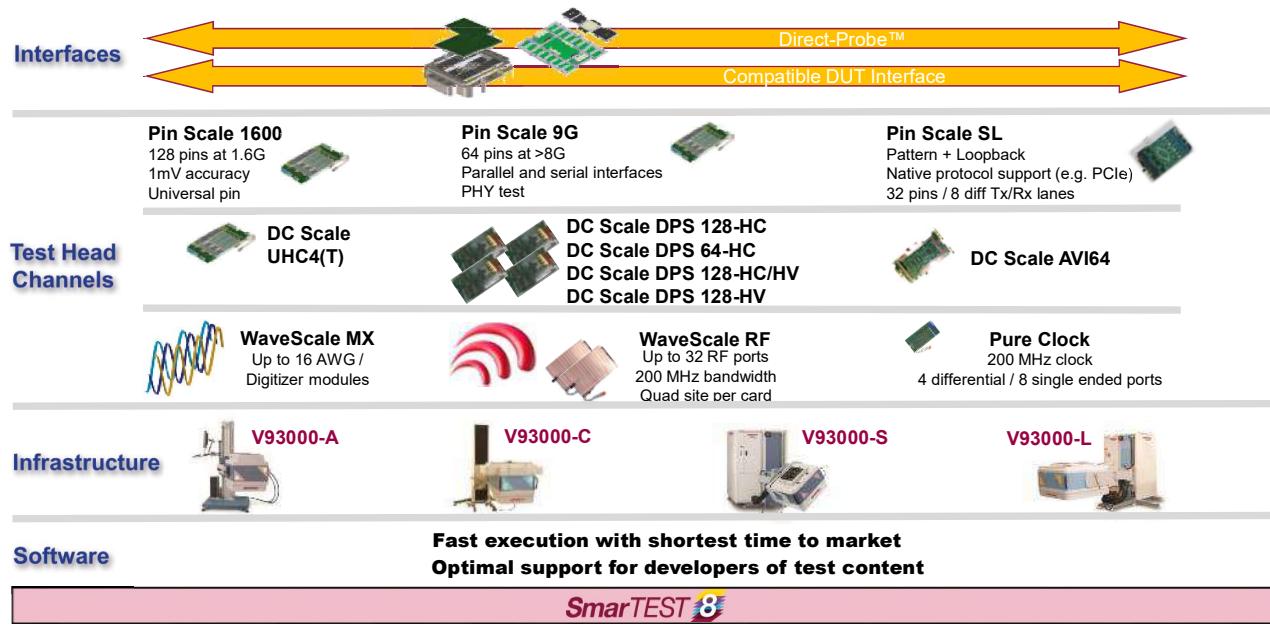
# Learning Objectives

- Understanding how SmarTest 8 makes the life of test engineers easier.
- Understanding the underlying basic concepts of SmarTest 8.

## Key Objectives of SmarTest 8

- Enable the **shortest time** from tape-out to volume production.
- Enable **best test program development efficiency**.
- Offer the **fastest test throughput** for manufacturing test of complex SOC/SiP/TSV and also small low complexity devices.

# SmarTest 8 enables WaveScale and SmartScale



## Agenda

Step by step walk through the list of SmarTest 8 values that simplify test program development and the enabling concepts.

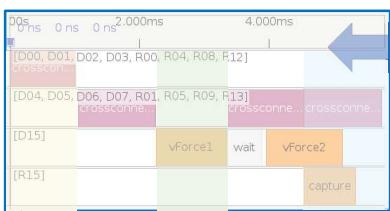
### SmarTest 8 Values

- Simple synchronization across domains
- Unified setups and tools
- Reuse
- Collaborative development
- Assisted setup generation
- Smart test methods
- Java

# Simple synchronization across domains

The **operating sequence** allows to execute test steps across all *instruments* with synchronized timing using a single sequencer start (i.e., fast).

- Easy assembling of test steps (e.g. patterns, transactions, actions) with synchronization.
- Multi-domain – supports all instrument types and all domains:  
Digital, DC, PA, MX, RF.



```
1 sequence ExampleIntroSlides {
2     parallel parGrp1 {
3         sequential seq1_1 {
4             patternCall crossconnect.operatingSequence.patterns.Init InitDUT;
5         }
6     }
7     parallel parGrp2 {
8         sequential seq2_1 {
9             patternCall crossconnect.operatingSequence.patterns.DigTestA FuncA;
10    }
11 }
12 parallel parGrp3 {
13     sequential seq3_1 {
14         actionCall vForce1;
15    }
16 }
17 parallel parGrp4 {
18     sequential seq4_1_1 {
19         patternCall crossconnect.operatingSequence.patterns.DigTestB FuncB;
20     }
21     parallel parGrp4_1_1 {
22         sequential seq4_1_1_1 {
23             patternCall crossconnect.operatingSequence.patterns.DigTestC FuncC;
24         }
25         sequential seq4_1_1_2 {
26             actionCall capture;
27         }
28     }
29     sequential seq4_1_2 {
30         wait 0.6ms;
31     }
32     actionCall vForce2;
33 }
34 }
```

# Unified Setups for Tester Resources

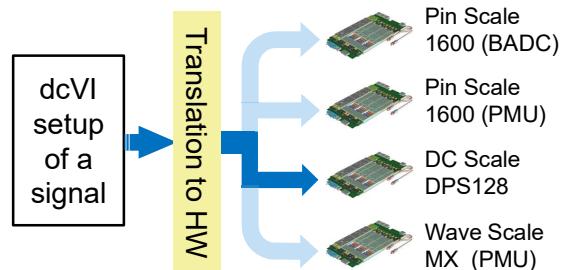
If a task can be done by various hardware resources, the way to set it up is always the same.  
Examples: Force a voltage or measure a current.

Key concepts:

- **Abstraction from hardware**
- Smart test methods

Details on “Hardware Abstraction”:

- Categories of tester resources are described as *instruments*:  
awg, clock, dcVI, digInOut,...
- *Instruments* are controlled via common user interfaces for setup and API, independent of the underlying tester resource.

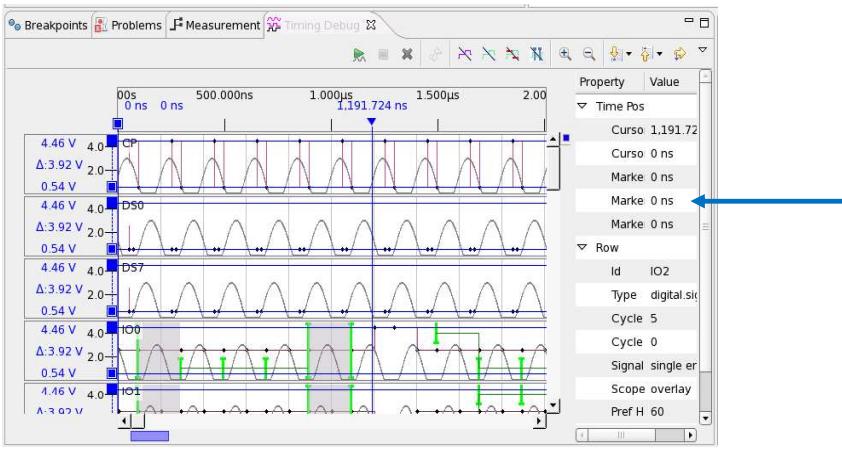


# Convergence of Digital, Analog & RF Tools

SmarTest provides many tools to facilitate debug, i.e. to control and modify settings and to view results.

These are always multi domain tools covering the following domains:

- digital
- analog
- RF
- DC
- PA



Supporting concept:  
• **Multi Domain Tools**

**Examples of multi domain tools:**

- Signal analyzer
- Viewer for operating sequences
- Measurement View
- Tester View
- Timing Debug View

## Reuse of Self-Contained Test Program Parts

Self-contained test program parts consisting of sub-flows with integrated setups and parameterization can be re-used:

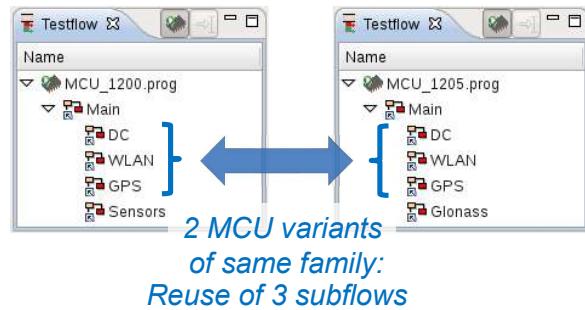
- multiple times with different parameters in a test program
- multiple times in various test programs.

Key concepts:

- **Hierarchical testflows with parameters**
- Modular setups

Details on hierarchical testflows:

- Every node of a *testflow* can be a *test suite* or another *testflow*.
- Every *testflow* can have input and output parameters.



# Team Development of Test Programs

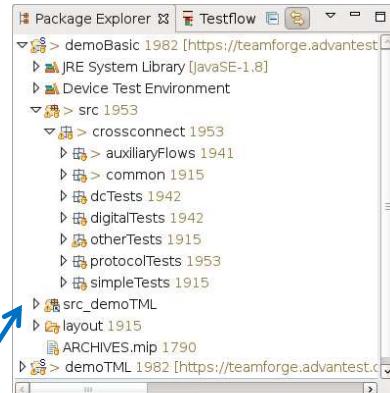
Foster team development: Simple partitioning of test programs and support of industry standard revision control systems.

Key concepts:

- Hierarchical testflows with parameters
- **Modular setups**

Details on modular setups:

- Flexible file structure for setup files
- Allows a dedicated file for a setup entity
- Integrated support for SVN and GIT
- Links to external setup files



*"External" content from another team*

## Setup Languages: Easy to Use

New concepts like the unified setups for tester resources based on *instruments* as well as the *operating sequence* and *testflow files* require new setup languages.

These languages are defined in the **SmarTest Setup Format (SSF)** and have been specifically developed for the different types of setup data with the goal, that they are easy to use.

Examples of different file types of setup data: *Test program file*, *testflow file*, *specification file* containing *instrument settings*, *operating sequence file*.

SmarTest provides APIs for external tools to generate and read setup files correctly.

```
1@ import crossconnect.digitalPattern.levels.FunctionalLevel;
2@ spec Functional_Par3Seq {
3@     // level settings
4@     vcc      = 1.4 V;
5@     IV_Swing = 1.4 V;
6@     OV_Swing = 0.4 V;
7@     // timing settings
8@     per      = 200 ns;
9@     t_drive   = per / 4;
10@    t_exp     = per - 2 ns;
11@    }
12@    setup digInout_allInputs+allOutputs {
13@        result.cyclePassFail.enabled = true;
14@    }
15@ }
```

*Example of a specification file*

```
1@ sequence FunctionalPar3x1x2x3 {
2@     parallel parGrp1 {
3@         sync = start;
4@         sequential seq1_1 {
5@             sequential seq2_1 {
6@                 sequential seq3_1 {
7@                     patternCall crossconnect;
8@                 }
9@             }
10@            }
11@           }
12@             }
13@               }
14@                 }
15@                   parallel parGrp2 {
16@                       }
17@                         }
18@                           }
19@                             }
20@                               }
21@                                 }
22@ }
```

*Example of an operating sequence file*

# Assisted Setup Generation and Navigation

All SmarTest 8 setup format files are based on an easy to learn, intuitive and self explanatory language, which has been newly developed for these purposes.

Content assist aids in using the format.

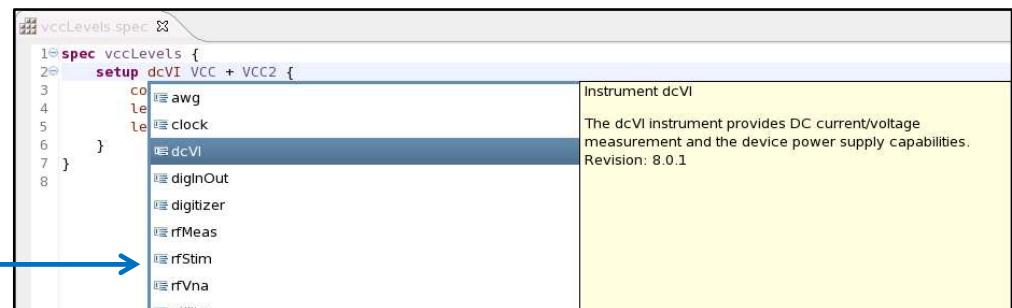
Supporting concept:

Strongly built upon the Eclipse IDE.

## Benefits of the (Eclipse) IDE:

- Improved navigation tools:
  - F3 = “Open Declaration”,
  - “show-in”,
  - context-sensitive help.
- Consistent look and feel.
- Consistent debug for flows and test methods.
- Content assist (suggests words).

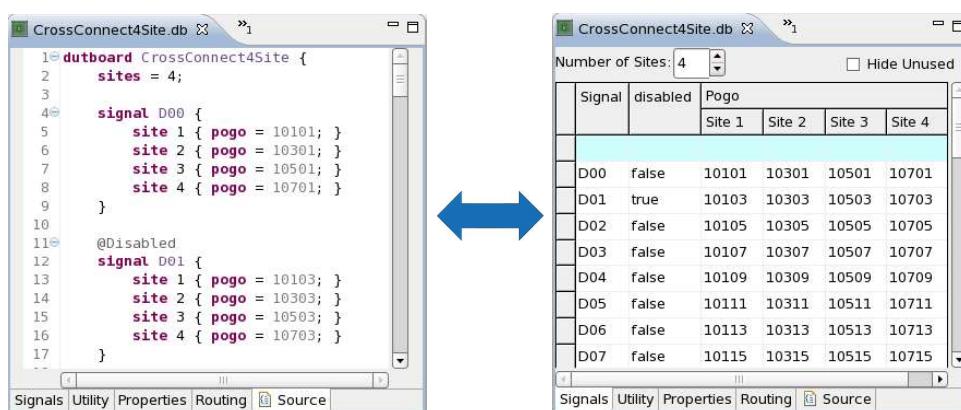
*Example:  
The syntax expects an instrument, so after pressing “ctrl-space” content assist suggests all available instrument types.*



# Assisted Setup Generation and Navigation (2)

Files in SmarTest Setup Format are supported by dedicated editing tools. Their data can be accessed in different views such as **Source** and **Tables**:

- DUT board description
- Instrument properties
- Level settings
- Timing settings



# Smart Test Methods

Examples:

- Test methods are implemented simply as for a single site:
  - SmarTest8 automatically expands to multi-site and manages tester resources between sites.
  - Site-specific data handling is supported.
- Device-test oriented APIs:  
“measurement.execute();” with built-in management for the resources needed, instead of “open relayX; close relayY; run;”.
- Consistent and compact APIs over all *instrument* types and domains.
- Single API call to perform results upload, analysis and evaluation in the background while the following test suites are executed.

## Java Programming Language

Java is the underlying programming language for test methods.

- Large and intuitive set of libraries.
- Large user base for good support.
- Thorough integration with Eclipse: Java is the **native** language of the Eclipse framework.
  - Syntax errors are shown as you write.
  - Automatically determination of variable types by context.
  - Compile as you go: Triggered when saving a file.
  - Integrated memory management.
- Test method debugging is faster and simpler.
- Faster time to market by maximizing the developer’s effectiveness.

# Summary – What you should have learned

We have reviewed SmarTest 8 as a new generation of ATE software targeting simplified test program development, as well as the concepts behind it.

## SmarTest 8 values

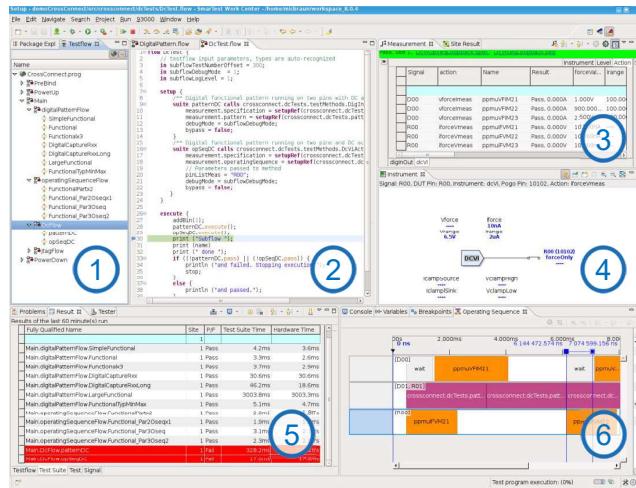
- Simple assembly of tests synchronized across domains
- Unified setups for tester resources
- Unified tools: Convergence of multiple domains
- Facilitated reuse
- Support for collaborative development
- Assisted setups and navigation
- Smart test methods
- Java is the programming language
- Additionally, will be shown in the following modules:  
New, state-of-the-art software with many (graphical) tools

# Backup

# Modern software with many (graphical) tools

New, state-of-the-art tester software based on the Eclipse framework.

The perspectives of the **SmarTest Work Center** contains views designed to let users efficiently setup, execute and debug tests.



(1) Package Explorer

(2) Testflow Editor

(3) Measurement View

(4) Instrument View

(5) Result View

(6) Operating Sequence View

## Previously Complex Setups are Simplified (1)

Examples:

- X-Modes
- Multi-site
- Synchronization
- Concurrency
- Background processing

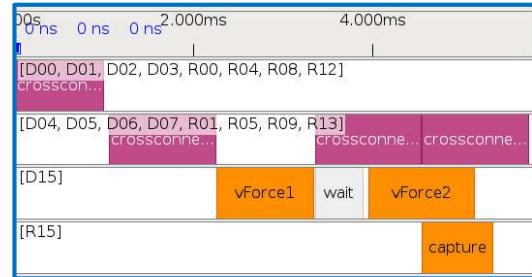
- Site-Interlacing
- ...

Supporting concepts:

- **Operating sequence**
- Smart test methods

Details on *operating sequences*:

- Any test execution is sequencer controlled.
- Patterns, protocols and *actions* across all domains (digital, analog and RF) are automatically synchronized.
- Concurrent execution using multiple instruments.



## Previously Complex Setups are Simplified (2)

Examples:

- X-Modes
- Multi-site
- Synchronization
- Concurrency
- **Background processing**
- Site-Interlacing
- ...

Supporting concepts:

- *Operating sequence*
- **Smart test methods**

Benefits of the new test method APIs:

- Multi-site aware variable types and API calls
- Simple command to start background processing

## Recap of SmarTest 8 Terms

• **Signal:**

Logical input to or output from a DUT;  
enters/leaves via a device pin.

• **Instrument:**

Category of tester resources:  
awg, clock, dcVI, digInOut,..

• **Action:**

Command to tester that is not a sequencer  
instruction; attached to vectors or included in  
operating sequences.

• **Operating Sequence:**

Arrangement of patterns and actions to be  
executed that apply to one or more signals or  
signal groups.

The arrangement can be serial, parallel, or a  
combination of both.

• **Binding:**

Process of converting and optimizing test data  
for one measurement into an executable test.

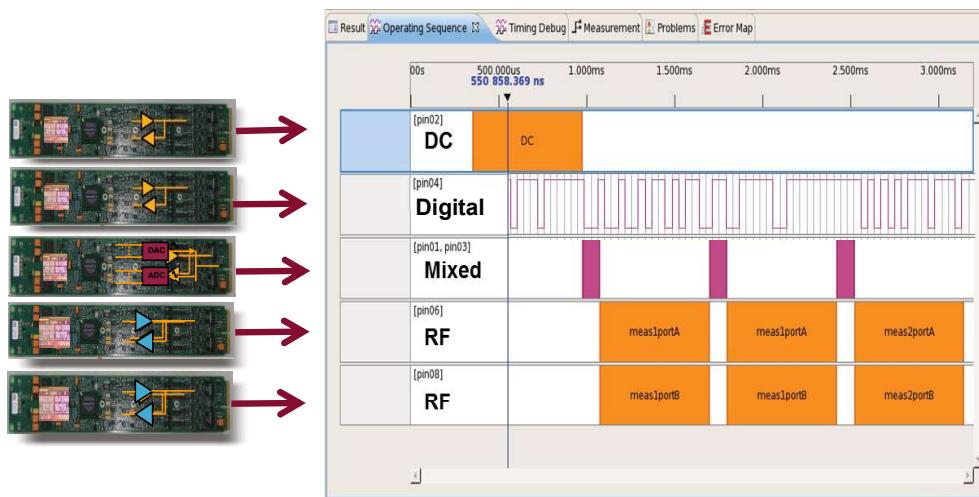
More: „Glossary of terms“ in the TDC.

# SWC Icons

SmarTest and Eclipse icons/label decoration overview

	Project		Class
	Program, SpecFile		Flow
	Container		Scrapbook Page
	Source Folder		Pattern
	Package		Warning
	Java Source Code		Error
	API Library		
	Java Byte Code		

## Change: Test Processors Drive Every Signal



Best Performance: One-Shot-Execution with defined timing

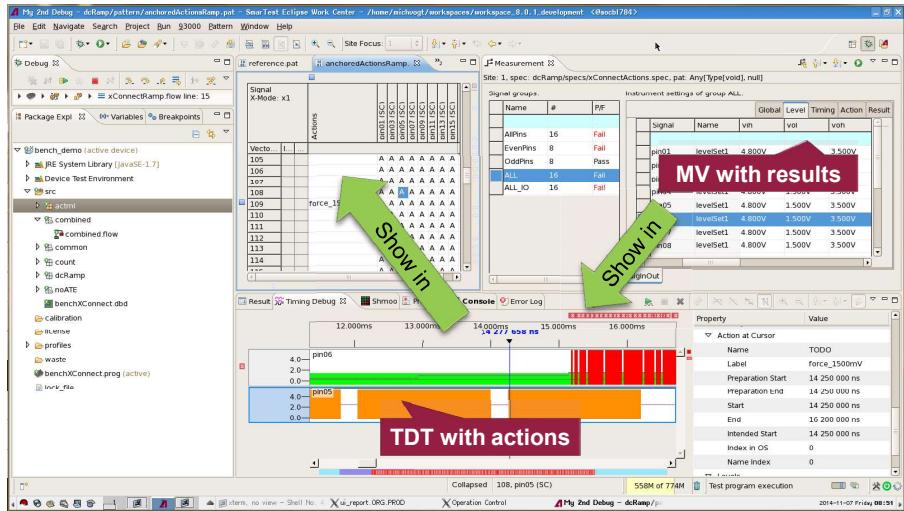
→ reproducible, high throughput measurements – but requires a test processor behind every signal.

All setups are translated into (sequencer) programs for the test processors.

Optimal throughput and built-in synchronization across all domains, concurrently.

# Navigation in Setups

- SOC test programs are often large, modularity increases the amount of setup files.
- Built-in navigation features (F3, right-click, “show in”) make the navigation easy.
- Typically no need to open files from *project explorer* – right-click navigation is faster.



SmarTEST 8

January 2020

All Rights Reserved - ADVANTEST CORPORATION

ADVANTEST

SmarTest Introduction & Software Overview - 24

SmarTEST 8

## SmarTest Projects

### SmarTest 8.2.5 Training

January 2020

SmarTEST 8

January 2020

All Rights Reserved – ADVANTEST CORPORATION

ADVANTEST

SmarTest 8 Projects - 1

# Learning Objective

- Understand the purpose and characteristics of a SmarTest project.

# Agenda

- Scope of a SmarTest project
- Creating a SmarTest project
- Importing an existing project
- SmarTest project structure

# Eclipse, the Foundation of SmarTest 8 and its Projects

The user interface and foundation for SmarTest 8 is Eclipse.

It is an open source software collection to support developing, building, deploying and maintaining of software.

For SmarTest 8, the features provided by Eclipse have been leveraged and expanded to build a state-of-the-art tester software, called the *SmarTest Work Center* (SWC).

## SmarTest 8 organizes test programs in enhanced Eclipse projects:

- It contains program code, libraries, compiled code and management information.
- The content is stored in a special folder in the file system.

The programming languages of SmarTest 8 are

- The *SmarTest Setup Format (SSF)* for setup files;
- Java for test methods.

# What is a SmarTest 8 Project?

A SmarTest project is a folder that includes one or more *source folders* as subfolders.

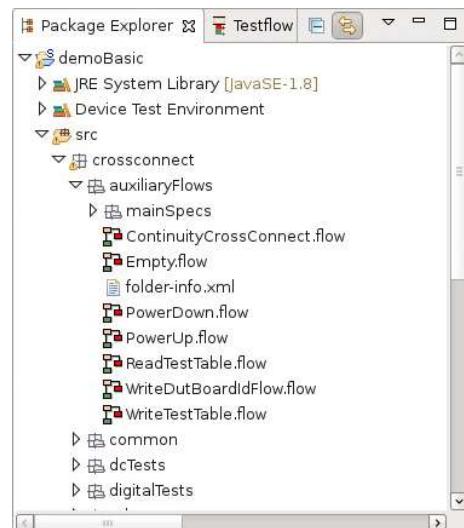
For setting up a test, the project must also contain at least one *test program file* and the referenced setup files, for example *testflow* and *pattern files*.

The shown *package explorer* allows to browse through the files of a project.

A project contains:

- Java Libraries
- V93000 System Libraries
- Source Folders
- Configuration Files
- Compile/Build Directories

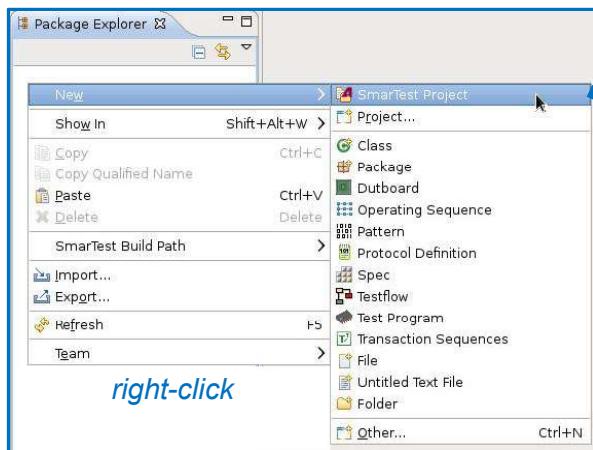
A SmarTest 8 project may refer to multiple other projects, for example a library of test methods..



# Example Scope of a SmarTest Project

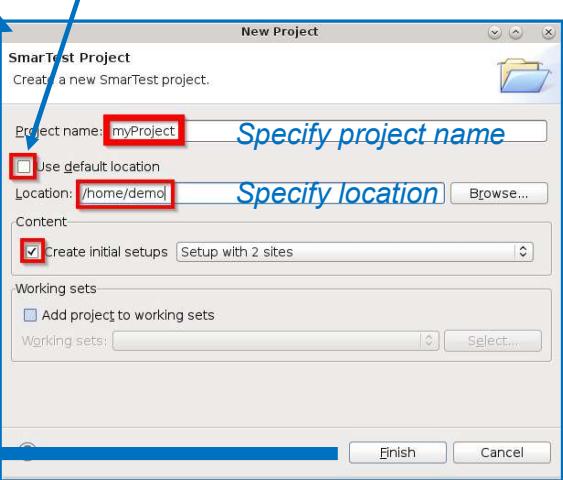
- A comprehensive test program for a semiconductor device.
- A test method library in a single, dedicated SmarTest project.
  - Measurement and data processing algorithms available to various test programs.
- A segmented portion of a specific device test program.
- A program library to process data with specific algorithms.
- Parts of test programs of multiple devices:
  - Multi-die module
  - Leveraged IP core
  - Integrated SOC

## Creation of a New SmarTest Project

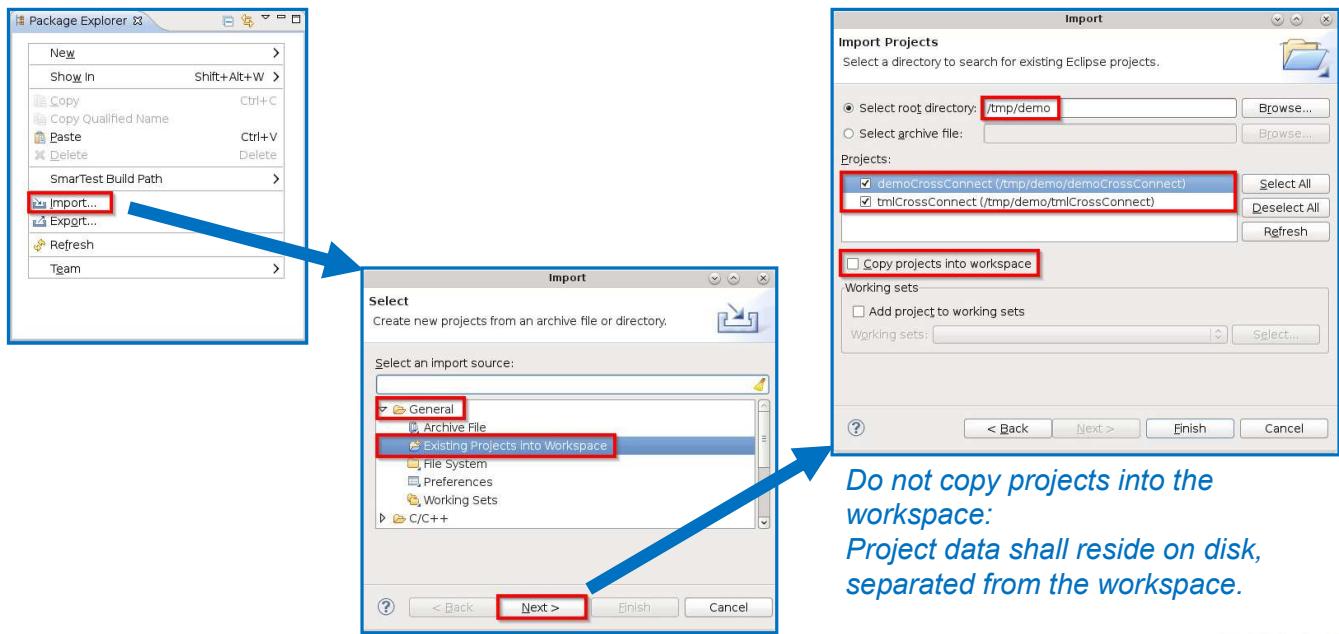


If "Create initial setup" is checked, the new project will already contain an example setup.

**Recommendation:**  
Do not use the default location.  
Select an empty folder which is not in the workspace.



# Import of an Existing Project

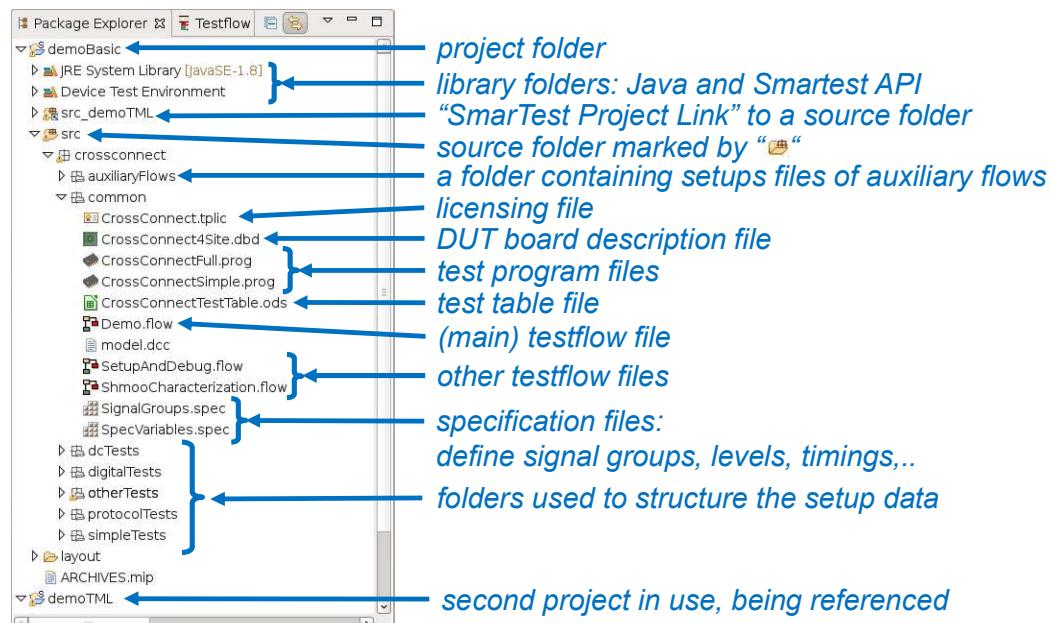


*Do not copy projects into the workspace:  
Project data shall reside on disk,  
separated from the workspace.*

## Logical Hierarchy in SmarTest Projects

- A SmarTest project for testing a device contains one or more *test program files*.
- A *test program file* references the *main flow* and *auxiliary flows*, which use setup information stored in a hierarchy defined by the user.
- All the setup information (test programs, testflows, test methods, specifications, patterns, limits, bins,...) can be found in specific folders declared as *source folders*.
- The *package explorer* allows to browse through the file structure of SmarTest projects.

# SmarTest Project: Folders and Files



## Summary - What you should have learned

- The *SmarTest Work Center* is based on Eclipse.
- Test programs are developed in SmarTest projects which build on Eclipse projects.
- The scope of a project can be
  - a complete test program, or
  - just a part of a test program or
  - (parts of) multiple test programs.



# Test Program File

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

SmarTest Test Program File - 1

## Learning Objective

- Learn about the role, the content, and the syntax of the *test program files*
- Understand the format of the test program files and other setup files and learn, how content assist facilitates working with this format



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

SmarTest Test Program File - 2

# Agenda

- Purpose and content of a *test program file*
- Examples of *test program files*
- Editing *test proram files*

## SmarTest 8 Building Blocks: Test Program File

Test  
Program

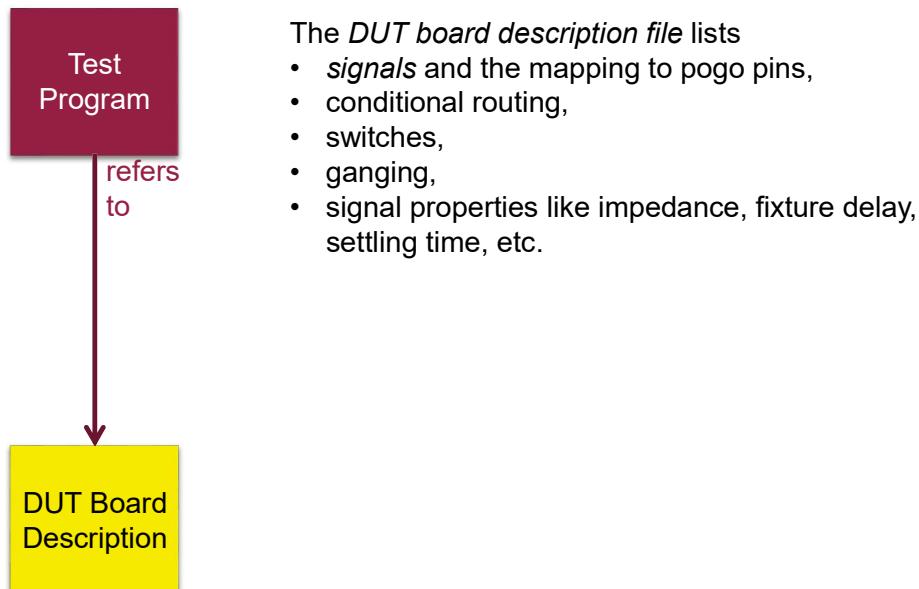
The *test program file* is in the **top position of the file hierarchy** of a SmarTest 8 project.

It contains or refers to all the information required to describe the test program for a specific DUT.

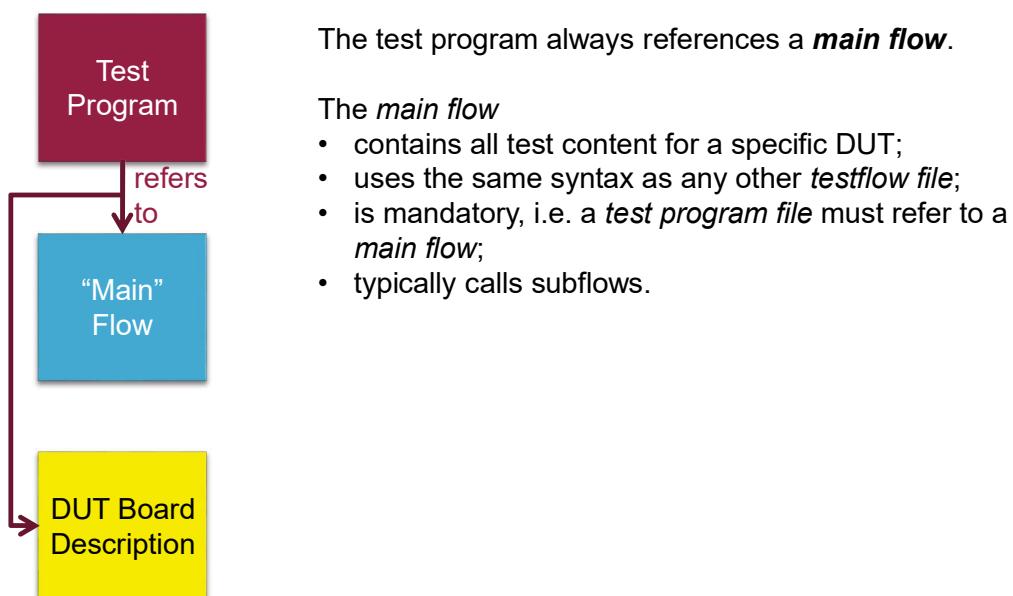
The *test program file* points to licenses, sets global variables, refers to flows to be executed, determines which sites are used, and more.

The *test program file* must be stored like all other setup files within a *source folder*.

# Building Blocks: DUT Board Description File



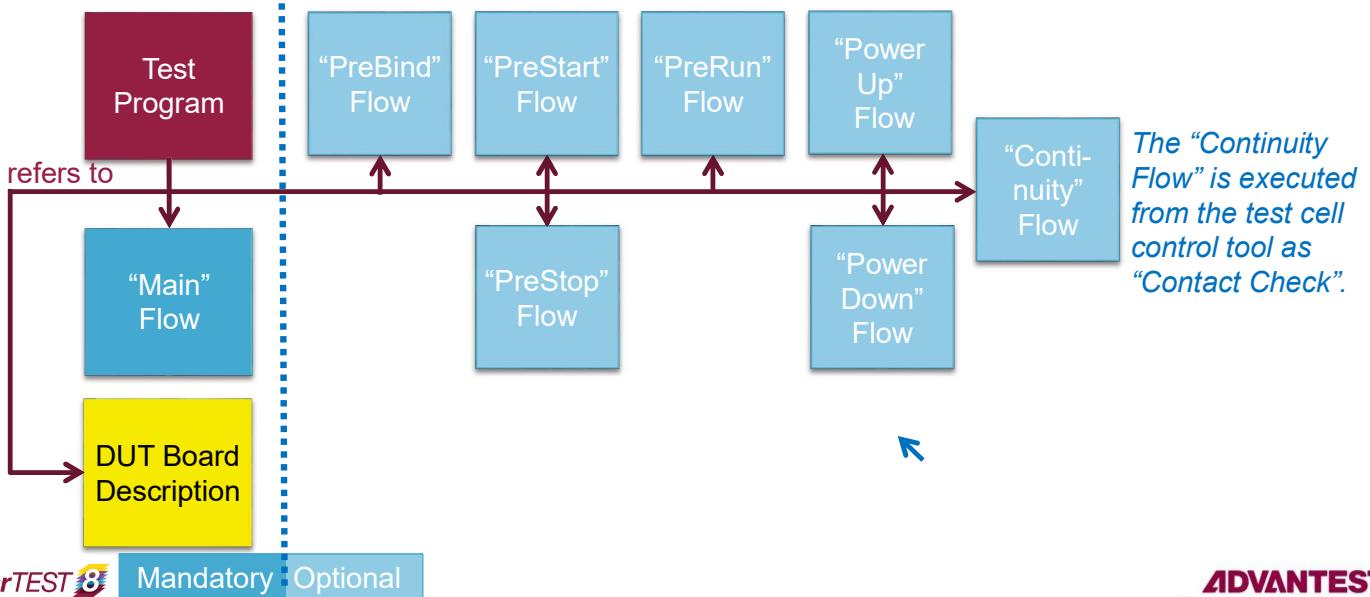
# Building Blocks: Main Flow File



# Building Blocks: Auxiliary Flow Files

The test program can also point to optional **auxiliary flows**.

Names and execution sequence of these flows are fixed.

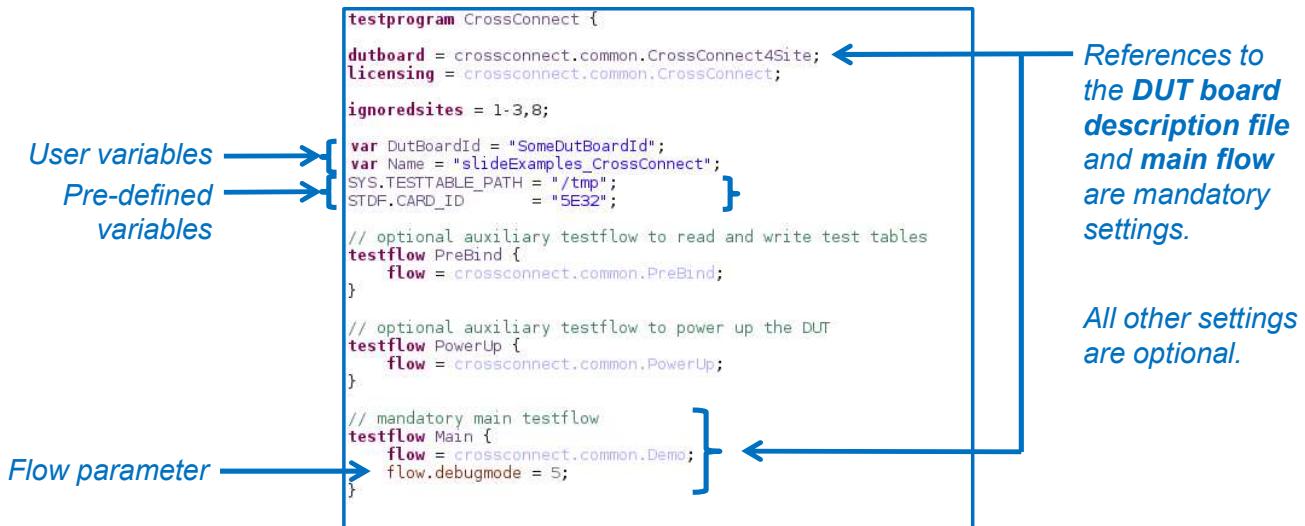


# Building Blocks: Configuration

The *test program file* allows to configure details of the test program:

- Sites which are defined in the *DUT board description file*, but which shall always be ignored during execution can be declared here
- Licensing requirements:  
The *licensing mode* SmarTest runs with and the *test program licensing file* describing the requirements.
- Variables of the test program:
  - user variables (free form);
  - system and STDF variables (set or alter pre-defined variables).
- Parameters of flows:  
The *main flow* and *auxiliary flows* can be parametrized.  
The declaration of these in the test program allows to overwrite default values of the flow parameters.

# Example of a Test Program File



The syntax is defined in the SmarTest 8 Setup Format (SSF).

## SmarTest Setup Format (SSF)

### Purpose

- Newly developed to describe setup elements and their relation based on the vocabulary of the domain / area: test program, DUT board, specification, etc.

### Benefits

- Exact fit of syntax to what needs to be described:  
→ Human readable and efficient.
- Instant check for errors.
- Content assist helps to work with the new syntax.

```
testprogram CrossConnect {
    dutboard = crossconnect.common.CrossConnect4Site;
    licensing = crossconnect.common.CrossConnect;

    var DutBoardId = "SomeDutBoardId";
    var Name = "slideExamples_CrossConnect";
    var printToConsole = false;
    var myPI       = 3.141569;
    SYS.TESTTABLE_PATH = "/tmp";
    STDF.CARD_ID   = "5E32";

    // mandatory main testflow
    testflow Main {
        flow = crossconnect.common.Demo;
        flow.debugmode = 5;
    }
}
```

# Editor Support: Content Assist “Ctrl+<Space>”

```
dutboard = crossconnect.common.CrossConnect4Site;  
licensing = crossconnect.common.CrossConnect;
```



Click on empty line:  
Show possible syntax  
elements of the edited  
file (test program file).

```
dutboard =  
licensing =
```

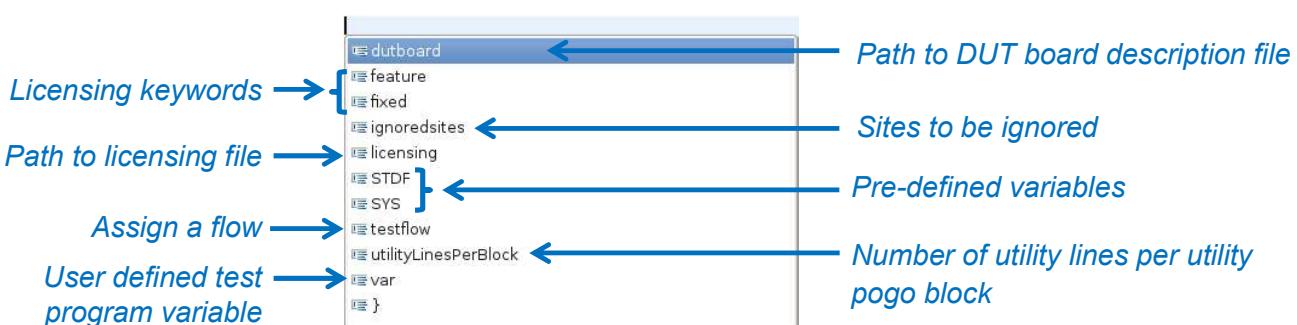


Fill-in / completion help  
whenever possible  
(suggestions on how to  
complete the statement).

Content assist makes syntax suggestions (auto-complete) and is available for the test program file.

Content assist is available as well for all other setup files based on the *SmarTest setup format* and Java test method files.

## Available Key Words in the Test Program File

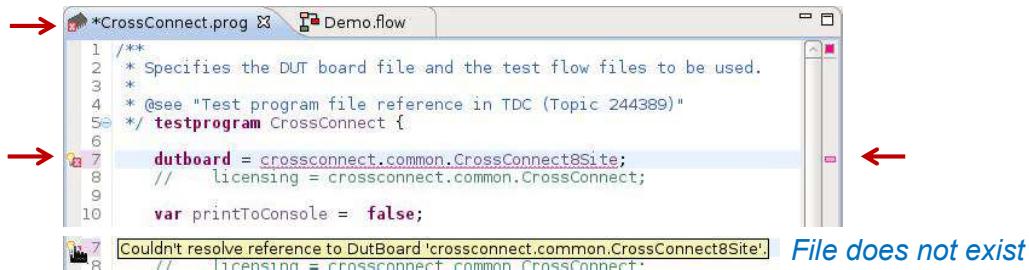


A test program must provide the reference to the *DUT board description file* and to the *main testflow*.

# Mouse Over: Error Messages and Quick Fixes

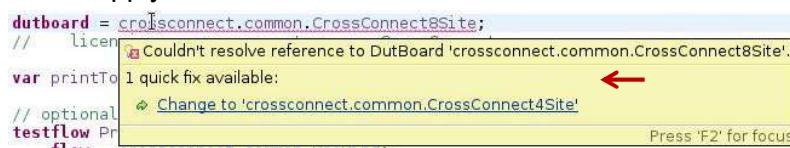
If a check is possible and it fails, the setup file is marked erroneous.

Error markers help identifying such files and resolving the issues.



Mouse over opens context help stating what is wrong

Quickfix: Holding the mouse over an erroneous entry will open a yellow box with info and a detailed suggestion. Click the link to apply the fix.



## Summary - What you should have learned

- The *test program file* is in the top position of the file hierarchy of a SmarTest 8 project and it contains or refers to all the information required to describe the test program for a specific DUT.
- It is mandatory, that any *test program file* contains a reference to a *DUT Board description file* and that it defines the *main flow* – the flow that will be executed when running the test program.
- Typically, a test program file may define *auxiliary flows* and user defined test program variables, set system variables, list sites to be ignored, set the licensing mode and refer to a licensing file.
- Test program files* are based on the *SmarTest Setup Format* (SSF) that provides instantly checking for errors and with pressing “Ctrl+<Space>” content assist is available.
- Often for errors a quick fix is suggested – for such a quickfix hold the mouse over an erroneous entry or press “Ctrl+1”.



# DUT Board Description

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved – ADVANTEST CORPORATION

**ADVANTEST**

SmarTest DUT Board Description - 1

## Learning Objective

- Understand the concept of the *DUT board description*
- Learn how to create and use *DUT board description files*



January 2020

All Rights Reserved – ADVANTEST CORPORATION

**ADVANTEST**

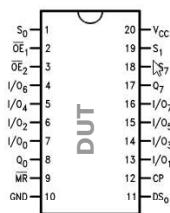
SmarTest DUT Board Description - 2

# Agenda

- Introduction of the *DUT board description*
- Properties of the *DUT board description files*
- Syntax of the *DUT board description files*
- Viewing content of *DUT board description files* in tables
- Special settings in *DUT board description files*
- Creating *DUT board description files*

## DUT Board Description Purpose

DUT data sheet excerpt



Pin Names	Description
CP	Clock Pulse Input
DS <sub>0</sub>	Serial Data Input for Right Shift
DS <sub>7</sub>	Serial Data Input for Left Shift
S <sub>0</sub> , S <sub>1</sub>	Mode Select Inputs
MR	Asynchronous Master Reset
OE <sub>1</sub> , OE <sub>2</sub>	3-STATE Output Enable Inputs
I/O <sub>0</sub> –I/O <sub>7</sub>	Parallel Data Inputs or 3-STATE Parallel Outputs
Q <sub>0</sub> , Q <sub>7</sub>	Serial Outputs

DUT board Signal list: 4 sites

	Name	DUT1	DUT2	DUT3	DUT4
1	S0	D10507	D10607	D11015	D30104
2	OE1	D30209	D30211	D30205	D30208
3	OE2	D30210	D30212	D30206	D30207
4	I/O_6	D10506	D10606	D11013	D30103
5	I/O_4	D10505	D10605	D11011	D30105
6	I/O_2	D10504	D10604	D11009	D30102

SmarTest  
DUT board description

Signal	disabled	Pogo	Site 1	Site 2	Site 3	Site 4
S0	false	10507	10607	11015	30104	
OE1	false	30209	30211	30205	30208	
OE2	false	30210	30212	30206	30207	
I/O_6	false	10506	10606	11013	30103	
I/O_4	false	10505	10605	11011	30105	
I/O_2	false	10504	10604	11009	30102	

DUT board allocation

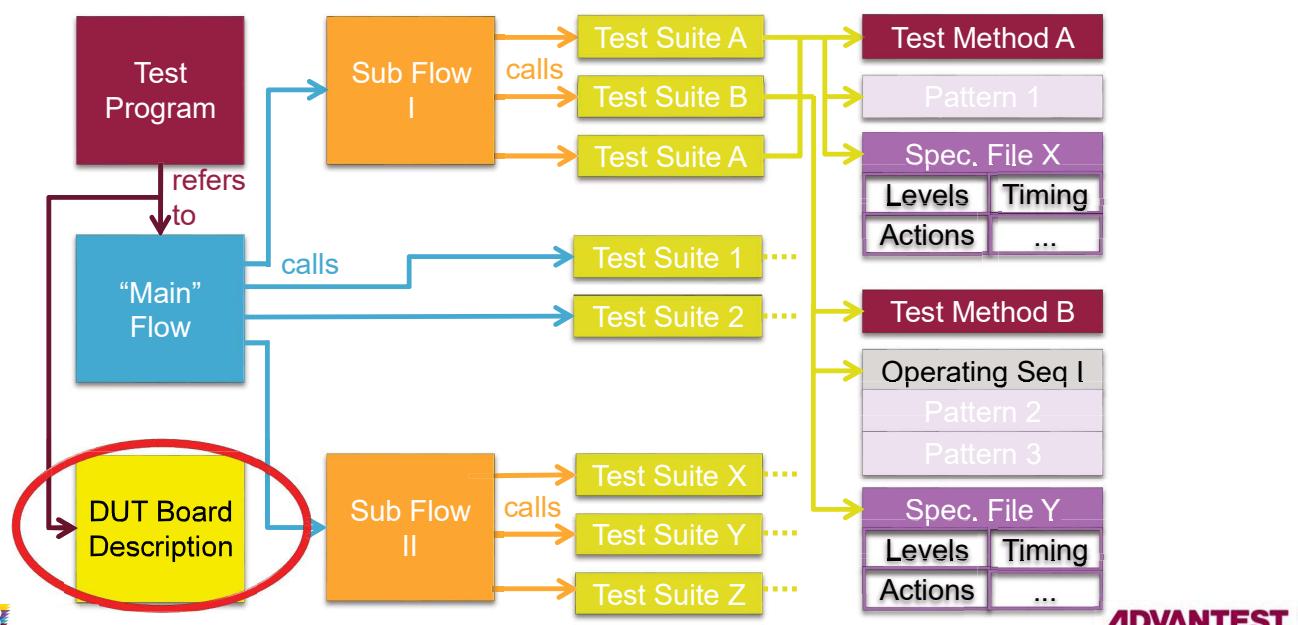


Trace on interface-PCB from Pogo-block to device pin

# DUT Board Description: Overview

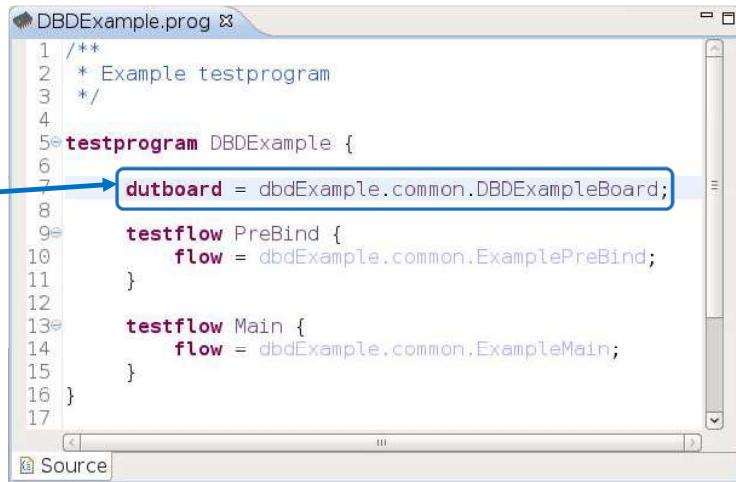
- The *DUT board description* specifies how the DUT board connects the pogo pins of the tester to the pins of the DUT (of the different sites).
- The DUT pins are described by *signals*, which are logical inputs to the DUT or outputs from the DUT.
  - At one time, only one logical *signal* can be mapped to a physical pogo pin.
  - At various times, different *signals* can be mapped to the same pogo pin.
- A *DUT board description* includes information on:
  - Number of sites configured on the board.
  - Connections between the physical pogo pins and the logical DUT signals which might include configurable routing.
  - Fixture delay data for digital traces.
  - Ganging of DPS channels.
  - Minimal and maximal voltage and current for DPS channels.
  - ...

## Test Program in SmarTest 8: Building Blocks



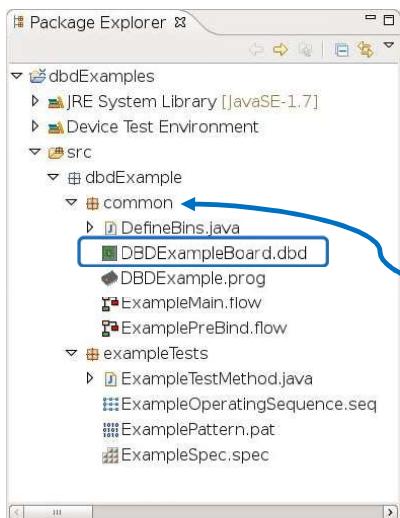
# DUT Board Description and Test Program

A *DUT board description file* is called by a test program using the keyword **dutboard**.



```
DBDExample.prog
1 /**
2 * Example testprogram
3 */
4
5 testprogram DBDEExample {
6     dutboard = dbdExample.common.DBDEExampleBoard;
7
8     testflow PreBind {
9         flow = dbdExample.common.ExamplePreBind;
10    }
11
12    testflow Main {
13        flow = dbdExample.common.ExampleMain;
14    }
15
16 }
17
```

## Properties of the DUT Board Description



- Only one per test program.
- All *DUT board description files* of loaded projects must declare the same set of signals. Use `@Disabled` for differences.
- The file should be located in the “common” folder. See “Recommendations for the Structure of Setup Files” for details.

# Example: Syntax of the DUT Board Description

```
DBDEExample.prog DBDEExampleBoard.dbd
1 dutboard DBDEExampleBoard {
2     sites = 4;
3
4     signal S0 {
5         site 1 { pogo = 10507; }
6         site 2 { pogo = 10607; }
7         site 3 { pogo = 11015; }
8         site 4 { pogo = 30104; }
9     }
10 }
```

“dutboard”, “sites”, “signal”, “site” and “pogo” are keywords of the syntax.

The signal “S0” is a logical input to the DUT or a logical output from the DUT.

Name of the DUT board description and its file name must match

The total number of sites

Signal name

The syntax for assigning a pogo is the same for the different types of channels:

- digital,
- analog,
- RF,
- power supplies.

## Views and Tabs of the DUT Board Description

```
DBDEExampleBoard.dbd
1 dutboard DBDEExampleBoard {
2     sites = 4;
3
4     signal S0 {
5         site 1 { pogo = 10507; }
6         site 2 { pogo = 10607; }
7         site 3 { pogo = 11015; }
8         site 4 { pogo = 30104; }
9     }
10
11     [Disabled]
12     signal OE1 {
13         site 1 { pogo = 30209; }
14         site 2 { pogo = 30211; }
15         site 3 { pogo = 30205; }
16         site 4 { pogo = 30208; }
17     }
18
19     signal OE2 {
20         site 1 { pogo = 30210; }
21         site 2 { pogo = 30212; }
22         site 3 { pogo = 30206; }
23         site 4 { pogo = 30207; }
24 }
```

Use the tabs of the SmarTest8 editor for *DUT board description files*

- to work on the source code (shown);
- to view and edit the settings in the tables for
  - signals (next slide),
  - utility lines,
  - properties,
  - routing.

Use these tabs to switch

# Signals Tab versus Source Tab

The screenshot shows two windows side-by-side. The left window is titled 'DBDEExample.prg' and contains a table with columns: Signal, disabled, Pogo, Site 1, Site 2, Site 3, Site 4. The right window is titled 'DBDEExampleBoard.dbd' and contains source code for a 'dutboard' board. Blue arrows point from specific cells in the table to the corresponding lines of code in the source tab, illustrating how changes made in one tab are reflected in the other.

Signal	disabled	Pogo	Site 1	Site 2	Site 3	Site 4
S0	false	10507	10607	11015	30104	
OE1	true	30209	30211	30205	30208	
OE2	false	30210	30212	30206	30207	
IO_6	false	10506	10606	11013	30103	
IO_4	false	10505	10605	11011	30105	
IO_2	false	10504	10604	11009	30102	
IO_0	false	10503	10603	11007	30101	
IO_7	false	10510	10610	11005	30107	
IO_5	false	10509	10609	11003	30108	
IO_3	false	10508	10608	11002	30109	
IO_1	false	10500	10600	11001	30100	

```

1 dutboard_DBDEExampleBoard {
2   sites = 4;
3
4   signal S0 {
5     site 1 { pogo = 10507; }
6     site 2 { pogo = 10607; }
7     site 3 { pogo = 11015; }
8     site 4 { pogo = 30104; }
9   }
10
11   @Disabled
12   signal OE1 {
13     site 1 { pogo = 30209; }
14     site 2 { pogo = 30211; }
15     site 3 { pogo = 30205; }
16     site 4 { pogo = 30208; }
17   }
18
19   signal OE2 {
20     site 1 { pogo = 30210; }
21     site 2 { pogo = 30212; }
22     site 3 { pogo = 30206; }
23     site 4 { pogo = 30207; }
}

```

Users can choose their preferred way for editing.

Smartest automatically aligns the contents of tables and of “Source”.

## DUT Board File with Fixture Delays

Fixture delay is the time it takes for an electrical signal to propagate on the DUT board trace from a pogo pin and to a DUT pin of a specific site.

The screenshot shows two windows. The left window is the 'Source' tab of 'DBDEExample.prg' and contains source code for a signal IO\_0 with fixture delays for four sites. The right window is the 'Properties' tab of 'DBDEExampleBoard.dbd' and contains a table for fixture delays across six sites. A blue arrow points from the 'Properties' tab table to the source code, and another arrow points from the source code to the 'Properties' tab table, illustrating the bidirectional relationship between the two.

```

46
47   signal IO_0 {
48     site 2 { pogo = 10603; fixtureDelay = 1 ns; }
49     site 1 { pogo = 10503; fixtureDelay = 1e-9 s; }
50     site 3 { pogo = 11007; fixtureDelay = 0.001 us; }
51     site 4 { pogo = 30101; fixtureDelay = 1000 ps; }
52   }
53

```

fixtureDelay			iforceMax		
Site 1	Site 2	Site 3	Site 4	Site 1	Site 2
1e-9 s	1 ns	0.001 us	1000 ps		

*Limits on voltage and current help to protect needles and socket pins*

*Use the “Properties” tab to edit fixture delays and other properties*

# Device Power Supply

DPS channels are setup the same way as any other types of channels.

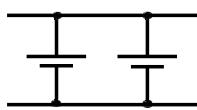
The screenshot shows two windows from the SmarTEST software interface. The top window is titled 'DBDExample.prg' and displays a portion of a DBD script with code for defining a 'VCC' signal with four sites, each having a specific pogo pin number. The bottom window is also titled 'DBDExample.prg' and shows a configuration dialog for 'Number of Sites' set to 4. It includes a table where the 'VCC' signal is mapped to four sites, with its state set to 'disabled' and its Pogo pins listed as 34201, 34205, 34209, and 34313 respectively.

## Ganging DPS Channels

To achieve a higher output current, you can operate multiple channels of DPS cards in parallel, known as ganging.

Ganging is the combination of multiple DPS channels to provide a higher current than a single DPS channel could provide.

Syntax:



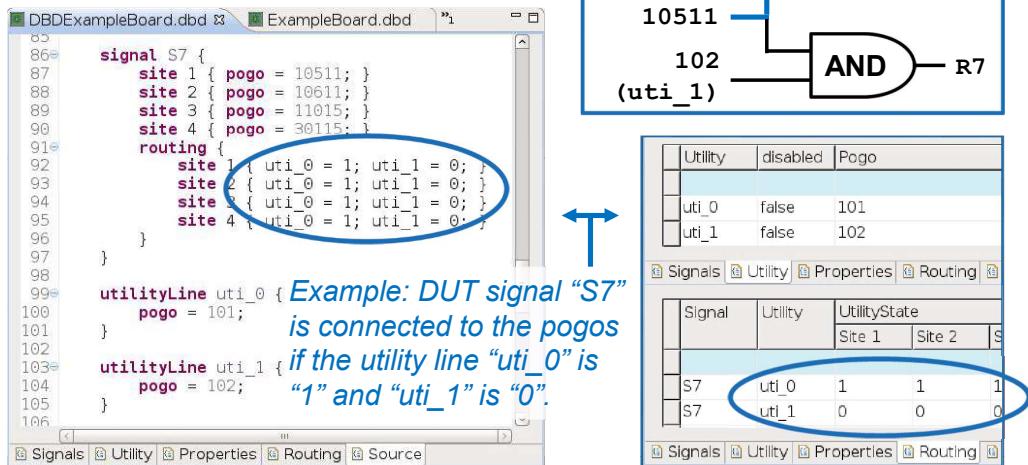
The screenshot shows a window titled 'DBDExample.prg' displaying a DBD script. The code defines a 'VCC' signal with four sites, where each site's pogo pin number is split into two parts: 34201, 34202, 34205, 34206, 34209, 34210, 34313, and 34214. This indicates that the four DPS channels are being ganged together in parallel.

Note:

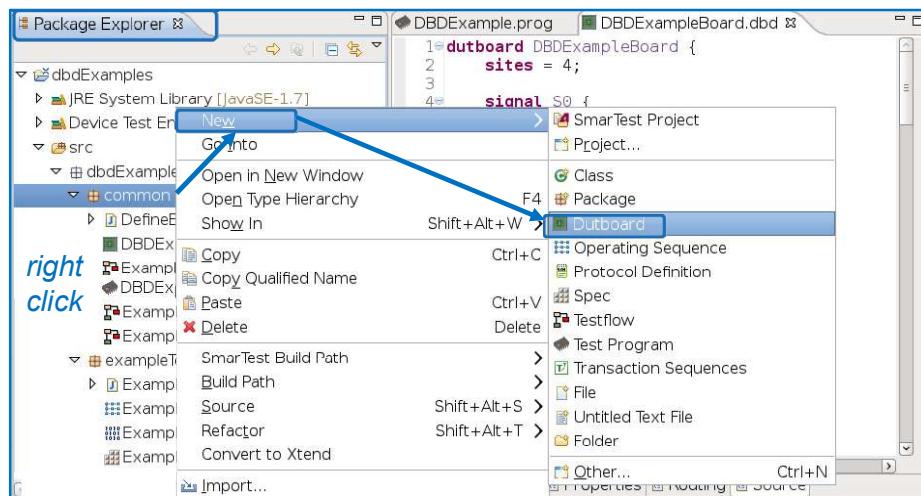
The channels of each ganged group of DPS channels have to be in sequence, for example 34311, 34312, 34313, 34314.

# DUT Boards with Switches

Often DUT boards make use of switches, for example to configure the routing of a single pogo to multiple DUT pins.



## Creating a New DUT Board Description File



# Summary - What you should have learned

- The DUT board description file specifies how the DUT board connects the pogo pins of the tester to pins of the DUT.
- The DUT pins are described by signals, which are logical inputs or outputs to the DUT.
  - *A one time, only one logical signal can be mapped to a physical pogo pin.*
  - *At various time, different signals can be mapped to the same pogo pin.*
- A DUT board description file includes information on:
  - Number of sites configured on the board.
  - Connections between the physical pogo pins and the logical DUT signals.
  - Fixture delay data for digital traces.
  - Ganging of DPS channels.
  - Voltage and current ranges for DPS channels.

## Backup

# RF Pins in DUT Board Description

RF pins are defined exactly the same way as other pins.

Example:

RF DUT pins to Wave Scale RF ports in the DUT board description file.

Number of Sites: 2

Select a cell in the table below

	Signal	disabled	Pogo	
			Site 1	Site 2
	rfIn1	false	90101	90109
	rfOut1	false	90208	90114

```
1 dutboard dutboard1 {
2
3     sites = 2;
4
5     signal rfIn1 {
6         site 1 { pogo = 90101; }
7         site 2 { pogo = 90109; }
8     }
9
10    signal rfOut1 {
11        site 1 { pogo = 90208; }
12        site 2 { pogo = 90114; }
13    }
14 }
15 }
```

# Deembedding in DUT Board Description

Deembedding is used to compensate for RF DUT board traces.

The optional data is stored in dedicated “\*.s2p” files (Touchstone format) or “\*.citi” files (Common Instruction Transfer and Interchange files).

Number of Sites: 2

Select a cell in the table below

	Signal	deembedding	
		Site 1	Site 2
	rfIn1	"deembedding/Gain_1.citi"	"deembedding/Gain_2.citi"
	rfOut1		

Signals Utility Properties Routing Source

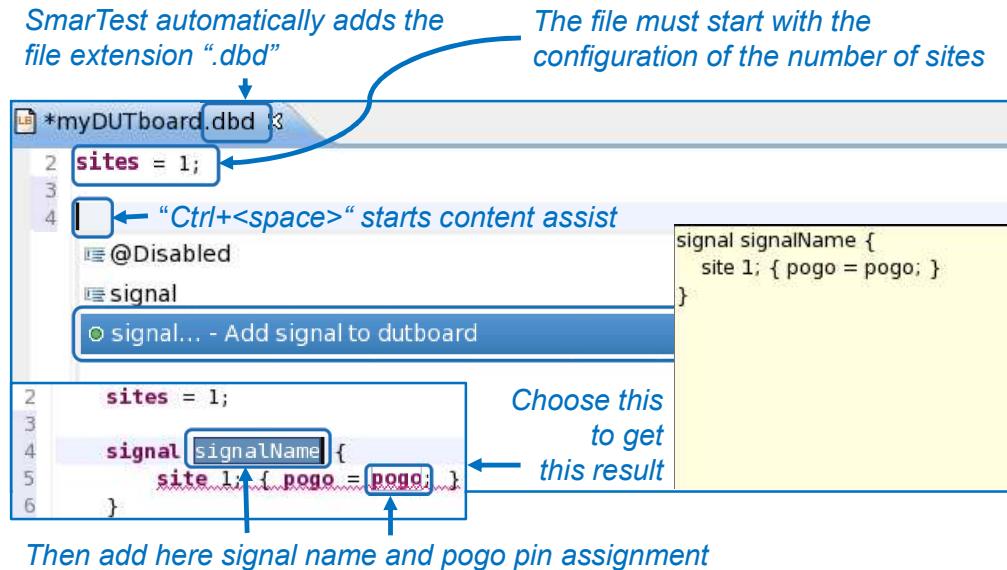
```
1 dutboard dutboard1 {
2
3     sites = 2;
4
5
6     signal rfIn1 {
7         site 1 { pogo = 90101; deembedding= "deembedding/Gain_1.citi"; }
8         site 2 { pogo = 90109; deembedding= "deembedding/Gain_2.citi"; }
9     }
10
11    signal rfOut1 {
12        site 1 { pogo = 90208; }
13        site 2 { pogo = 90114; }
14    }
15 }
```

The *property element* allows to define multiple deembedding settings, which can be distinguished by *tags*.

Such a *tag* can be referenced by a *specification file*, for example, when setting up an *action “modulated”* of the “rfSim” *instrument*.

For details see the topic “DUT board file reference” in the TDC.

# Editing Source Code Using Content Assist



## Building Blocks of Test Programs

SmarTest 8.2.5 Training

January 2020

# Learning Objective

- Get an overview over the building blocks of a test program:
  - Testflow files
  - Test methods
  - *Specification files*
  - Pattern files
  - *Operation sequence files*
- Learn how to activate and execute a test program

# Agenda

- Details of the building blocks of a test program
- Activating and executing a test program

# Building Blocks: Test Program File and its Content



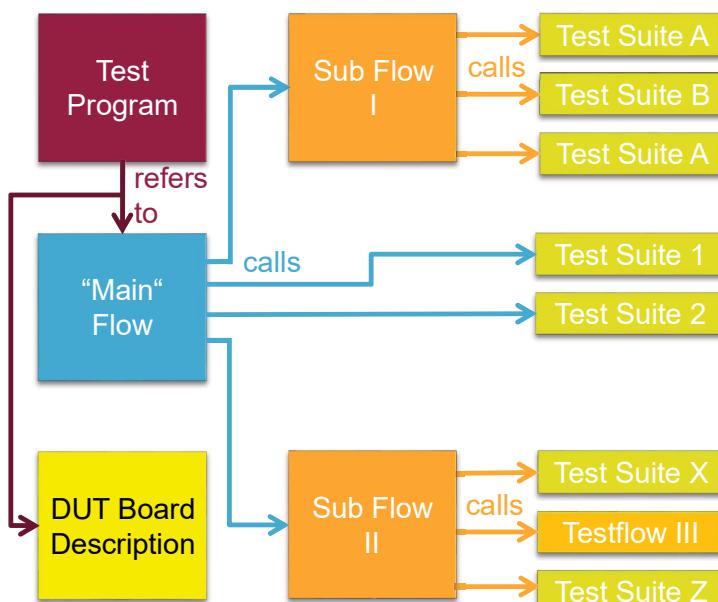
The *test program file* must contain references to the *main flow* and to the *DUT board description file*.

The *main flow*

- contains all test content for a specific DUT;
- typically calls subflows;
- uses the same syntax as any other *testflow file*.

The *DUT board description file* specifies the properties of the interface between the device under test (DUT) and the V93000 tester, which is the DUT board.

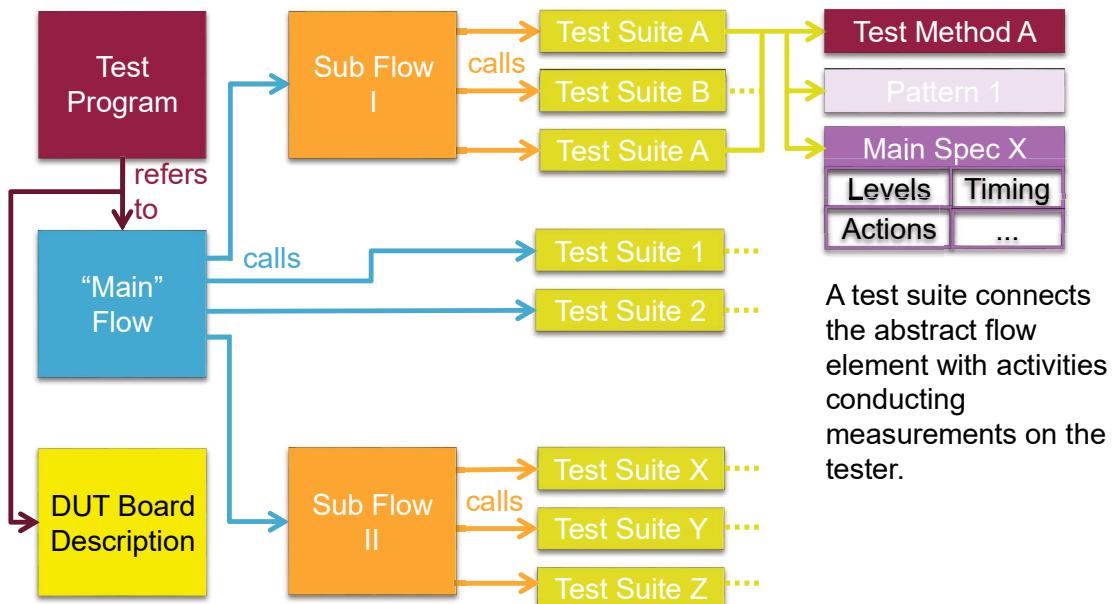
# Building Blocks: Testflow and Test Suites



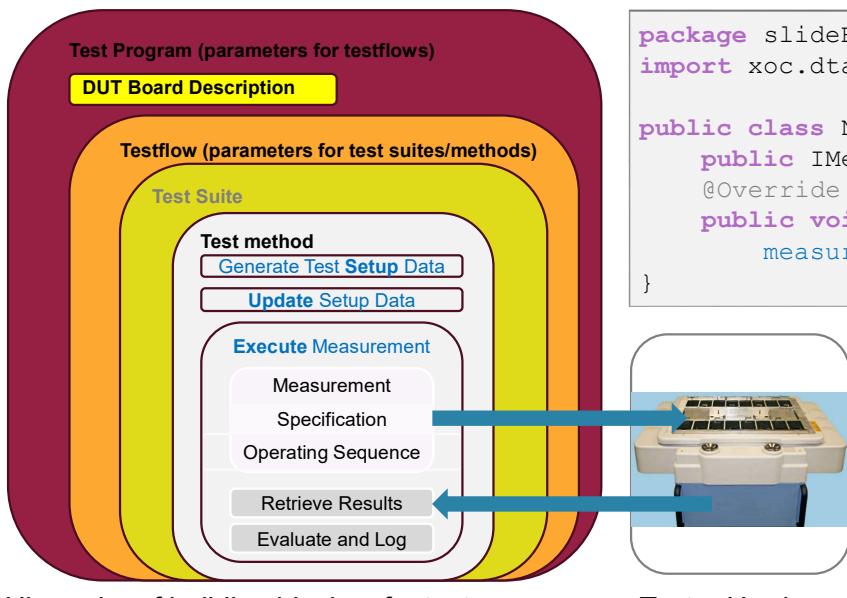
Testflows are hierarchical:  
An element of a testflow  
can call another testflow  
or a test suite.

- The same test suite can be executed multiple times within the test flow that calls it.
- The same sub flow can be called multiple times.
- A testflow as well as a test suite can have input and output parameters.

# Building Blocks: Architecture of Test Suites



## Building Blocks: From Test Program to measurement



Hierarchy of building blocks of a test program

```
package slideExamples.testMethods;
import xoc.dta.TestMethod;

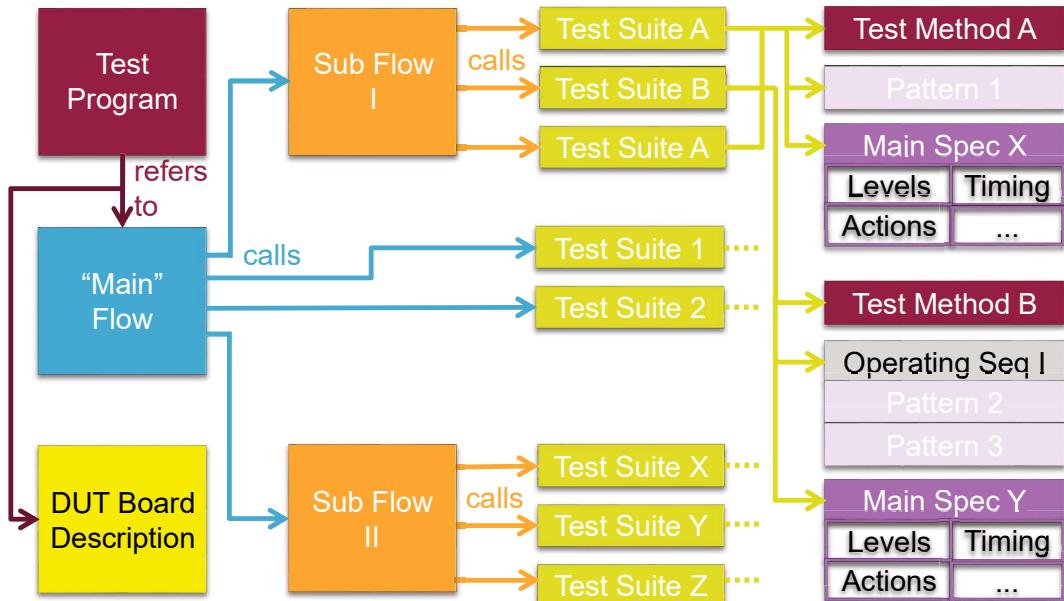
public class NewTestMethod extends TestMethod {
    public IMeasurement measurement;
    @Override
    public void execute() {
        measurement.execute();
    }
}
```



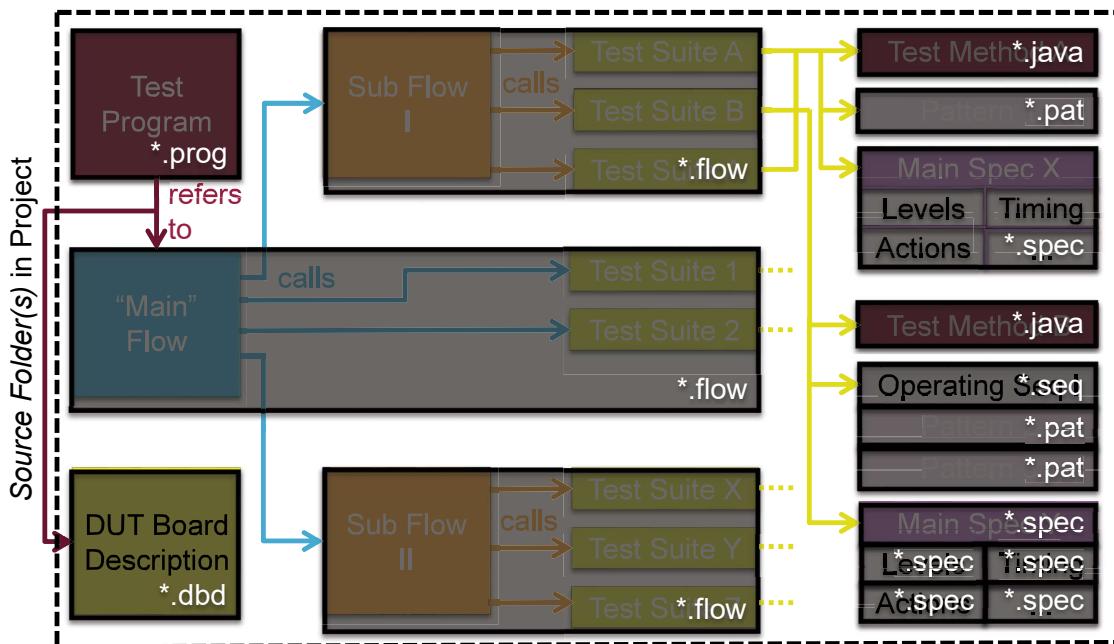
Tester Hardware

- Accessing the Tester Hardware:
- *Measurement* objects in the test method classes are the one and only execution engines for tests on the tester

# Building Blocks: Summary

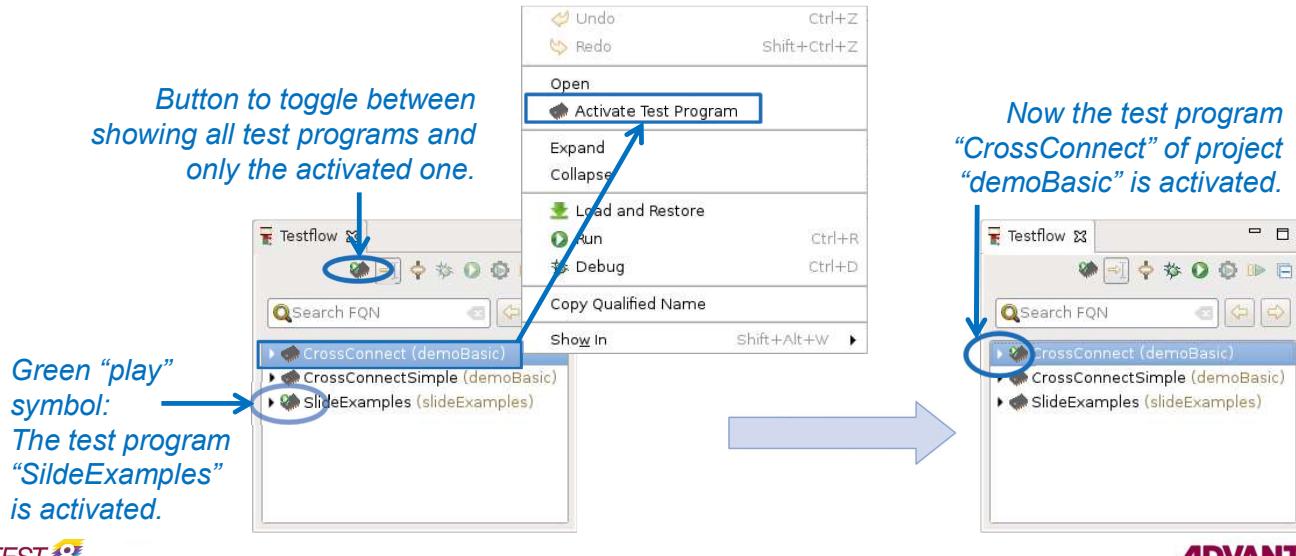


# Building Blocks: File Types and File Locations



# How to Activate a Test Program

A test program can be activated with the *testflow* view which is also used to execute and debug test programs, testflows and test suites.



## Activate Test Program - Purpose

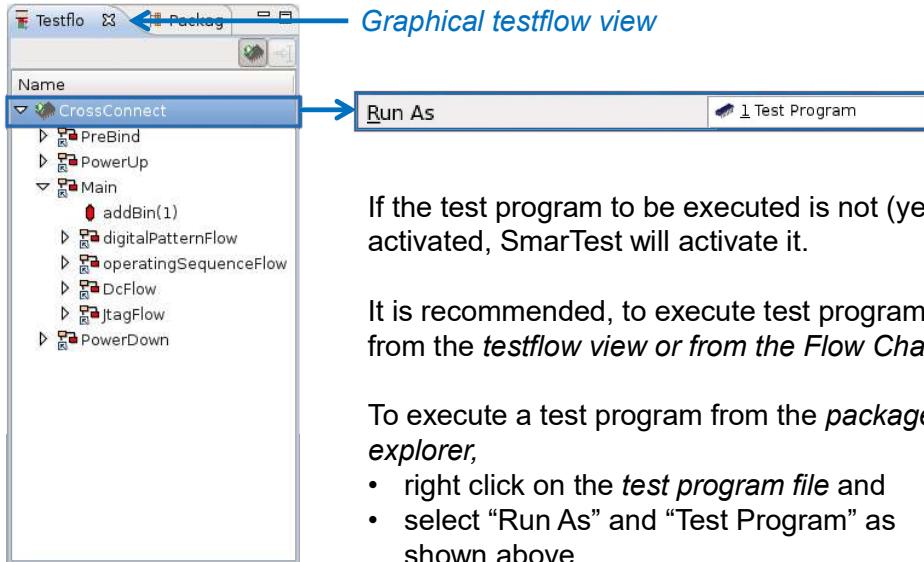
- Read selected *test program file*.
- Read *DUT board description file* defined in the test program.
- Activate sites according to *DUT board description*.
- Read licensing file and check needed licenses.
- Reset settings in the card channels of the test head.

An activated test program is needed for:

- Executing a test program or a testflow.
- Viewing of a pattern in the *pattern debug editor*.

In order to propagate modifications of the test program file or the referred *DUT board description file*, the *test program file* must be activated, even if it is already activated.

# Executing a Test Program



If the test program to be executed is not (yet) activated, SmarTest will activate it.

It is recommended, to execute test programs from the *testflow view or from the Flow Chart*.

To execute a test program from the *package explorer*,

- right click on the *test program file* and
- select “Run As” and “Test Program” as shown above.

## Summary - What you should have learned

- Building blocks of a test program and the related files:
  - The file on the top of a test program is the *test program file*, its name has the suffix “.prog”.
  - A *DUT board description file* (suffix “.dbd”) must be referenced from the *test program file*.
  - In the *test program file* must be specified a *main flow* and optionally *auxiliary flows*.  
These and all called subflows are defined in testflow files which have names with the suffix “.flow”.
  - Test suites are not defined in dedicated files. The definition of a test suite is part of a testflow file.
  - A test suites must call a test method which is defined in dedicated Java file with suffix “.java”.  
Typically, a test suite definition references
    - instrument setups which are defined in one or multiple *specification files* (suffix “.spec”);
    - patterns which are stored in dedicated pattern files that have names with the suffix “.pat”
    - or *operating sequence* which have files with the suffix “.seq” in their names
- In the *testflow view*, a right click on a test program brings up a menu, that allows to activate the test program and to execute it or to run it in debug mode.

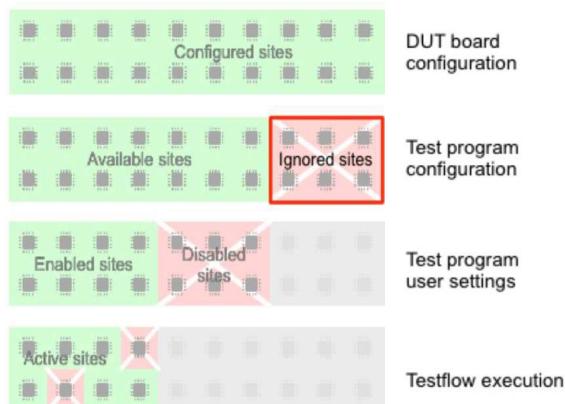
# Backup

## Setting “ignoredsites” in the Test Program File

The DUT board description file might configure more sites than wanted. For example, the tester hardware does not support all configured sites.

Then “ignoredsites” entry can be used to define “ignored sites”.

The configured sites minus the ignored sites are called “available sites”.





# Testflows and Test Suites

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

SmarTest Testflows and Test Suites - 1

## Learning Objective

- Understand the relationship of testflows, test suites and measurements.
- Learn about the role and the content of testflow files.



January 2020

All Rights Reserved - ADVANTEST CORPORATION

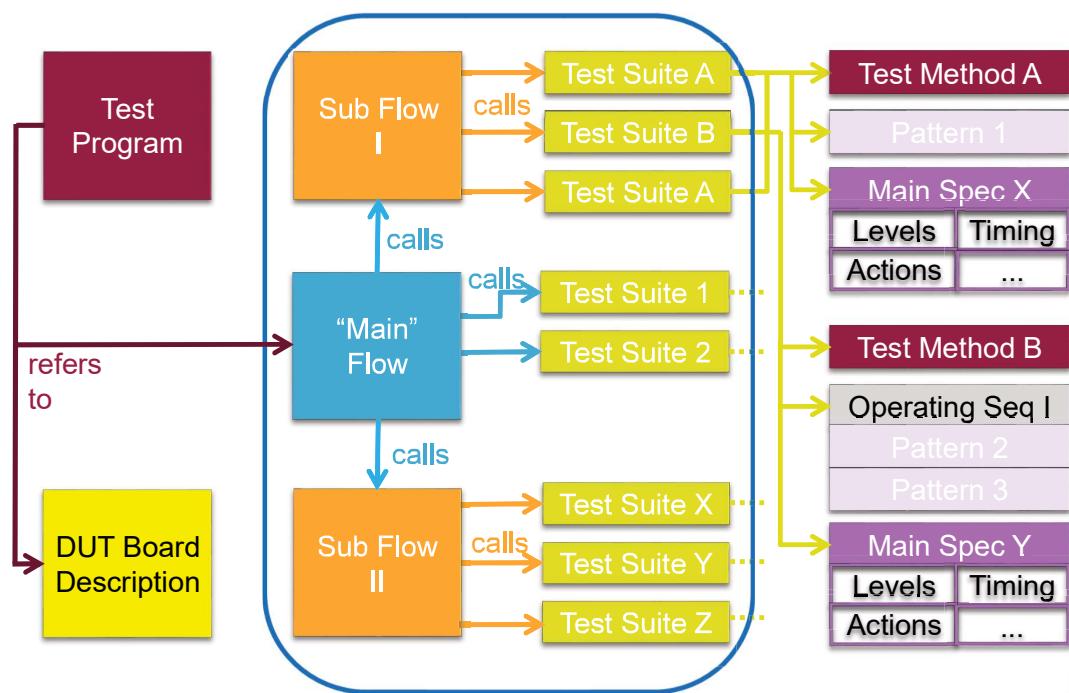
**ADVANTEST**

SmarTest Testflows and Test Suites - 2

# Agenda

- Purpose and content of testflow files
- Setup of test suites and testflows
- Execution sequence of test suites and testflows
- Test program activation and execution
- Graphical *testflow view*
- *Flow Chart view*

## Building Blocks: Testflows and Test Suites



# Testflow Files: Overview

Testflow files define the test suites and other (sub) flows, that can be executed, and the sequence of their execution.

Optional input parameters can be used to parametrize testflow files.

Results can be returned via output parameters.

A testflow file contains two parts:

1. "setup": Defines test suites and sub flows.
2. "execute": Lists, which of the test suites and the sub flows defined in the setup part are executed in what sequence.

The execution flow can be controlled by conditional commands and by loops. The execution can be completely stopped.

Note:

The execution sequence of test suites or testflows is not related to the order of the definitions in the setup part.

## Basics of Test Suites

Test suites are the elements of the testflow that trigger the tester to run *measurements* on devices.

Test suites call test methods that use the programming language Java to implement (some of) the following steps:

- Definition of one or more *measurement* setups of the tester.
- One or more physical executions of the *measurements* that are set up.
- Collection and optionally post processing of raw test results.
- Pass/fail judgements based on the (post-processed) results.
- Data logging.

Input parameters of test methods allow parameterization typically required for calling one test method by multiple test suites.

Note:

Test suites are part of a testflow file – no dedicated test suite files exist.

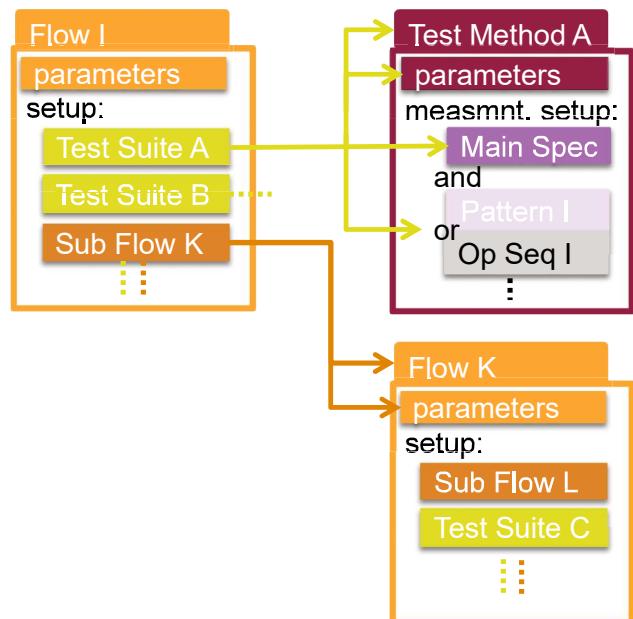
# Setup Part in Testflow Files

The definition of a test suite

- is done in the setup part of a testflow file;
- always calls a test method;
- typically sets values for input parameters of the called test method;
- typically sets *specification files*, patterns or *operating sequences* of the *measurements* of the test method.

The definition of a subflow

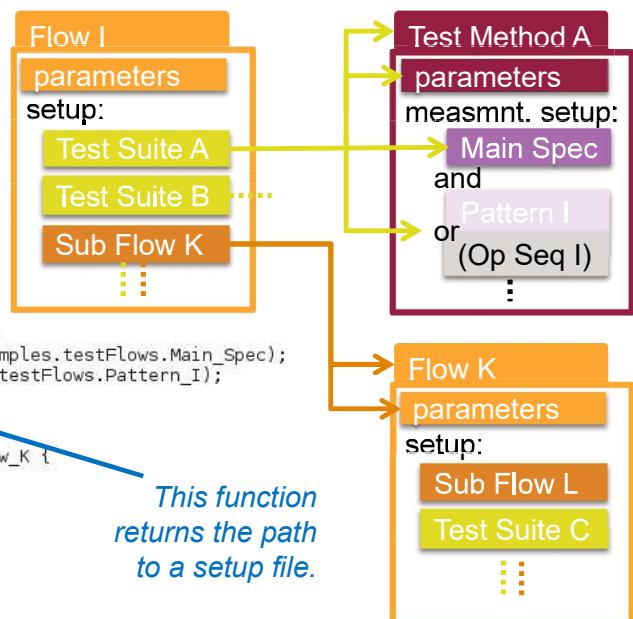
- is done in the setup part of a testflow file;
- always calls another testflow file;
- might set values for input parameters.



## Setup Part in Testflow Files – Source Code

Editors allow to collapse lines as shown here in the testflow file

```
Flow_I.flow
1 flow Flow_I {
2     in flow_I_Parameter = 0;
3     setup {
4         // test suite definitions
5         suite Test_Suite_A calls Test_Method_A {
6             testSignals = "ROI"; // test method parameter
7             measurement.specification = setupRef(slideExamples.testFlows.Main_Spec);
8             measurement.pattern = setupRef(slideExamples.testFlows.Pattern_I);
9         }
10        suite Test_Suite_B calls Test_Method_B {
11            // test flow definitions
12            flow Sub_Flow_K calls slideExamples.testFlows.Flow_K {
13                flow_K_Parameter = 1; // testflow parameter
14            }
15        }
16    execute {}
17}
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
```



# Execute Part in Testflow Files And Testflow View

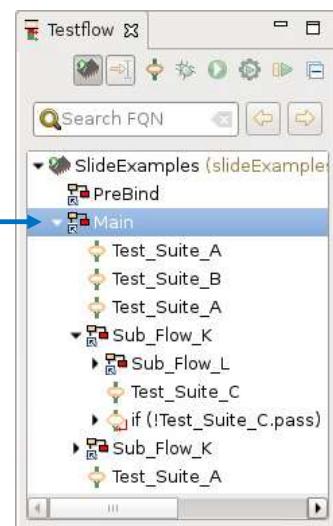
The execution part defines, which of the test suites and the sub flows, that are defined in the setup part, are executed in which sequence.

```

1 flow Flow_I {
2     in Flow_I_Parameter = 0;
3     setup {}
4
5     execute {
6         Test_Suite_A.execute();
7         Test_Suite_B.execute();
8         Test_Suite_A.execute();
9         Sub_Flow_K.execute();
10        Sub_Flow_K.execute();
11        Test_Suite_A.execute();
12    }
13}
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

```

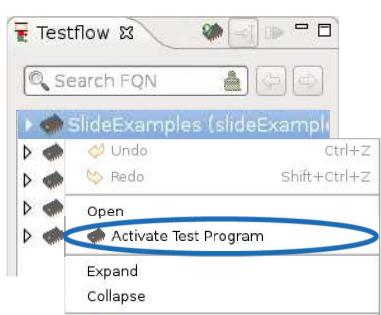
*"Flow\_I" is the main flow of test program "SlideExamples"*



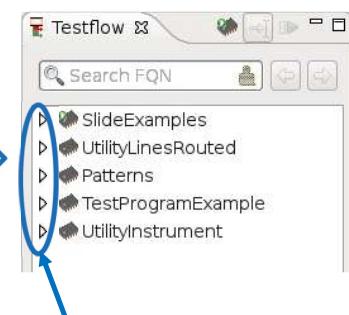
The *testflow view* shows graphically in which sequence testflows and test suites are executed.

## Testflow View – Test Program Activation

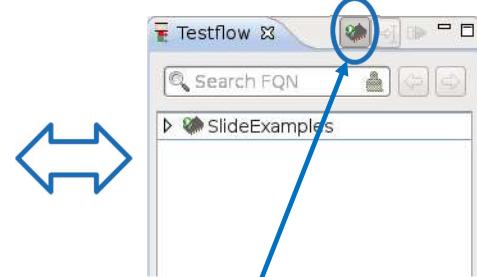
The scope switch of the *test flow view* allows to toggle between the activated test program only and all test programs of imported SmarTest projects.



Use right-click and then "Activate Test Program" to activate a test program



Buttons to expand - here to show the contents of the test programs



Button to switch the scope between displaying all test programs and only showing the activated test program

# Activate Test Program - Purpose

- Read selected test program file.
- Read DUT board description file defined in the test program.
- Activate sites according to DUT board description.
- Read licensing file and check needed licenses.
- Reset settings in the card channels of the test head.

An activated test program is needed for:

- Executing a test program or a testflow.
- Viewing of a pattern in the pattern debug editor.

In order to propagate modifications of the test program file or the referred DUT board description file, the test program file must be activated again, even if it is already activated.

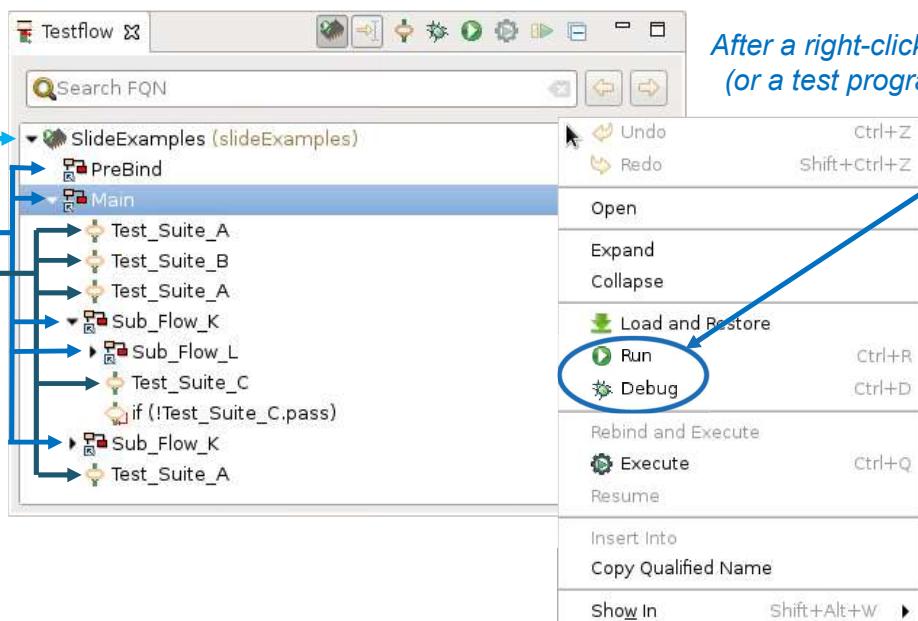
## Testflow View - Details

The graphical testflow view displays:

- test programs,
- testflows,
- test suites.

The scope is:

- The activated test program
- or all test programs.

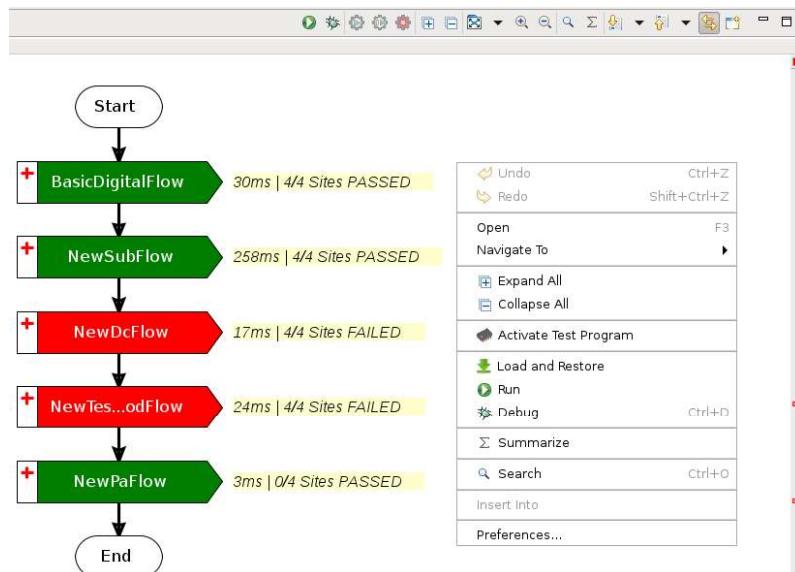


After a right-click on a testflow (or a test program) this menu pops up that allows to run or debug it.

# Flow Chart View

The graphical Flow Chart view displays:

- test programs,
- testflows,
- test suites
- Control elements.



## Summary - What you should have learned

- The one testflow called from the test program is the *main flow*.
- Testflow files are parametrizable and consist of a setup and an execute part:
  - In the setup part subflows and test suites are defined.
  - In the execute part is specified, which previously defined subflows and test suites are executed.
- The definition of a test suite typically consists
  - of a test method call, that uses a *measurement* object to run tests,
  - of an assignment of a specification file and an operating sequence to this *measurement* object,
  - of value assignments to input parameters of the test method.
- The testflow view graphically visualizes the execution hierarchy of a test program.
- The testflow view is the recommended tool
  - to activate a test program;
  - to trigger execution of a test program or subflow;
  - to start debugging of a test program or subflow.
  - Flow Chart view to display, execute and debug testflows and test suites



# Operating Sequence

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

SmarTest Operating Sequence - 1

## Learning Objectives

- Understanding how *operating sequences* are used to aligning order and time of execution for a test consisting of multiple elements like
  - digital patterns;
  - protocol operations;
  - *actions* (i.e. commands sent to instruments during test execution);
  - waiting times.
- Know the graphical tools to debug *operating sequences*.



January 2020

All Rights Reserved - ADVANTEST CORPORATION

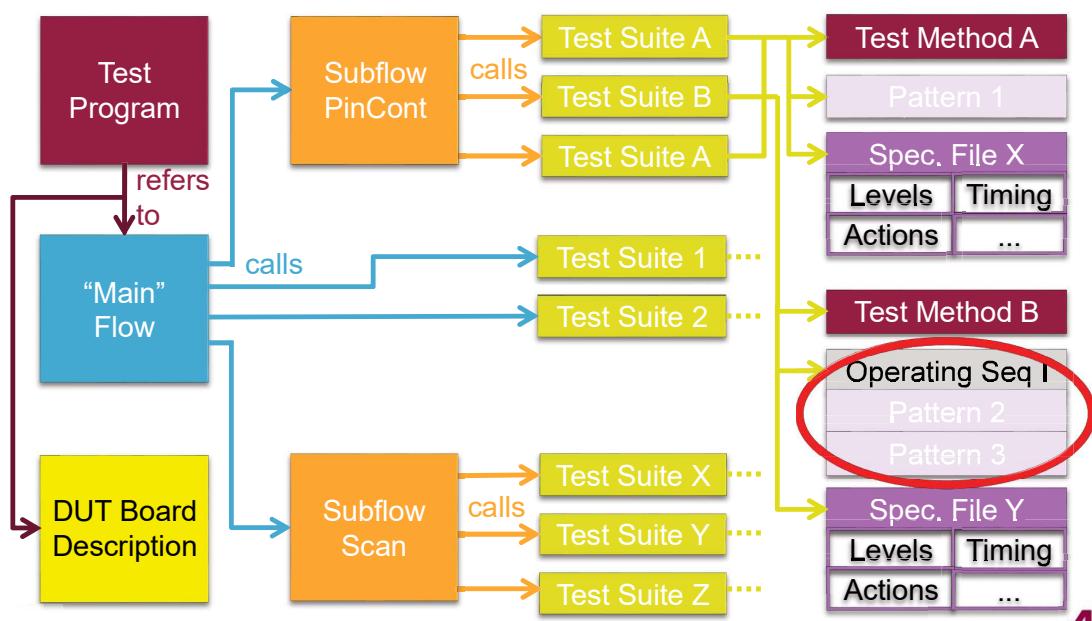
**ADVANTEST**

SmarTest Operating Sequence - 2

# Agenda

- Purpose and content of *operating sequences*
- Examples of *operating sequences*
- Setup of new *operating sequences*
- Debug of *operating sequences*

## Test Program in SmarTest 8: Building Blocks



# Purpose of Operating Sequences

When a test program is executed,

- one or several testflows are called;
- the testflows define the order of execution of various test suites;
- the executed test suites usually perform *measurements*;
- the *measurements* usually run digital patterns, protocols and/or *actions* - these must be aligned.

An **operating sequence** is a fixed arrangement of calls of one or more patterns, *actions*, and protocol *transaction sequences* – across *instrument domains*.

It enables **synchronized parallel execution** on **instruments of different domains** (digital, DC, mixed-Signal, RF and protocol aware).

```
ExecutionListExample.seq
1 sequence ExecutionListExample {
2   sequential seq1 {
3     patternCall patterns.PatternA0 FirstPattern;
4     patternCall patterns.PatternB0 SecondPattern;
5   }
6 }
```

Example of an operating sequence setup

Although an operating sequence might contain multiple patterns, *actions* and *transactions sequences*, the execution of these typically triggers only a single sequencer start per signal.

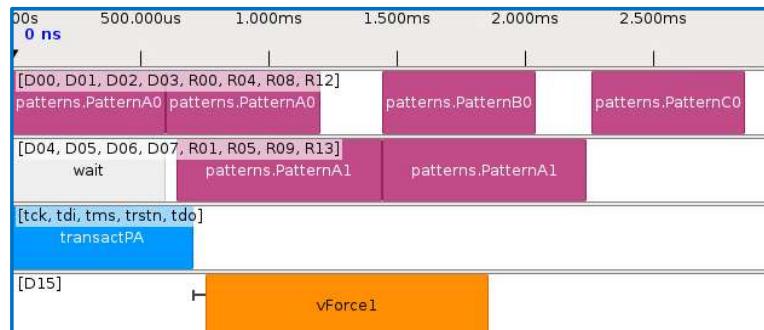
## Building Blocks of Operating Sequences

*Operating sequences* arrange the execution of the following elements:

- Digital patterns.
- *Actions* of all test domains.
- Transaction sequences of protocols.
- Delays to postpone the next element.

These elements are arranged within two different types of building blocks:

**Sequential groups** and **parallel groups**.



Visualization of an operating sequence for debug

# Operating Sequence: Sequential Groups

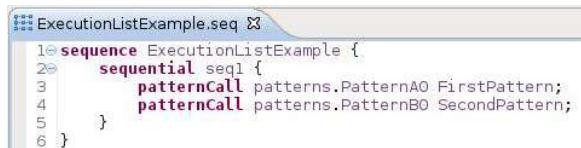
A *sequential group* can consist of patterns, *actions*, *transaction sequences*, delays and *parallel groups*. The elements of a *sequential group* are executed one after the other.

Example of *sequential group* “`seq1`”:

Two patterns are sequentially executed in a “burst”.

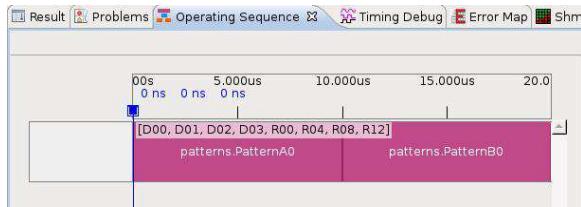
Note:

The sequencers of the tester channels used in the two patterns are started only once.



```
1 sequence ExecutionListExample {
2     sequential seq1 {
3         patternCall patterns.PatternA0 FirstPattern;
4         patternCall patterns.PatternB0 SecondPattern;
5     }
6 }
```

Setup: Operating sequence (file)



# Operating Sequence: Parallel Group

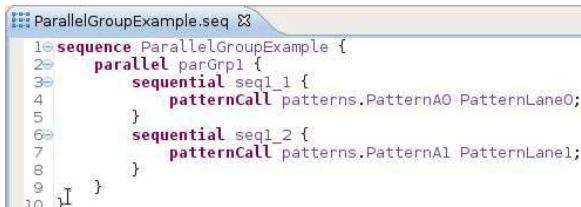
A *parallel group* consists of *sequential groups* that are executed concurrently.

*Sequential groups* of a *parallel group* must have no resource overlap.

Typically, concurrent execution means starting within a time interval of maximal 1 ns. This can be tightened or relaxed.

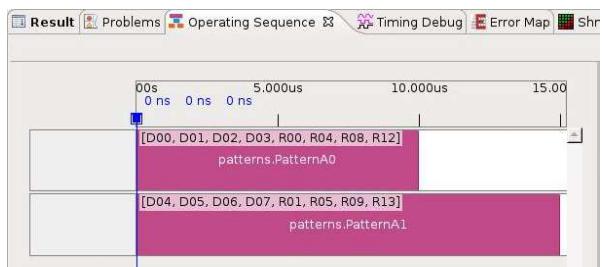
Example of *parallel group* “`parGrp1`”:

Two patterns are executed concurrently.



```
1 sequence ParallelGroupExample {
2     parallel parGrp1 {
3         sequential seq1_1 {
4             patternCall patterns.PatternA0 PatternLane0;
5         }
6         sequential seq1_2 {
7             patternCall patterns.PatternA1 PatternLane1;
8         }
9     }
10 }
```

Setup: Operating sequence (file)



# Operating Sequence: Nesting Elements

Parallel groups and sequential groups can be nested inside each other:

- A parallel group contains sequential groups – but it does not contain directly another parallel group.
- A sequential group can contain parallel groups – but it does not contain directly another sequential group.

This nesting is not limited in depth.

The following elements can only be contained in sequential groups:

- Digital patterns.
- Actions.
- Transaction sequences of protocols.
- Delays to postpone the next element.

```
OpSeqParSeqExample.seq
1 sequence OpSeqParSeqExample {
2     parallel parGrp1 {
3         sequential seq1_1 {
4             patternCall patterns.PatternAO PAO_1;
5             patternCall patterns.PatternAO PAO_2;
6         }
7         sequential seq1_2 {
8             wait 0.6ms;
9             patternCall patterns.PatternAI PAI_1;
10    }
11    }
12    parallel parGrp2 {
13        sequential seq2_1 {
14            patternCall patterns.PatternBO;
15        }
16        sequential seq2_2 {
17            patternCall patterns.PatternAI PAI_2;
18        }
19    }
20 }
```

## Simple Example of Nesting

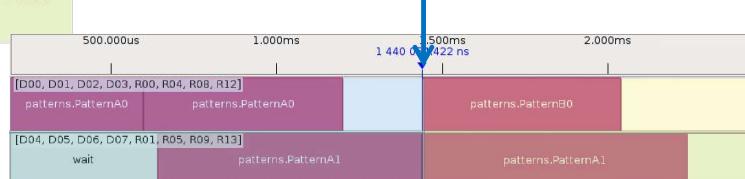
```
OpSeqParSeqExample.seq
1 sequence OpSeqParSeqExample {
2     parallel parGrp1 {
3         sequential seq1_1 {
4             patternCall patterns.PatternAO PAO_1;
5             patternCall patterns.PatternAO PAO_2;
6         }
7         sequential seq1_2 {
8             wait 0.6ms;
9             patternCall patterns.PatternAI PAI_1;
10    }
11    }
12    parallel parGrp2 {
13        sequential seq2_1 {
14            patternCall patterns.PatternBO;
15        }
16        sequential seq2_2 {
17            patternCall patterns.PatternAI PAI_2;
18        }
19    }
20 }
```

This operating sequence shows a simple example of nesting:

Each of two parallel groups contain two sequential groups.

### Synchronization:

After the two sequential groups of the blue parallel group have finished, the sequential groups of the yellow parallel group are started within one nanosecond.



# Complex Example of Nesting

```

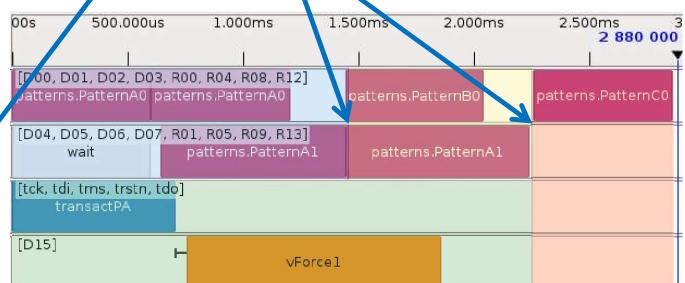
OpSeqNestingExample.seq
sequence OpSeqNestingExample {
    parallel parGrp1 {
        sequential seq1_1 {
            parallel parSubGrp1_1 {
                sequential seq1_1_1 {
                    patternCall patterns.PatternAO PA0_1;
                    patternCall patterns.PatternAO PA0_2;
                }
                sequential seq1_1_2 {
                    wait 0.6ms;
                    patternCall patterns.PatternAI PA1_1;
                }
            }
            parallel parSubGrp1_2 {
                sequential seq1_2_1 {
                    patternCall patterns.PatternBO;
                }
                sequential seq1_2_2 {
                    patternCall patterns.PatternAI PA1_2;
                }
            }
        }
        sequential seq1_2 {
            transactSeqCall transactPA;
            actionCall vForceI;
        }
    }
    parallel parGrp2 {
        sequential seq2_1 {
            patternCall patterns.PatternCO LastPat;
        }
    }
}

```

This *operating sequence* shows a more complex example of nested *parallel groups* and *sequential groups*.

## Synchronization:

After starting synchronically, the operating sequence performs two more synchronizations because of parallel groups



# Operating Sequence Setup: Elements

```

OpSeqNestingExample.seq
sequence OpSeqNestingExample {
    parallel parGrp1 {
        sequential seq1_1 {
            parallel parSubGrp1_1 {
                sequential seq1_1_1 {
                    patternCall patterns.PatternAO PA0_1;
                    patternCall patterns.PatternAO PA0_2;
                }
                sequential seq1_1_2 {
                    wait 0.6ms;
                    patternCall patterns.PatternAI PA1_1;
                }
            }
            parallel parSubGrp1_2 {
                sequential seq1_2_1 {
                    patternCall patterns.PatternBO;
                }
                sequential seq1_2_2 {
                    patternCall patterns.PatternAI PA1_2;
                }
            }
        }
        sequential seq1_2 {
            transactSeqCall transactPA;
            actionCall vForceI;
        }
    }
    parallel parGrp2 {
        sequential seq2_1 {
            patternCall patterns.PatternCO LastPat;
        }
    }
}

```

Name of the operating sequence and file name must match

Start of a parallel group

Start of a sequential group

Delay element

Pattern call

Call of a transaction sequence

Call of an action

# Operating Sequence Setup: Calls

```
OpSeqNestingExample.seq
sequence OpSeqNestingExample {
    parallel parGrp1 {
        sequential seq1_1 {
            parallel parSubGrp1_1 {
                sequential seq1_1_1 {
                    patternCall patterns.PatternAO PA0_1;
                    patternCall patterns.PatternAO PA0_2;
                }
                sequential seq1_1_2 {
                    wait 0.6ms;
                    patternCall patterns.PatternAI PA1_1;
                }
            }
            parallel parSubGrp1_2 {
                sequential seq1_2_1 {
                    patternCall patterns.PatternB0;
                }
                sequential seq1_2_2 {
                    patternCall patterns.PatternAI PA1_2;
                }
            }
        }
        sequential seq1_2 {
            transactSeqCall transactPA;
            actionCall vForce1;
        }
    }
    parallel parGrp2 {
        sequential seq2_1 {
            patternCall patterns.PatternCO LastPat;
        }
    }
}
```

The same patterns, transaction sequences and actions can be called multiple times in an operating sequence

Pattern calls use as a parameter a qualified name. The path is relative to the source folders, without the file suffix ".pat".  
"." is the delimiter of folder hierarchies

Pattern call with optional user defined name

Transaction sequences and actions are called by their name as defined in an associated specification file

## Reference to Related Main Specification File

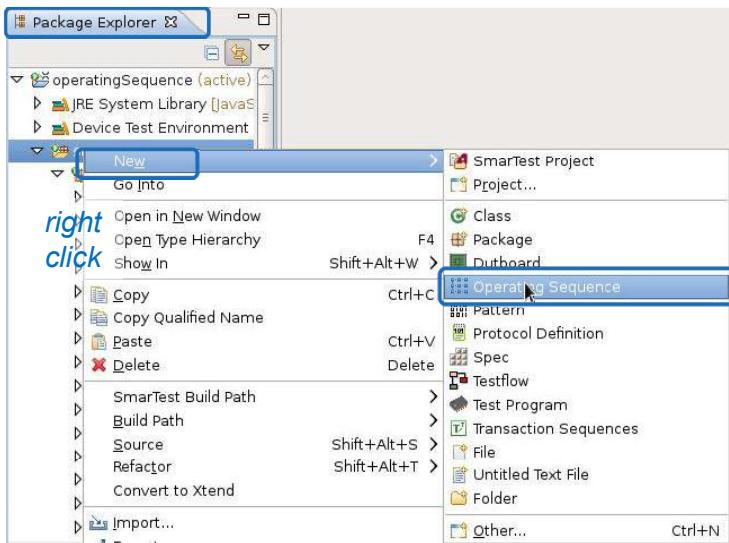
```
OpSeqNestingExample.seq
sequence OpSeqNestingExample uses mainSpecs.OpSeqSync2Pat1Trans2Act {
    parallel parGrp1 {
        sequential seq1_1 {
            parallel parSubGrp1_1 {
                sequential seq1_1_1 {
                    patternCall patterns.PatternAO PA0_1;
                    patternCall patterns.PatternAO PA0_2;
                }
                sequential seq1_1_2 {
                    wait 0.6ms;
                    patternCall patterns.PatternAI PA1_1;
                }
            }
            parallel parSubGrp1_2 {
                sequential seq1_2_1 {
                    patternCall patterns.PatternB0;
                }
                sequential seq1_2_2 {
                    patternCall patterns.PatternAI PA1_2;
                }
            }
        }
        sequential seq1_2 {
            transactSeqCall transactPA;
            actionCall vForce1;
        }
    }
    parallel parGrp2 {
        sequential seq2_1 {
            patternCall patterns.PatternCO LastPat;
        }
    }
}
```

If an operating sequence is always used with the same main specification file, then it is recommended to refer to this specification file via "uses".

As a consequence, pressing "F3" on actions or protocol sequences opens their declaration in the referred specification file.

"uses" allows to easily access declarations of actions or transaction sequences.

# Creating a New Operating Sequence File



Like for other files in the *SmarTest Setup Format*, content assist is provided for editing *operating sequence files*.

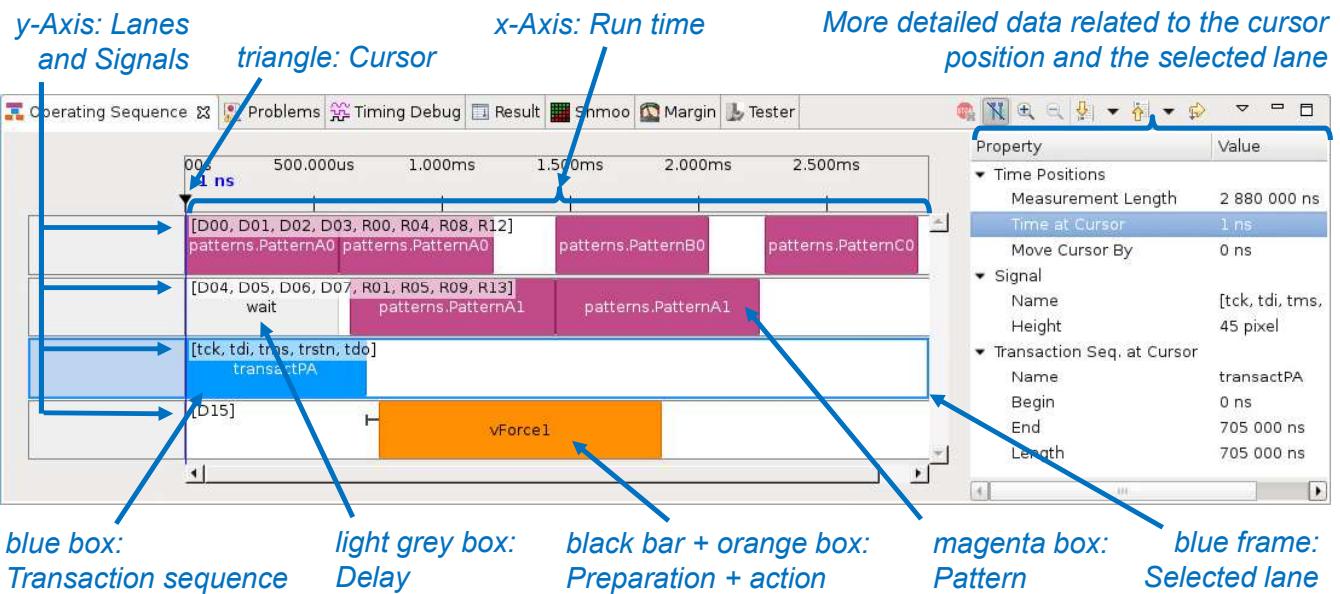
A screenshot of the SmarTest IDE showing the code editor for a new operating sequence file named 'NewOpSeq.seq'. The code is as follows:

```

1 sequence NewOpSeq {
2     patternCall some_existing_pattern;
3         crossconnect.basicDigital.patterns.DigitalCap
4             crossconnect.basicDigital.patterns.DigitalCapt
5             crossconnect.basicDigital.patterns.DigitalCapt
6             crossconnect.basicDigital.patterns.Functional
7             crossconnect.basicDigital.patterns.Functional
8             crossconnect.basicDigital.patterns.Functional
9             crossconnect.basicDigital.patterns.Functional
10            crossconnect.basicDigital.patterns.Functional
11            crossconnect.basicDigital.patterns.Functional
12            crossconnect.basicDigital.patterns.Functional
13            crossconnect.basicDigital.patterns.Functional
14            crossconnect.basicDigital.patterns.Functional
15            crossconnect.dcLab.patterns.DcTrigger
}

```

## Debug: Operating Sequence View



# Summary - What you should have learnt

- The possible elements of an *operating sequence* are:
  - digital patterns;
  - protocol operations;
  - *actions*;
  - waiting times;
  - *parallel groups* (which can contain *sequential groups*);
  - *sequential groups* (which can contain all the other elements listed above).
- For *sequential groups*, that are part of the same *parallel group*, the execution of the first elements of these *sequential groups* is started at the same time, means within 1 nanosecond (default value).
- Debugging of a *operating sequence* setup with a graphical view is supported by the *operating sequence view* after running the corresponding test suite in debug mode.



# Hardware Overview

SmarTest 8.2.5 Training

January 2020

# Learning Objectives

- Know and understand the scalable architecture of the V93000 SOC test system.
- Know and understand the components of the V93000 SOC test system.
- Know and understand the characteristics of the device power supply, digital and analog boards of the V93000 SOC test system.
- Know and understand the instruments of hardware cards.

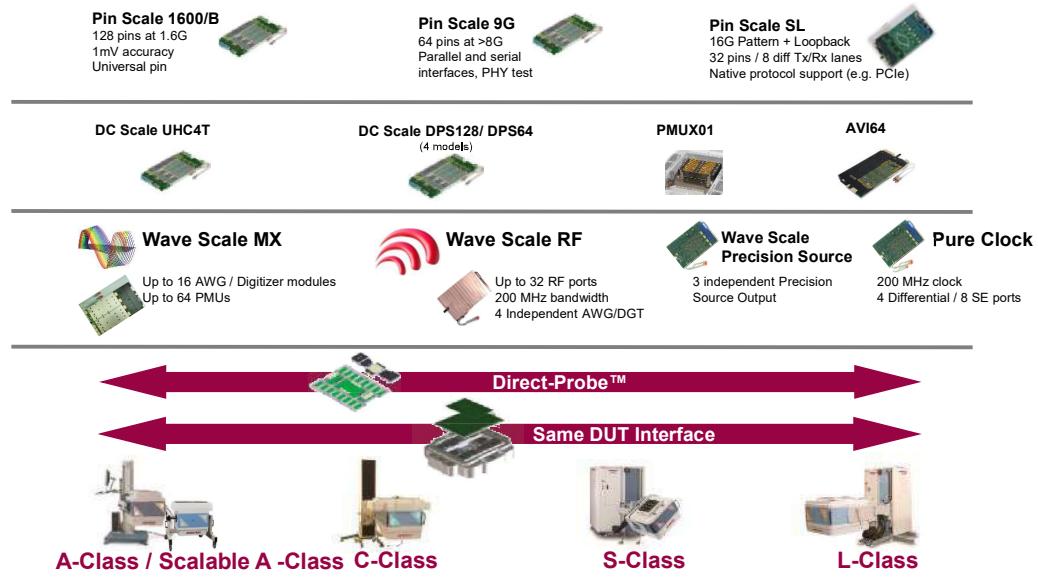
# Agenda

- V93000 scalable platform
- V93000 SOC Family Overview
- Digital cards (PS1600)
- Instruments model
- Advanced Hardware features (optional)
  - RF cards
  - MX cards
  - DPS cards
  - Utility lines
  - Model file

# V93000 Scalable Platform

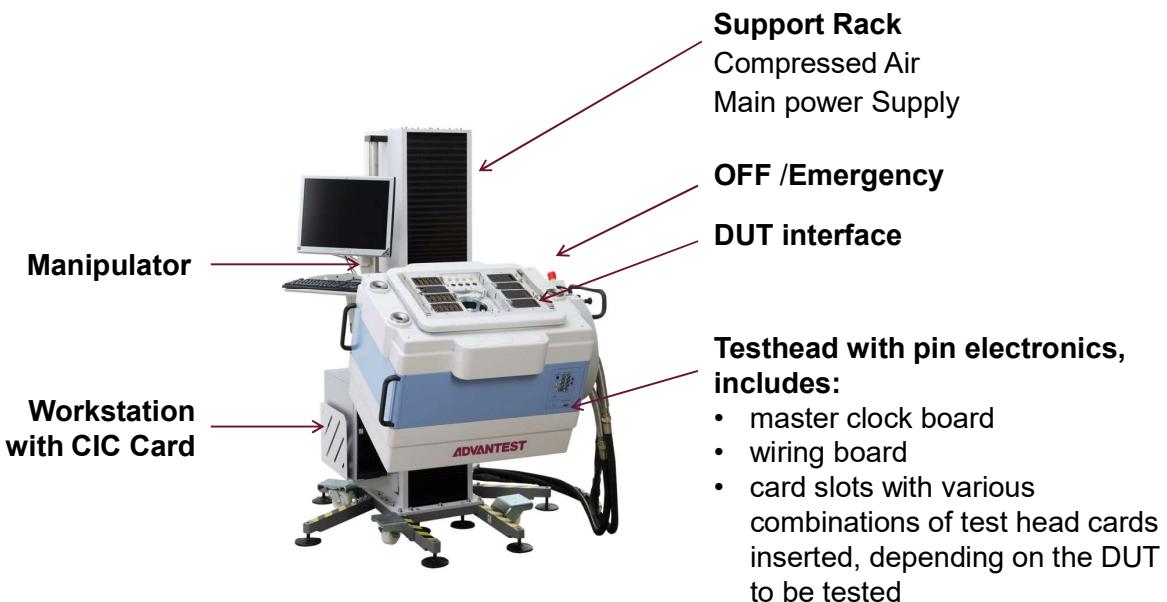


# V93000 Scalable Platform



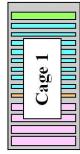
SmarTEST 8 + Libraries + Test Programs + InstaPin™ Licensing

# V93000 SOC Tester Platform

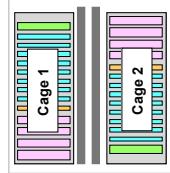


## V93000 SOC Family Overview

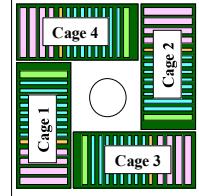
A test head  
1 card cage  
8 Slots



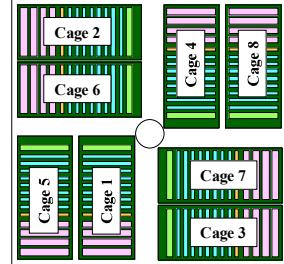
C test head  
2 card cages  
16 Slots



S test head  
4 card cages  
32 Slots

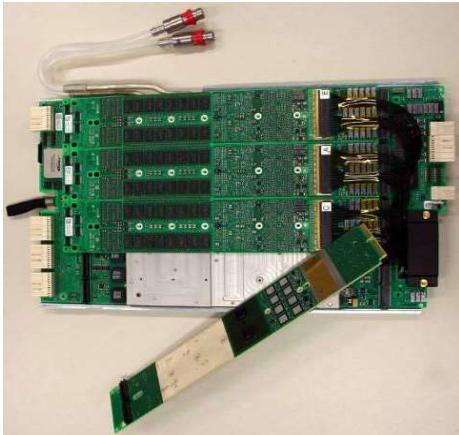


L test head  
8 card cages  
64 Slots



# Pin Scale 1600 Testhead Card

Pin Scale 1600  
128 digital channels per card  
1600Mbps data rate  
High accuracy DC, HV PE

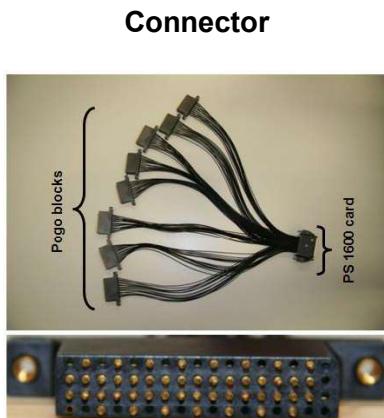


- 128 digital pins / card
- Clock Domain per Pin
- Single ended & differential signals
- 0..1600Mbps data rate
- OTA/EPA = ±200/100ps(system), ±160/80ps(board)
- 112MB/pin, 448MVec/pin (x4 mode)
- Driver: -1.5 V to 6.5 V
- High Voltage Driver (6V to 13.5V) per 16 pins
- Active Load with ≤25mA
- Pin PMU     -- - 2.0 V to 6.50 V  
              -- ± 40mA, ± 65mA (with active load)  
              -- Sequencer controlled
- Board ADC per 16 pins
  - -3.0 V to 8.0 V
  - High impedance, accurate V-measure
  - Sequencer controlled
- Time Measurement Unit per Pin
- SmartLoop™ per Pin
- PRBS per Pin
- Protocol Engine per Pin – Tracking & Payload detection
- AWG and DGT capability

## Pin Scale 1600 Pogo Blocks

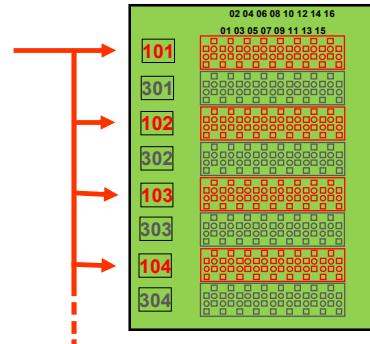
Example:

Boards with 8 Pogo Connectors (e.g. PS1600) in Group 1/Card Cage 3



The pogo connector of a PS1600 card link (8x16) 128 channels.

### Pogo Blocks on DUT Board



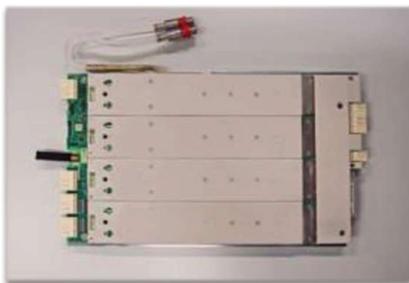
Compatible Pogo Blocks  
16 Channels / Block

# DC Scale DPS128HC, DPS64HC

## DC Scale DPS128HV, DPS128HC/HV

High Current DPS Cards and Combined Mixed High Current / High Voltage DPS Card

**Four new dense general purpose DPS cards for large multi-site setups with high measurement throughput**



### Target Applications:

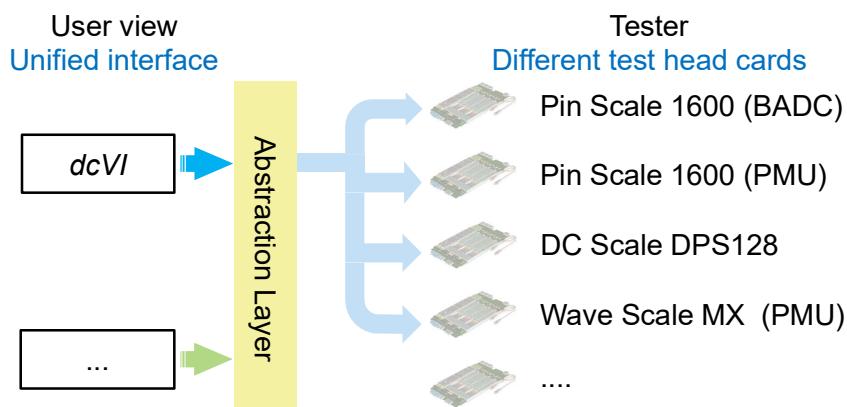
- For large multi-site setups and/or many power domains requiring hundreds of DPS channels.
- MCUs/eFlash/mobile/sensor devices.

### Key Characteristics

- 128 channels.
- High Voltage channels with -6V/+15V.
- Fits in any number of I/O slots.
- All DPS64/128 versions can be installed next to each other.
- May gang as many channels on the same card as needed.
- Synchronous triggering.
- Waveform generation (200ksps) and sampling (90ksps) with 28MB sample memory.
- Slew rate control and profiling.

## Key Concept of the SmarTest 8 Instruments

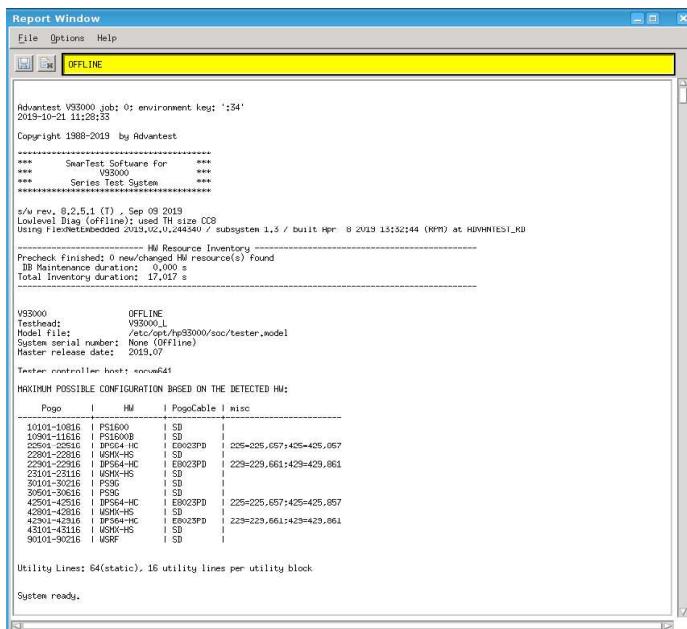
An *instrument* represents a unified interface to allow to program the same way different test head cards that provide the same core functionality.



# Testhead Cards vs. Instruments

	dcVI	digInOut	clock	rfMeas	rfStim	awg	rfVna	digitizer	utility	Loopback
PS1600	Yes	Yes	Yes			Yes			Yes	
PS 9G	Yes	Yes	Yes							Yes
PS SL	Yes	Yes	Yes							
DPS128	Yes									
DPS64	Yes									
AVI 64	Yes	Yes				Yes		Yes		
UHC4T	Yes									
RF Pure Clock			Yes							
WSRF				Yes	Yes		Yes			
WSMX	Yes					Yes		Yes		
Wiring board									Yes	

## Model File



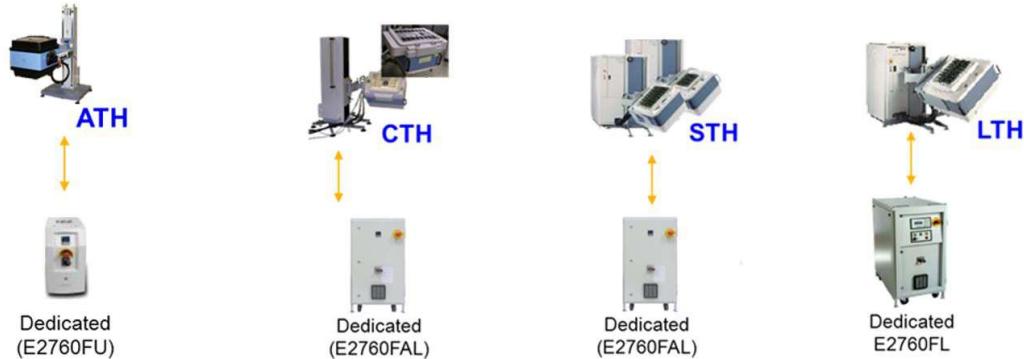
- Default location:  
/etc/opt/hp93000/soc/
- Default name: tester.model
- In addition to the default path, the path of the model file can be specified as well:  
Use the environment variable V93000\_MODEL to specify the model file with its absolute path and its file name.

## Summary - What you should have learned

- The main components of the V93000 scalable platform are the support rack, the manipulator, the test head, the DUT interface, the workstation and the emergency off switch.
- A test head consists of card cages and each card cage provides 8 channels slots for test head cards.
- Four different sizes for the test head with different numbers of card cages allow to scale with the number of required tester channels over a wide range.
- The various test head cards targeting the domains digital, mixed signal, RF and power provide a high flexibility in equipping a test head as needed to cover all the different test applications.
- SmarTest 8 makes it simple to use all the possible tester configurations with the various tester resources simple and it facilitates reuse – for example with the concept of the SmarTest 8 Instruments.
- Pogo cables connect the test head cards with the DUT interface.
- The PS1600 card is a digital card providing 128 channels with a data rate of up to 1600Mbps.
- The DC Scale DPS128/DPS64 is a high density power supply.

## Advanced Hardware Features (optional)

# Water Cooling Technology



## V93000 Controller Hardware/ Workstation

**HP Z640**  
- special Advantest HW configuration  
ONLINE workstation bundle E7039NA  
OFFLINE workstation bundle E7039NC

**Computer Interface Card / PCI-E**  
CIC card (Advantest proprietary) to connect to V93000 tester  
Included in ONLINE workstation bundle E7039NA



**HP NC361T DP Gbit NIC / PCI-E**  
2-port Gigabit PCI-E card  
Included in ONLINE & OFFLINE workstation bundles



**NVIDIA NVS 310 512MB 1st UEFI GFX / PCI-E**  
512MB Graphics Card - supports 2 displays  
Included in ONLINE & OFFLINE workstation bundles

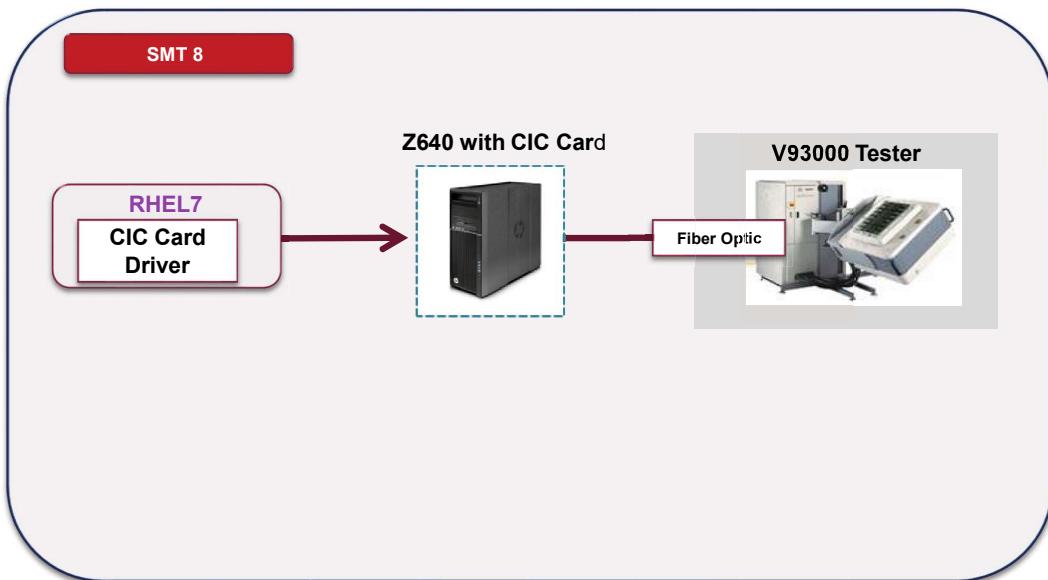


**HP Z24i**  
24-inch LED Backlit IPS Monitor  
Option E7039FP

**LAN to GPIB adapter Kit**  
(includes Keysight E5810B, aka Milan box)  
Option E7032C



# V93000 Tester with Work Station Z640



## DC Scale UHC4T

**Scalable High-Current DPS  
Solution for performance  
SOC and Processor  
Applications.**



### Target Applications:

- Multisite testing of large SOC devices/processors

### Key Characteristics

- 4 x 40A per board
- Gangable across multiple units
- Ultra-Fast Load Response
- Integrated High-speed Differential Sampler for current/voltage profiling
- Integrated Precision Measurement Capabilities

# AVI64 Universal Analog Pin

**Next generation  
“General purpose” DCVI  
with Analog and High Voltage  
Digital capabilities.**

## Target Applications:

- General purpose DC and analog
- MCU, Smart Sensor
- PMIC
- Smart LED
- Automotive, industrial

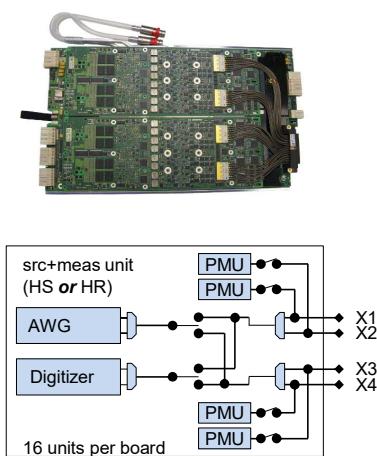


## Key Characteristics

- 64 analog VI channels (4Q, Kelvin)
  - -40V...+80V, 200 mA @  $\pm 10V$
  - Ganging for higher currents supported
  - Current and voltage monitoring
  - Programmable voltage/current clamp
  - AWG and DGT (V & I)
  - Digital IO + TMU
- 8 floating high current units
- 8 high resolution AWG units
- 8 floating differential voltmeters
- Test processor controlled set/measure

# Wave Scale MX

**Highest flexibility & density**



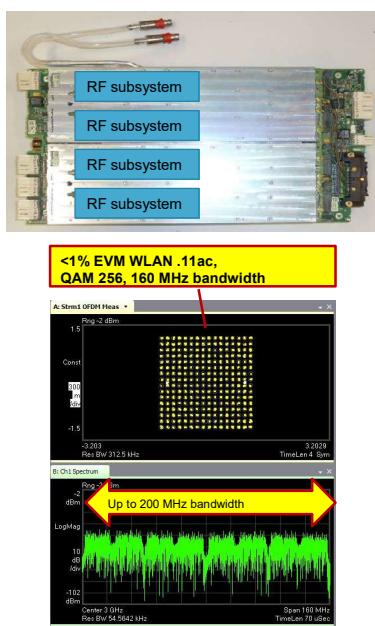
## Key Characteristics

- 32 instruments per card
- 64 bidirectional analog pogos incl. PMU
- High resolution (HR) & high speed (HS) units
  - HS: optimized performance for baseband communication standards
  - HR: 24bit AC source & measure Precision DC Voltage source & measure
- Parallel, independent operation of all instruments controlled by HW sequencer (test processor)
- Internal triggering & synchronization
- Sampling clock per instrument
- No dedicated calibration equipment needed (only IQ calibration requires dedicated board)
- Built-in 2:1 multiplexer if only AWG or DGT is needed

## Products

- Wave Scale High Speed: 16x high-speed (HS)
- Wave Scale High Resolution: 16xhigh-resolution (HR)
- Wave Scale Hybrid HS and HR: 8x HS and 8x HR

# Wave Scale RF



## Key Characteristics

4 independent RF subsystems on each card

- 32 RF ports per card

Innovative integration: higher flexibility & density

- Fits into smaller infrastructure
- Allows higher multi-site e.g. 16x with native ATE resources
- Embedded AWG & DGT for each RF subsystem
- No external trigger lines needed

RF performance: equal or better RF performance (over PSRF)

- Bandwidth (200 MHz, 350 MHz with under-sampling)
- Dynamic range of 140 / 145 dB (LNA / direct), ACLR (-65dBc)

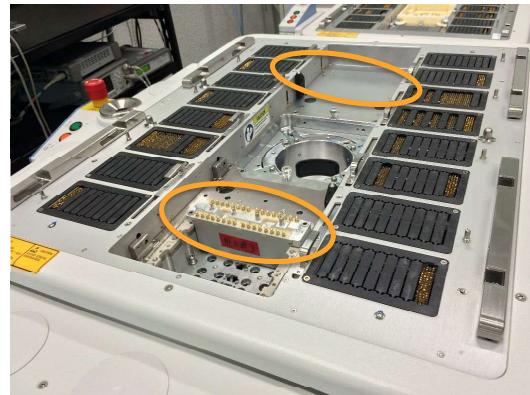
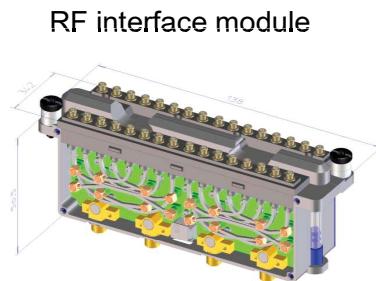
Embedded RF calibration standards per module

**Scalability for in-site parallelism and multi-site**

**Provides a choice to run tests in parallel for shortest test time.**



# WSRF DUT Interface

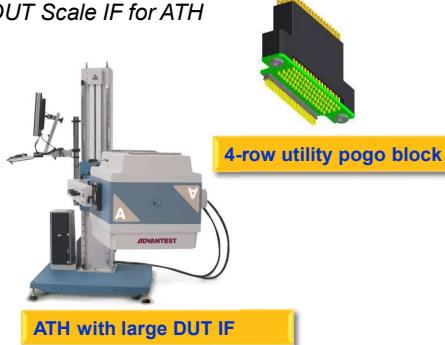


Up to 6 Wave Scale cards / modules per DUT IF

# Utility Lines

## Control and power for relays and active components on load boards

- 256 utility lines and more utility power with new 4-row utility line pogo block for all large DUT Scale interfaces
- New DUT Scale IF for ATH



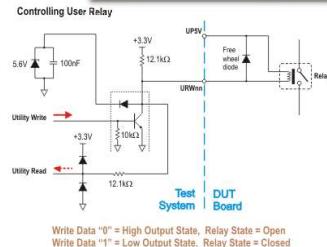
## Target Applications

- All SOC testing

## Key Characteristics

- 256 utility lines on all test heads with new large DUT Scale interface together with new utility line pogo block. (optional, default is 128 utility lines using 2-row utility line pogo block)
- All utility lines can be sequencer controlled
- Optional external utility power supply unit 8A @ +5V / 6A @ -5V / 6A @ 3.3V
- Upgrade options for installed test system base

### Utility Line – Example Circuit



Write Data "0" = High Output State, Relay State = Open  
Write Data "1" = Low Output State, Relay State = Closed

# Model File

The model file is the description of the tester which should be emulated. It has two purposes:

1. Reflect the test system in case no HW is available. To be able to use the SmarTest software in offline mode.
2. Not all of the HW settings and environmental settings can be automatically detected by the SmarTest software (e.g. the pogo cabling cannot be detected). These settings are also part of the model file and explain why the model file also in online mode is an important configuration file.

# Model File Location and Name

- Location: /etc/opt/hp93000/soc/
- Name: tester.model
- In addition to the default path the model file can be specified as well using the environment variable V93000\_MODEL which requires specifying the file with an absolute path and file name.

# Model File Structure

The tester.model is organized into the following four sections:

## GLOBAL

- specifies what kind of system is used and what features are available system-wide.

## CARD\_CONFIGURATION

- specifies the physical position of all the cards and the mapping between card slots and pogo blocks on the DUT interface.

## ETC

- defines several general parameters, for example the reference clock.

## DMM

- specifies optionally installed external digital multimeters.

## Model File Entry Example (1/3)

```
GLOBAL
testhead      = TH_8CC
dut_interface = SOC_8GROUP
hpib_interface = vxi11/e5810a/192.168.0.100/gpib0
utility_lines   = 64, type=static, pogo_size=16
pogoblocks    =866,868,870,872,881,883,885,887
```

## Model File Entry Example (2/3)

### CARD\_CONFIGURATION

```
slot = 301, HW = PS1600, pogoblocks=101-108
slot = 303, HW = WSMX,  pogoblocks=HS:(231 431) HR:(233 433)
slot = 304, HW = WSMX,  pogoblocks=HS:(232 632) HR:(234 634),
                           type=HD2x32

slot = 105, HW = DPS128, pogoblocks=HC:(341 342 459 460),
                           type=REDIRECT, subtype=E8023PF,
                           config=1:(341-342) 2:(459-460)
```

## Model File Entry Example (3/3)

ETC

REFERENCECLOCK: input = INTERNAL, output\_at\_rf\_connector = ON

DMM (Digital Multimeter)

- 1: cardcage= 1,type=HP3458A,address= 18
- 2: cardcage= 2,type=HP3458A,address= 19
- 3: cardcage= 3,type=HP3458A,address= 24



## Instruments

### SmarTest 8.2.5 Training

January 2020

# Learning Objective

- Understanding the concept of the SmarTest 8 *instruments*
- Learn and be able to program and use the SmarTest 8 *instruments*

# Agenda

- Concept of the SmarTest 8 *instruments*
- Programming *instruments*
- Setup examples

# What is a SmarTest 8 Instrument?

**Instruments** are abstractions of common bench equipment – aimed to set up and perform typical measurements, such as:

- Digital multi-meter → DC VI (*dcVI*)
- Waveform generator → Analog source (*awg*)
- Oscilloscope → Analog measure (*digitizer*)
- RF signal source → RF source (*rfStim*)
- Spectrum analyzer → RF measure (*rfMeas*)
- Network analyzer → RF VNA (*rfVna*)
- Logic analyzer → Digital IO (*digInOut*)

Additional available instruments:  
• *clock*  
• *loopback*  
• *utility*

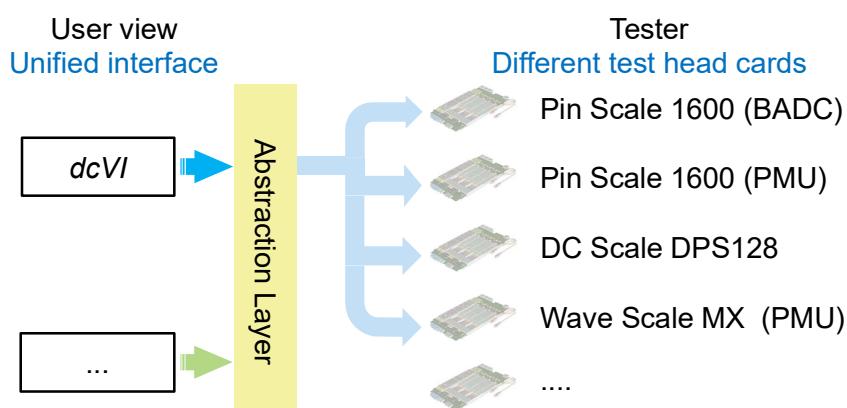
Note:

The test head cards are often also called “instruments”.

In the context of SmarTest 8, use “*instruments*” as described above.

## Key Concept of SmarTest 8 Instruments

An *instrument* represents a unified interface to allow to program the same way different test head cards that provide the same core functionality.



# List of SmarTest 8 Instruments

- *awg instrument:*  
Represents an arbitrary waveform generator that sources a waveform (analog signal).
- *clock instrument:*  
Provides a clock.
- *dcVI instrument:*  
Provides DC current/voltage measurement and device power supply capabilities.
- *digInOut instrument:*  
Provides digital driver and receiver capabilities, and the capability to execute specific DC and time measurement.
- *loopback instrument:*  
Receives data from one or more output pins of the DUT and routes them to one or more input pins of the DUT.

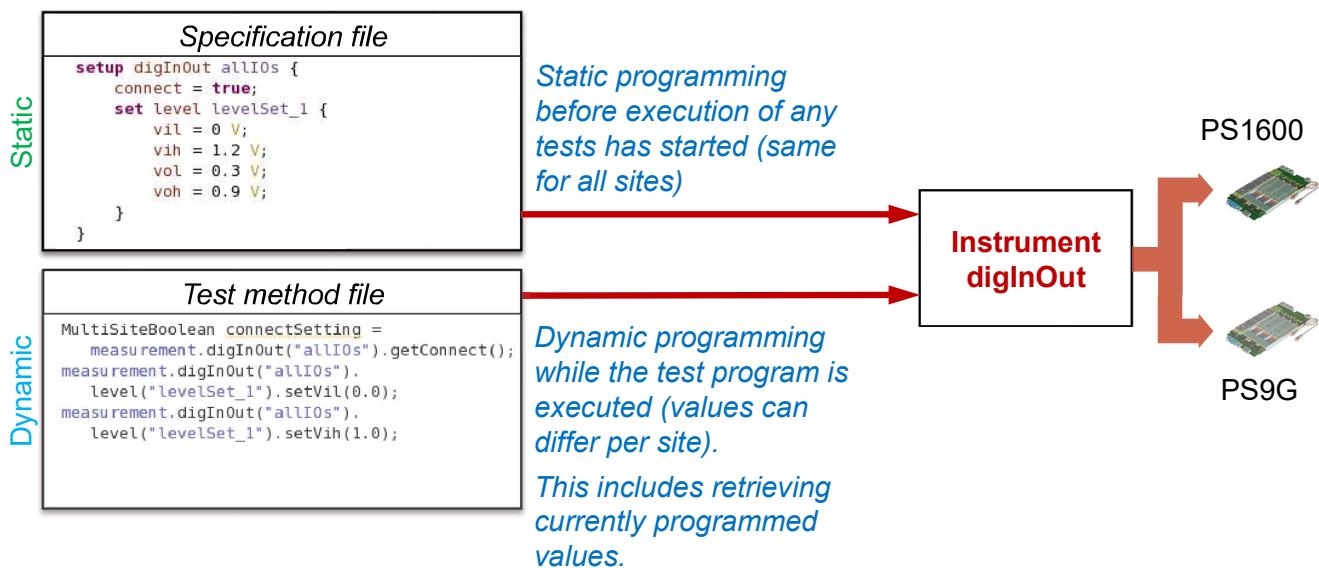
# List of SmarTest 8 Instruments

- *digitizer instrument:*  
Digitizes an analog waveform.
- *rfStim instrument:*  
Provides a continuous waveform (CW), a modulated waveform or a noise signal.
- *rfMeas instrument:*  
Provides scalar measurement capabilities.  
It has one measurement port and can be run in high resolution or high speed mode. The mode controls the used digitizer.
- *rfVna instrument:*  
Provides vector (S-parameter) measurement capabilities (up to 2 measurement ports).
- *utility instrument:*  
Allows control of the utility lines.

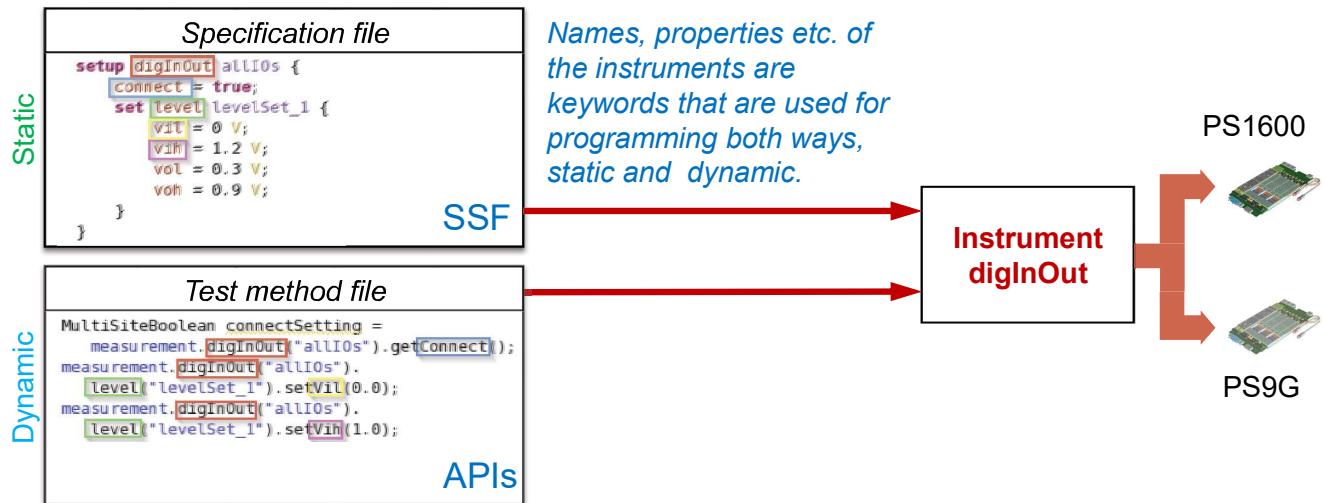
# Agenda

- Concept of the SmarTest 8 *instruments*
- Programming *instruments*
- Setup examples

## Instrument Programming



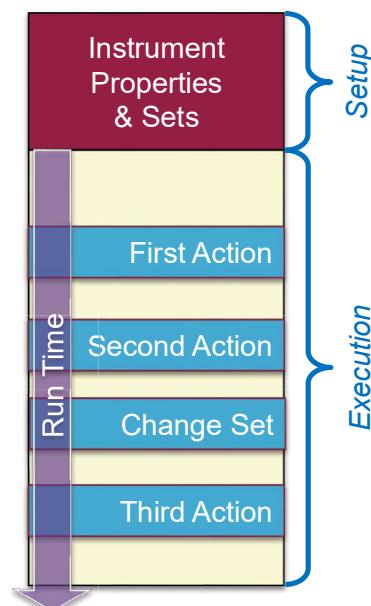
# Instrument Programming



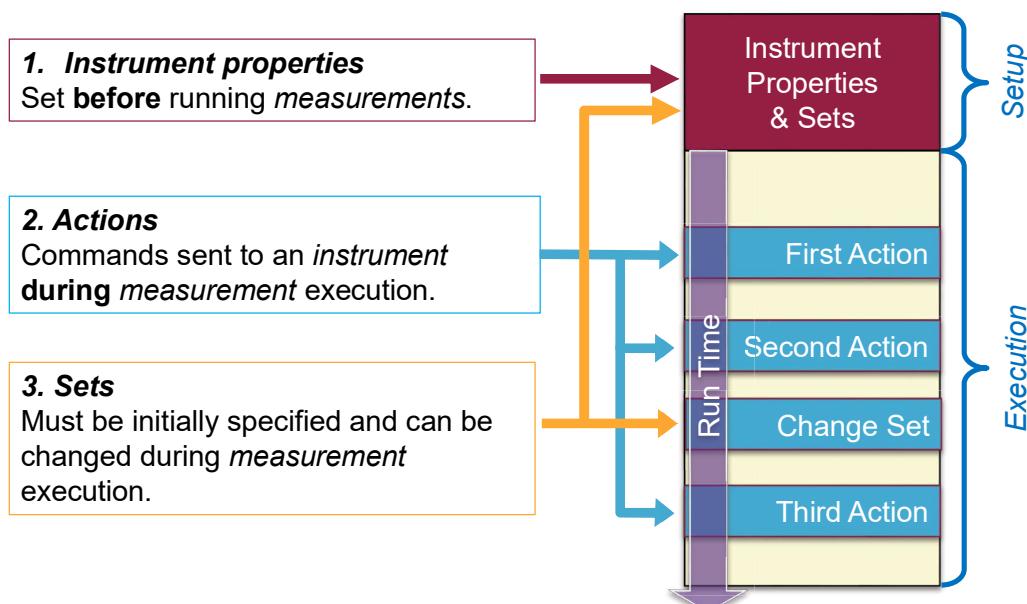
## Setup & Programming: Overview

Setup and programming of *instruments* is based on 3 concepts:

1. *Instrument properties*
2. *Actions*
3. *Sets (digInOut instrument only)*



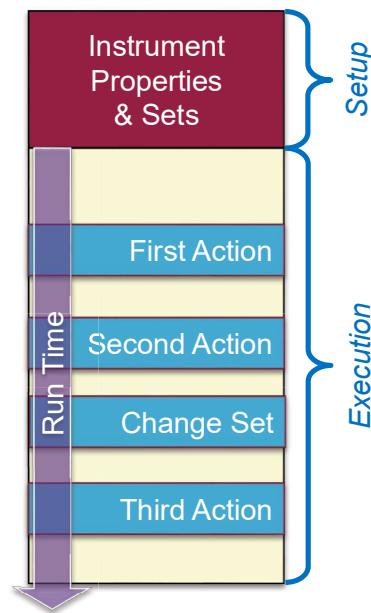
# Setup & Programming: Overview



## Setup & Programming: 1. Instrument Properties

### (1) Instrument properties - including extensions, options, modes, connections

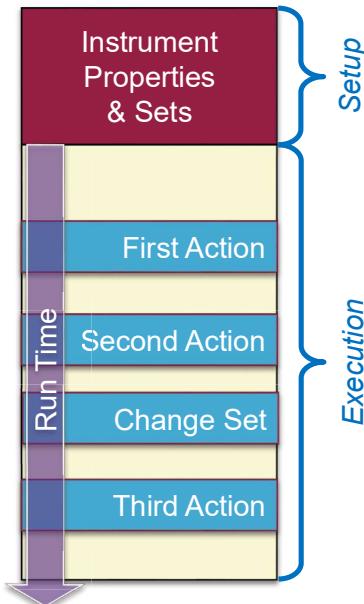
- Defined in specification files or test method code.
- Describe initial settings for the instrument or a behaviour that applies to the complete measurement.
- Examples (*dcVI instrument*):
  - Defining the connect behaviour;
  - Setting voltages;
  - Specify current clamp settings.
- Exception to initial settings:  
The instrument property “disconnect” controls if the instrument disconnects after measurement execution.



# Setup & Programming: 2. Actions

## (2) Actions

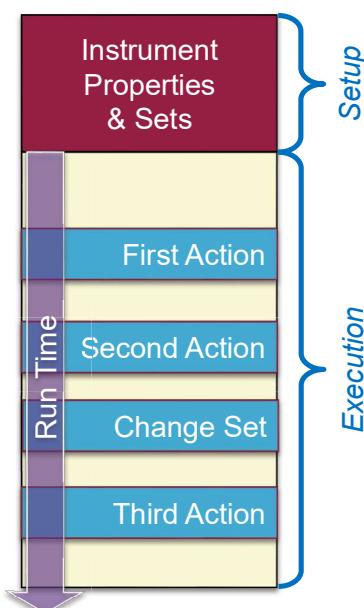
- Defined in *specification files* or test method code.
- Commands sent to the *instrument* **during** a *measurement execution*.
- To define the timing of an *action* execution:
  - Call it in an operating sequence (“*actionCall*”).
  - Attach to a vector of a pattern an *anchor* referring to it.
- Examples (*dcVI instrument*):
  - vforceImeas*,
  - iforceVmeas*.



# Setup & Programming: 3. Sets (digInOut Only)

## (3) Sets

- Mixture of *instrument properties* and *actions*.
- Must be set initially but can be changed **during** measurement execution.
- Only two types: *level sets* and *timing sets*.
- Changes of *sets* are triggered by specific *sequencer instructions* in patterns.



# Special Types of Properties

Special types of *instrument properties*:

- **Extensions**

Access to feature sets that are not part of the base functionality of an instrument and may not be supported by all test head cards.

Examples *digInOut instrument*: “equalization”.

- **Options**

Access to optional hardware resources or capabilities.

Example *digInOut instrument*: “differential”.

- **Modes** (rfStim & rfMeas instruments only)

A set of options, but only one of these can be active.

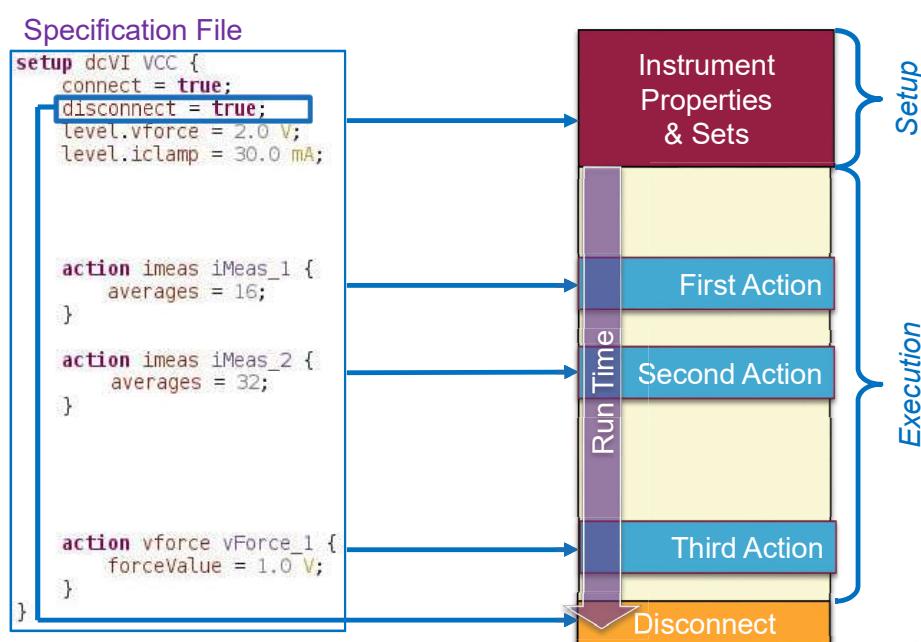
- **States**

For debugging only. To retrieve the state of the instrument.

Multiple instrument or action properties affecting the same functionality are grouped in a **property group**.

Example: “level.vforce” and “level.iclamp“ of the “level” group of the *dcVI instrument*.

## Setup & Programming: dcVI Example



# Example: Instrument Reference in TDC

The screenshot shows the SmarTest 8.0.4 Documentation interface. The left sidebar contains a tree view of topics under 'SmarTest 8.0.4 Documentation'. The main content area is titled 'clock instrument reference' and describes the clock instrument providing a reference clock to the device. It details various properties and actions related to clocking, such as connections, property groups, actions, and states. A 'Related information' section links to 'Instrument reference'. At the bottom, there are buttons for 'Send Feedback', 'Send Link', and 'Last change 2016-07-25 Topic 0094755854 (clock)'.

## Setup in Specification Files: Source and Tables

An alternative for setting up *instruments* is through **tables** which provide an easy way to check all possible settings of an *instrument*.

The screenshot shows two windows. The top window displays a specification file named 'clockSetup.spec' with the following code:

```
1 spec clockSetup {
2   setup clock D01 {
3     level.vil = vcc/2 - IV_Swing/2;
4     level.vih = vcc/2 + IV_Swing/2;
5     timing.period = 42ns;
6     timing.dutyCycle = 0.4;
7     timing.phaseShift = 0 Deg;
8     shape = square;
9 }
```

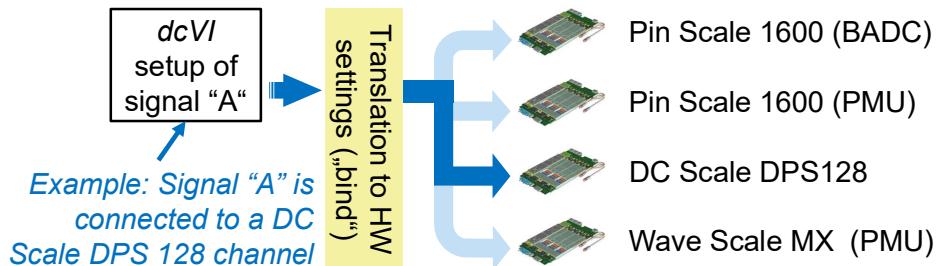
The bottom window is a 'Properties' table editor for the 'D01' row. The table has columns for Signal, connect, disconnect, keepAlive, level (with sub-columns for vil, vih, term), shape, s..., timing (with sub-columns for phaseShift, period, minA), and other parameters. The 'level' row for 'D01' is highlighted.

# Execution of Instrument Setups

In order to execute tests, setups of *instruments* must be translated

- from the abstract level of *instruments*
- to settings for the channels of the actual test head cards in the V93000.

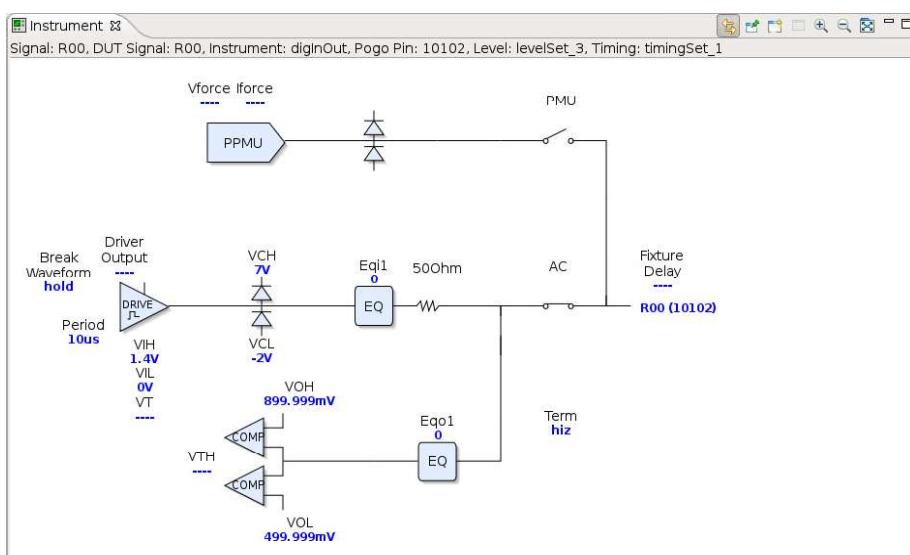
This process is called "bind".



If the capabilities of the actual test head card do not match the instrument settings, then SmarTest throws a *bind* error.

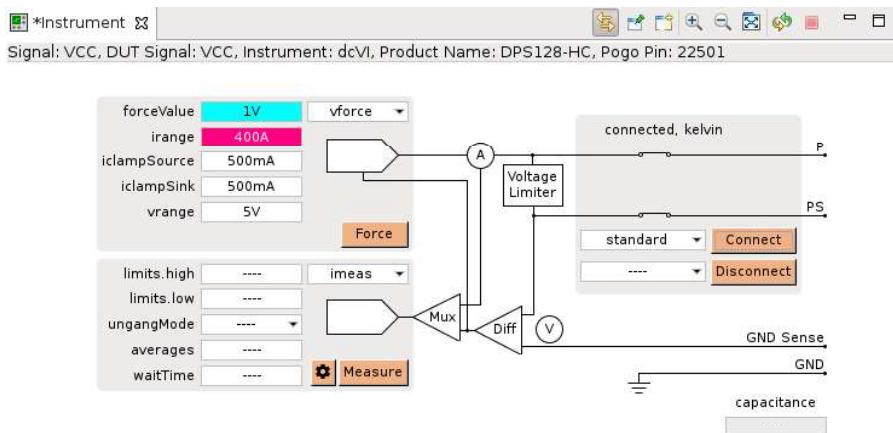
## Debug: Viewing Setup of an Instrument

The *instrument view* shows the state of *instruments* while in debug mode.



# Debug: Viewing Setup of an Instrument

You can use the *instrument view* to interactively debug dcVI measurements.



## Agenda

- Concept of the SmarTest 8 *instruments*
- Programming *instruments*
- Setup examples

## Setup Example: *digInOut*

```

spec DigInOutSetup {
    // declare sets and actions
    set levelSet_1;
    set timingSet_1;
    action ppmuIFVM;
}

setup digInOut DSO {
    // property
    connect = true;
    // level set
    set level levelSet_1 {
        vil = 0 V;
        vih = 1 V;
        vol = 0.5 V;
        voh = 0.5 V;
    }
    // timing set
    set timing timingSet_1 {
        period = 100 ns;
        d1 = 0 ns;
        r1 = 50 ns;
    }
    // action
    action iforceVmeas ppmuIFVM {
        forceValue = 100 uA;
        waitTime = 10 ms;
    }
}

```

*Name of the specification file*

*Signal (pin) or group of signals*

*Instrument setup starts with "setup"*

*Instrument type*

*Static setups of instruments are stored in specification files.*

*Declaration of sets and actions with user defined names*

*Definition of the level set*

*Definition of the timing set*

*Unit is required*

*Definition of the action*

## Setup Example: *dcVI*

```

spec DcVISetup {
    // declare action
    action iMeas_1;

    setup dcVI VCC {
        // properties
        connect = true;
        disconnect = true;
        level.vforce = 2.0 V;
        level.waitTime = 10.0 ms;
    }

    // action
    action imeas iMeas_1 {
        averages = 16;
        limits.high = 20.0 mA;
        limits.low = 1.0 mA;
    }
}

```

*Instrument properties that set up the instrument before (and after) executing the measurement*

*Action properties that define how the action is performed when called*

*"vforce" and "waitTime" of the "level" property group*

*Items of a property group are connected by a dot “.”*

## Setup Example: *clock* (via SSF and Table)

The screenshot shows two windows side-by-side. The top window is titled 'clockSetup.spec' and contains the following code:

```
1 spc clockSetup {
2   setup clock D01 {
3     level.vil = vcc/2 - IV_Swing/2;
4     level.vih = vcc/2 + IV_Swing/2;
5     timing.period = 42ns;
6     timing.dutyCycle = 0.4;
7     timing.phaseShift = 0 Deg;
8     shape = square;
9 }
```

The bottom window is titled 'Properties' and displays a table for the 'clock' instrument. The table has columns for Signal, connect, disconnect, keepAlive, level (with sub-columns for vil, vih, term), shape, s..., timing (with sub-columns for phaseShift, period, minA), and actions. A single row is shown for 'D01'.

Signal	connect	disconnect	keepAlive	level	shape	s...	timing	actions
				vil      vih      term			phaseShift      period      minA	
D01				vcc/2 - ... vcc/2 +...	square	0 Deg	42ns	

Instruments are setup in specification files using the SmarTest Setup Format ("Source") or using tables.

## Summary - What you should have learned

- An *instrument* represents a unified and abstract software interface that allows to set up the same way different test head cards that provide the same core functionality.
- Therefore, *instrument* setups can be re-used for different hardware configurations.
- Ten different types of *instruments* exist:  
digInOut, dcVI, awg, digitizer, rfStim, rfMeas, rfVna, clock, utility and loopback.
- The setup of the *instrument* consists of specifying *instrument properties* and defining *actions* and, in case of the *digInOut instrument*, setting up timing and level sets.
- Instrument properties specify the setup of the instrument before and after the *measurement* execution.
- *Actions* are commands sent to the *instrument* during a *measurement* execution.
- *Instrument* settings can be specified in two ways:
  - statically in *specification files* before any test has been executed or
  - dynamically in test methods while the test program is running and tests have been executed.



# Basics of Level and Timing Setups

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

Basics of Level and Timing Setups - 1

## Learning Objective

- Understanding the purpose of level and timing setups for digital patterns
- Understanding the test cards resources used when defining levels and timings for patterns



January 2020

All Rights Reserved - ADVANTEST CORPORATION

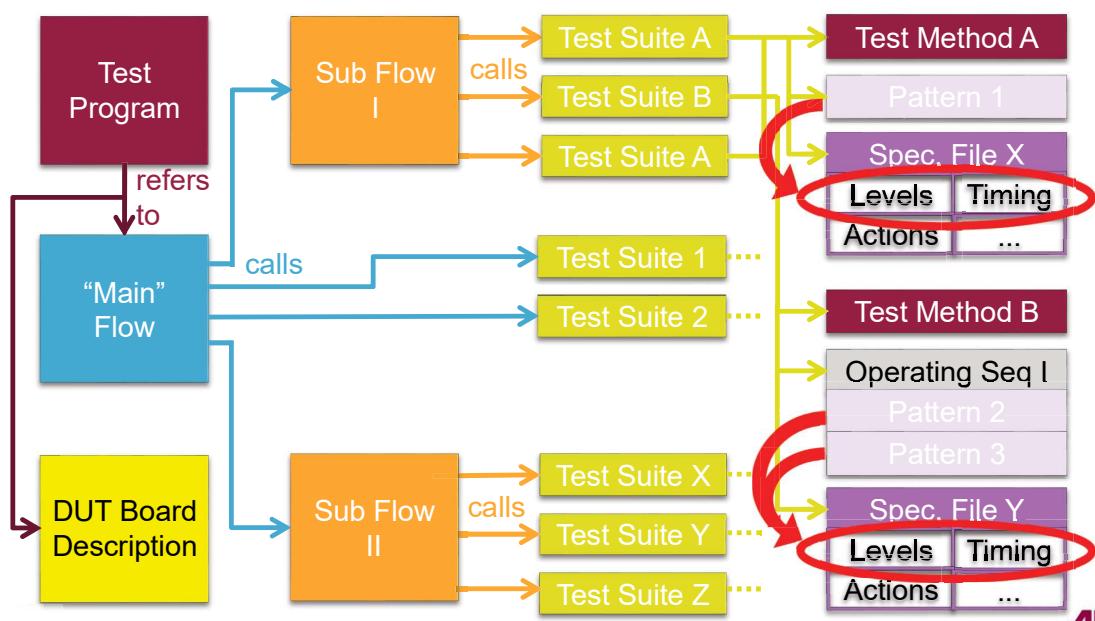
**ADVANTEST**

Basics of Level and Timing Setups - 2

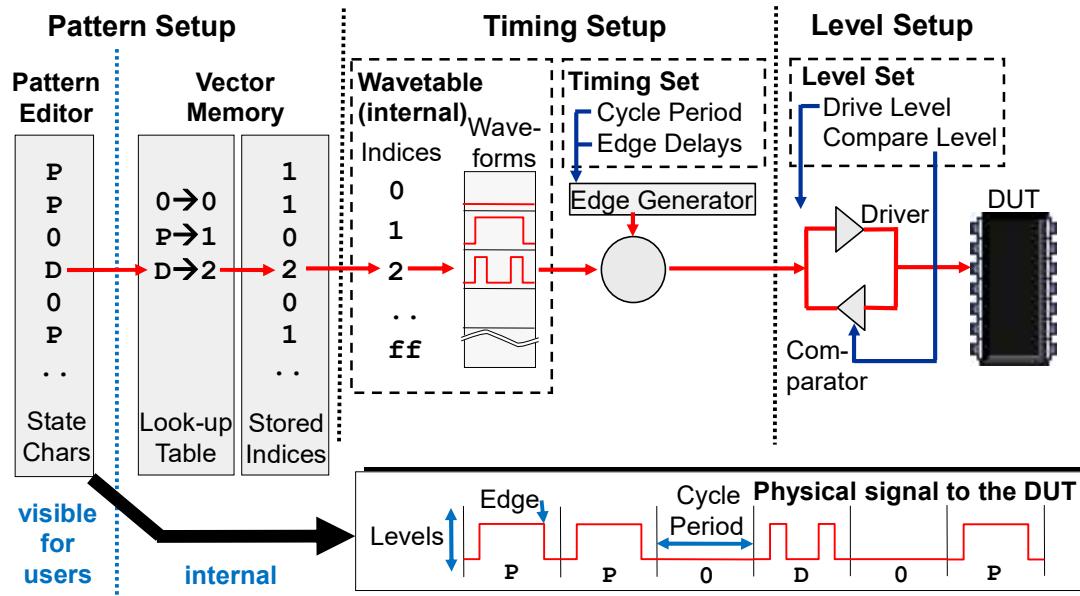
# Agenda

- Overview
- Purpose of timing setups
- Example of timing settings in a specification file
- Purpose of level setups
- Setup for a power supply

## Test Program in SmarTest 8: Building Blocks



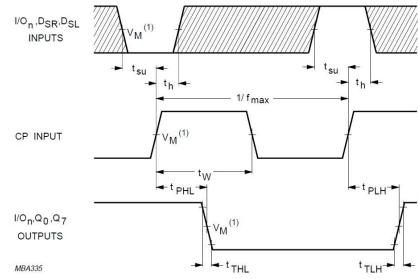
# Overview: Pattern & Timing & Level Setup



## Timing Setup: Purpose

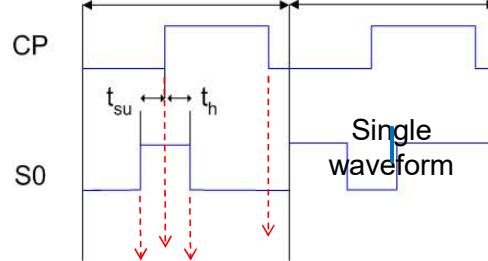
*Device requirements* →

*Translate to SmarTest specification files  
(timing sets & wavetables)*



AC CHARACTERISTICS FOR 74HCT		GND = 0 V; $t_L = t_H = 6$ ns; $C_L = 50$ pF					
SYMBOL	PARAMETER	T <sub>amb</sub> (°C)				UNIT	
		74HCT		-25	-40 to +85	-40 to +125	
$t_{PHL}$	propagation delay CP to $I/O_n$	27	46	56	69	ns	
$t_{THL}/t_{TLH}$	output transition time standard ( $Q_0, Q_7$ )	7	15	19	22	ns	
$t_W$	clock pulse width HIGH or LOW	20	10	25	30	ns	
$t_{SU}$	set-up time $S_0, S_1$ to CP	32	18	40	48	ns	
$t_h$	hold time $S_0, S_1$ to CP	0	-17	0	0	ns	
$f_{max}$	maximum clock pulse frequency	25	42	20	17	MHz	

1. Device period



2. Position of edges defined through timing set

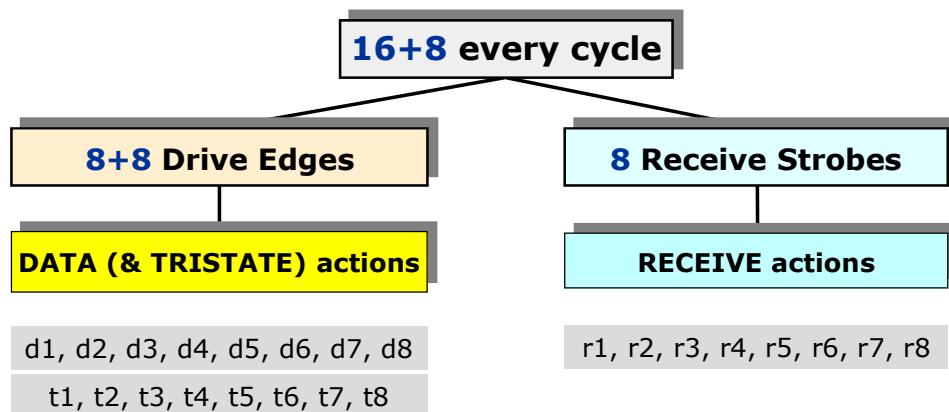
- Position of edges defined through timing set
- Sequence of actions concatenated to a waveform
- All waveforms are assembled in the wavetable
- Sequence of waveforms is stored in vectors of the pattern

# V93000 Timing Resources: Edges and Strobes

Edges and strobes are used to position the waveform transition in horizontal direction, that is the time delay vs. start of tester cycle.

Note:

There are **8 special tri-state/third-level edges**.



## Drive Syntax

### List of Drive Actions

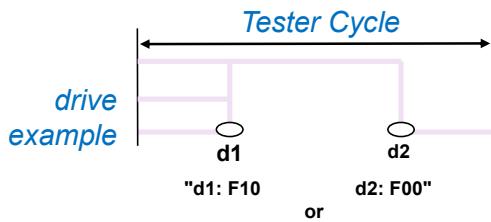
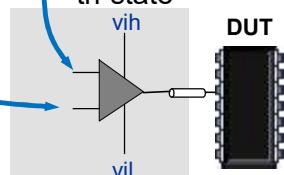
FNN
F0N
F1N
FN0 or !Z
FNZ or Z
F00 or 0
F0Z
F10 or 1
F1Z

### Drive-level Control

- 1: drive high
- 0: drive low
- N: do not switch drive level

### Tri-state Control

- Z : tri-state on
- 0 : tri state off
- N : do not switch tri-state

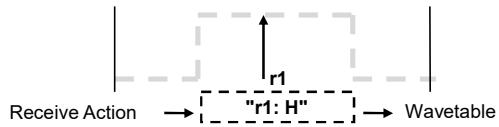


# Receive Syntax

## Edge Compare

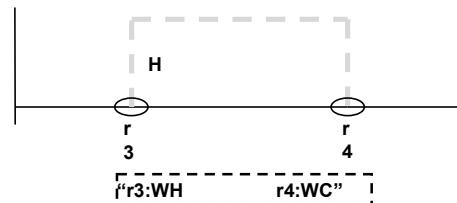
L	compare to low
H	compare to high
M	compare to intermediate
X	don't care (mask)

## Receive Examples

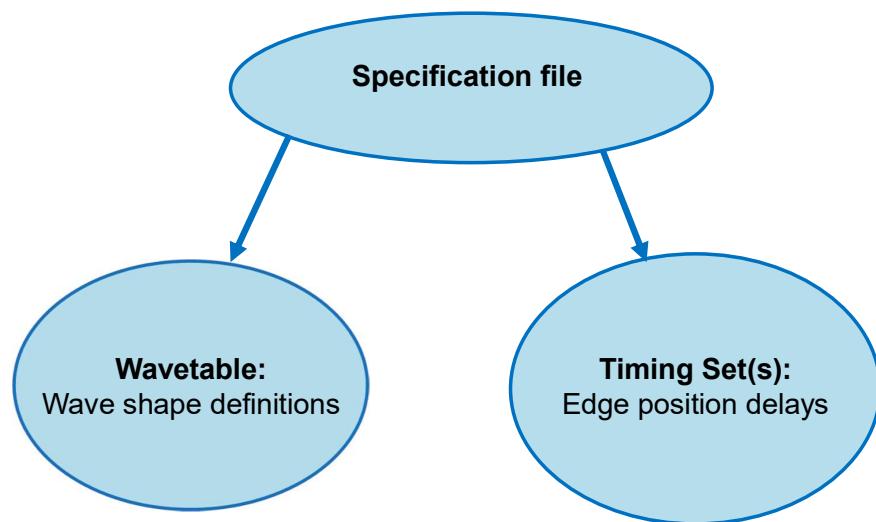


## Window Compare

WL	compare to low
WH	compare to high
WM	compare to intermediate
WX	don't care (mask)
WU	compare to unstable
WC	close Window



# Timing definition



# Example of Signals in a Specification File

```

set timingExample;
setup digInOut gDrive + gReceive
{
    // digital setup: wavetable
    wavetable defaultWvt {
        xModes = 1 {
            0: d1:0;
            1: d1:1;
            L: d1:Z r1:L;
            H: d1:Z r1:H;
            X: d1:Z r1:X;
        }
    }
    // digital setup: timing
    set timing timingExample {
        period = per;
        d1 = 0.0 ns;
        r1 = per/2.0;
    }
}

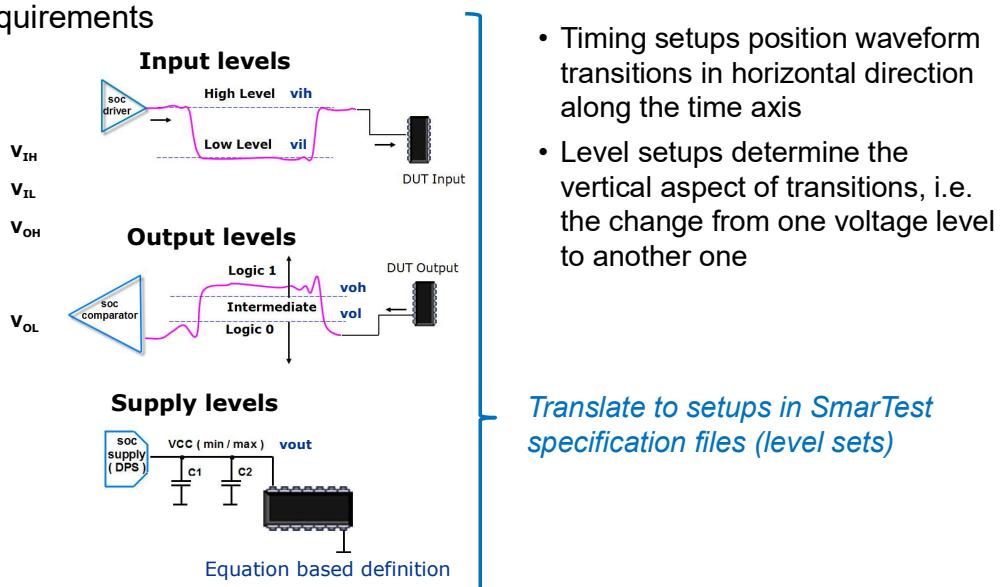
```

Annotations pointing to specific parts of the code:

- Signal names / groups**: Points to `gDrive + gReceive`
- Instrument**: Points to `digInOut`
- Wavetable name**: Points to `defaultWvt`
- Waveform shapes**: Points to the `xModes` block.
- State character names**: Points to the state characters `0`, `1`, `L`, `H`, and `X`.
- Edge delay: timing set**: Points to the `timing` block.

## Level Setup: Purpose

- Device requirements



- Timing setups position waveform transitions in horizontal direction along the time axis
- Level setups determine the vertical aspect of transitions, i.e. the change from one voltage level to another one

*Translate to setups in SmarTest specification files (level sets)*

# Level Setup Purpose

Device requirements → V93000 level setup realization

DC Electrical Characteristics for ACT

Symbol	Parameter	V <sub>CC</sub> (V)	T <sub>A</sub> = 25 °C		T <sub>A</sub> = -40 °C to +85 °C		Units	Conditions
			Typ	Guaranteed Limits	Typ	Guaranteed Limits		
$V_{IH}$	Minimum HIGH Level Input Voltage	4.5	1.5	(2.0)	2.0		V	$V_{CC} > 3.4V$ or $V_{CC} = 0.1V$
		5.5	1.5	(2.0)	2.0			
$V_{IL}$	Maximum LOW Level Input Voltage	3.0	1.5	(0.8)	0.8		V	$V_{CC} < 0.1V$ or $V_{CC} = 0.1V$
		4.5	1.5	(0.8)	0.8			
$V_{OH}$	Minimum HIGH Level	4.5	4.49	(4.4)	4.4		V	$I_{OUT} < 50\mu A$
		5.5	5.49	(5.4)	5.4			
		4.5	0.0001	3.86	3.76	V	$V_{IN} = V_{IL} \text{ or } V_{IH}$ , $I_{OH} = 24\text{ mA}$	
		5.5	0.0001	4.86	4.76			
$V_{OL}$	Maximum LOW Level Output Voltage	4.5	0.001	0.1	0.1	V	$I_{OUT} = 50\mu A$	
		5.5	0.001	(0.1)	0.1			
		4.5	0.36	0.36	0.44	V	$V_{IN} = V_{IL} \text{ or } V_{IH}$ , $I_{OL} = 24\text{ mA}$ , $I_{OL} = 24\text{ mA}$ (Note 5)	
		5.5	0.36	0.44				
$I_{IN}$	Maximum Input Leakage Current	5.5		±0.1	±1.0	µA	$V_1 = V_{CC} \text{ or GND}$	
$I_{ICCT}$	Maximum $I_{C}/I_{IN}$	5.5	0.6		1.5	mA	$V_1 = V_{CC} - 2.1V$	
$I_{OLD}$	Minimum Dynamic Output Current (Note 6)	5.5			75	mA	$V_{OL} = 1.85V \text{ Max}$	
$I_{OHD}$	Output Current (Note 6)	5.5			-75	mA	$I_{OHD} = 3.85V \text{ Min}$	
$I_{QC}$	Maximum Quiescent Supply Current	5.5		4.0	40.0	µA	$V_{IN} = V_{CC} \text{ or GND}$	
$I_{IOZT}$	Maximum I/O Leakage Current	5.5		±0.3	±3.0	µA	$V_1 (OE) = V_{IL}, V_{IH}$ , $V_1 = V_{CC}, \text{ GND}$ , $V_0 = V_{CC}, \text{ GND}$	

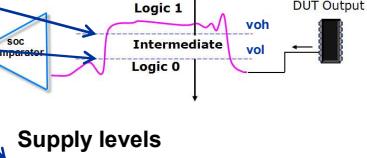
Note 5: All outputs loaded; thresholds on input associated with output under test.

Note 6: Maximum test duration 2.0 ms, one output loaded at a time.

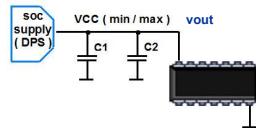
Input levels



Output levels

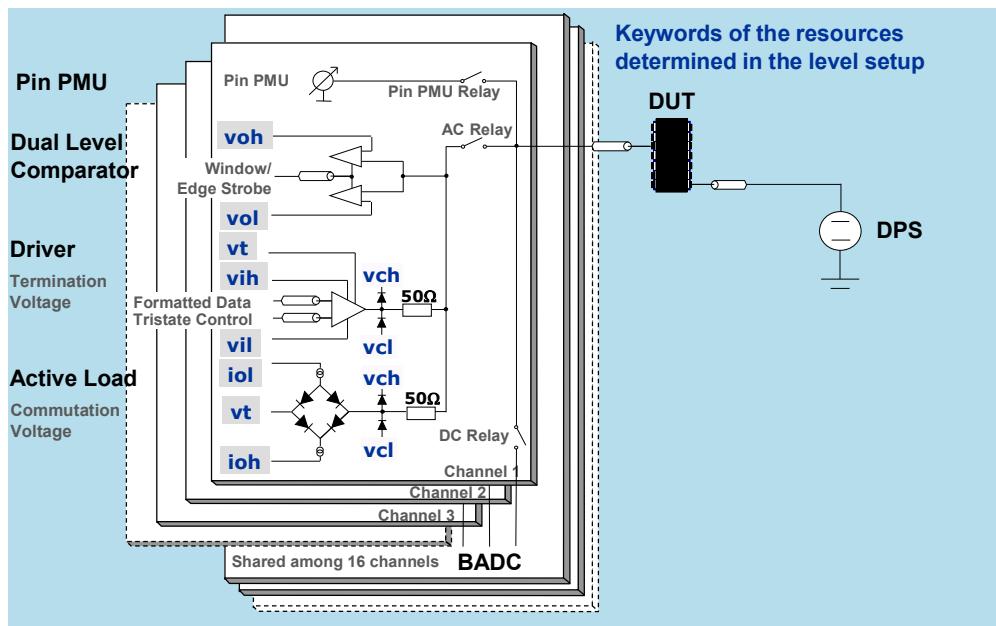


Supply levels



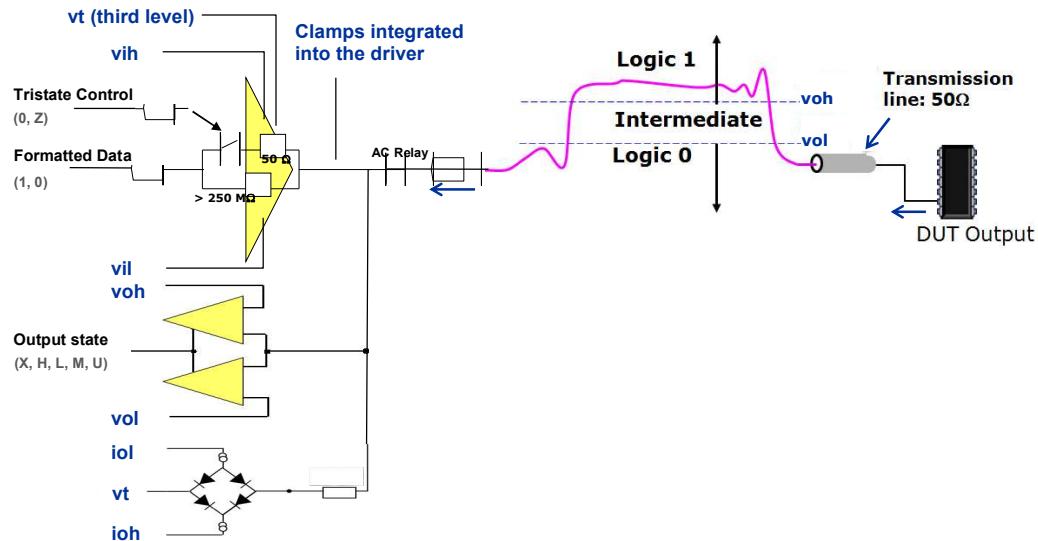
Equation based definition

## SmartScale I/O Level Resources



# Pin Electronics

To handle overshooting or signal reflections, for example proper termination or clamping can help.



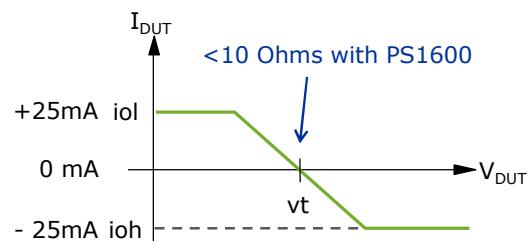
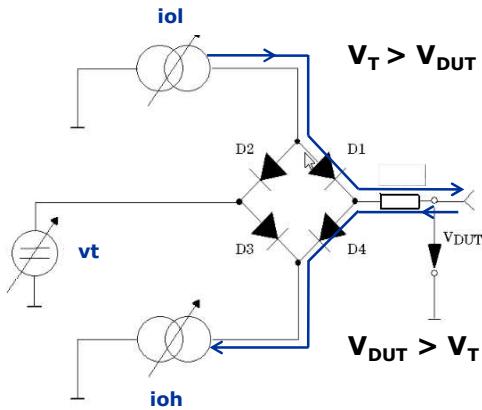
## Termination

Termination modes for a PS1600 test head channel

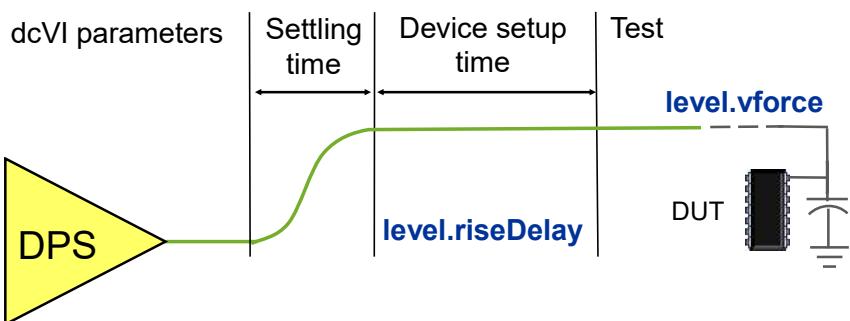
Mode	Parameters to declare
High-impedance termination (default)	$v_{oh}$ , $v_{ol}$ , $v_{ch}$ , $v_{cl}$ , $\text{term}=\text{hiz};$
50 Ohm driver termination with specified voltage	$v_{oh}$ , $v_{ol}$ , $v_{ch}$ , $v_{cl}$ , $v_t$ , $\text{term}=\text{R50Ohm};$
Driver high termination with 50 Ohm	$v_t = v_{ih}$ , $\text{term}=\text{R50Ohm}$
Driver low termination with 50 Ohm	$v_t = v_{il}$ , $\text{term}=\text{R50Ohm}$
Active load	$v_{oh}$ , $v_{ol}$ , $v_{ch}$ , $v_{cl}$ , $v_t$ , $i_{oh}$ , $i_{ol}$ , $\text{term}=\text{load};$
Differential center tap termination	$v_{oh}$ , $v_{ol}$ , $v_{th}$ , $\text{term}=\text{cttap};$

# Active Load

- $i_{ol}$ : source load current for low output levels
- $i_{oh}$ : sink load current for high output levels
- $vt$ : commutation voltage



## Device Power Supply (DPS): Settings of the dcVI Instrument



```
// power supply setup
setup dcVI VCC {
    level.vforce = 1.0 V;
    level.vrange = 1.8 V;
    level.iclamp = 400 mA;
    level.riseDelay = 1 us;
    disconnectMode = safe;
}
```

**Limit connect current:** `level.iclamp`; prevents destruction of the DUT

**Mode of disconnect:** `disconnectMode = hiz`; high impedance  
`disconnectMode = loz`; drive 0V up to ilimit  
`disconnectMode = safe`; ramp to 0V, then hiz

**Other settings for the DC Scale DPS128/64:** `level.iclamp`, `level.iclampSink`, `level.iclampSource`, `level.vrange`, `level.irange`, etc...

# Summary - What you should have learned

- To run patterns or protocol transactions, timing and level specifications given in the datasheet of the DUT need to be translated to level and timing setups in SmarTest based on the *digInOut instrument*.
- The timing setup is based on cycles running with a user defined period and a wavetable defines the available waveforms for a cycle.
- A waveform of a cycle is described by various types of edge actions to drive or expect logical values that are performed at user defined time points during the cycle.
- The level setup for a digital test is about setting the voltage levels for the logical values driven or expected.
- The pin electronic of a digital channel provides various types of termination and allows to program clamps and active loads.
- Power supplies are set up with the *dcVi instrument*, in particular with the level group of the instrument properties.
- All the settings mentioned are stored in *specification files*.



January 2020

All Rights Reserved - ADVANTEST CORPORATION



Basics of Level and Timing Setups - 19



# Specification Files – Digital Setups

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest Specification Files - Digital Setups – 1

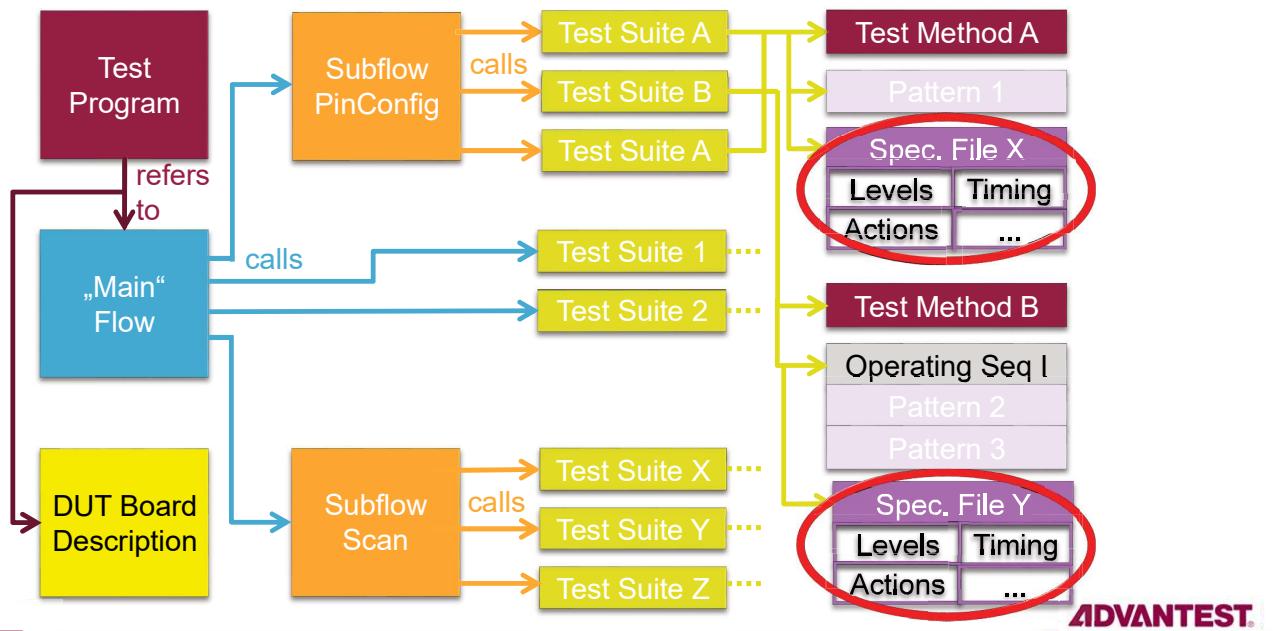
# Learning Objectives

- Understand the purpose of the *specification files*
- Know and understand how to define in *specification files* settings for:
  - initial tester instrument setups before the test execution
  - *actions*, i.e. commands for tester channels during the test execution
  - level and timing setups for digital patterns
- For the settings above, know and understand how to
  - use variables and equations
  - define signal groups
- Understand the import mechanism of *specification files*

# Agenda

- Purpose and content of the *specification files*.
- Examples of digital setups in the *specification files*.
- Import mechanism of the *specification files*.
- Creating a new *specification file*.

# Test Program in SmarTest 8: Building Blocks



## Purpose of Specification Files

When a test program is executed,

- one or several testflows are called;
- the testflows define the order of execution of various test suites;
- the executed test suites usually perform *measurements*;
- for performing *measurements* the tester instruments must be configured and controlled.

The purpose of the specification files is:

**Define tester instrument setups used for a *measurement* execution.**

A test program might contain hundreds of different *specification files* because different settings for *measurements* are needed in a test program.

For example, the same signal driven by a PS1600 test head channel

- can be configured as a digital pin in *specification file X*,
- can be configured as a power supply pin in *specification file Y*,
- can be configured as a clock in another *specification file Z*.

# Introduction: Example of a Digital Test

The following example of a *specification file* contains an almost complete setup for a digital test – only a digital pattern is missing.

Patterns (and *operating sequences*) are described in dedicated files with suffix „\*.pat“ (and „\*.seq“, respectively).

The example *specification file* sets up four signals as *digInOut instruments*.

Therefore, all tests/*measurements* based on this file:

- Will run with exactly these four signals (and no other signals).
- Will use these signals as digital inputs/outputs (no other *instrument* type).

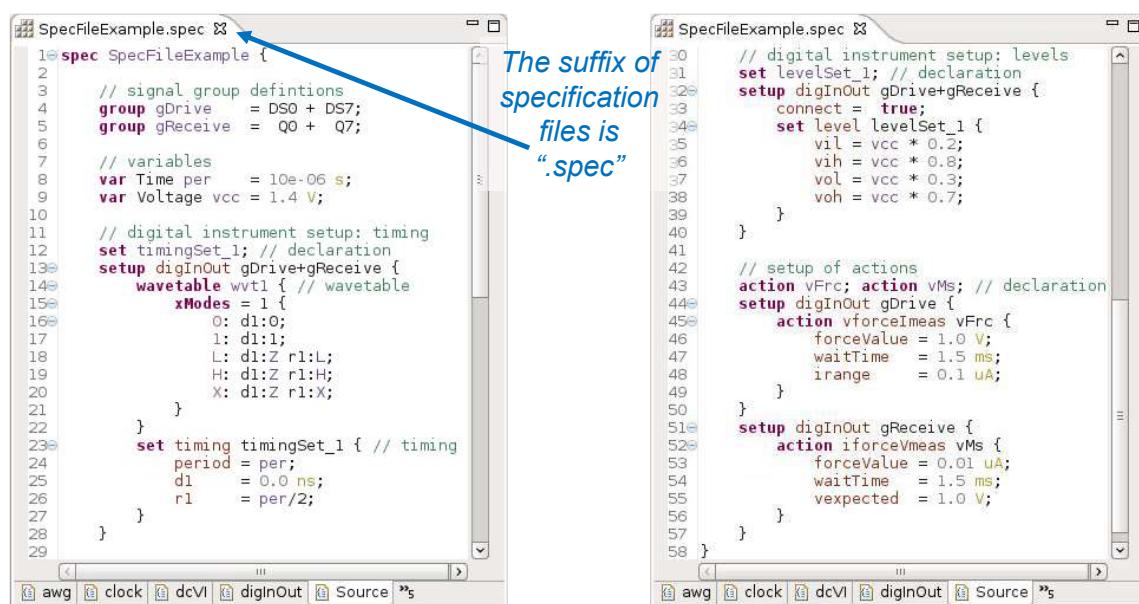
Additionally, the *specification file* defines two *actions* for the four signals.

An *action* is a command for an *instrument* that can be triggered during *measurement execution*.

As shown in the example, *digInOut instruments* can perform DC *actions*:

Force a voltage and measure the current and vice versa.

## Example of a Specification File for a Digital Setup



```
SpecFileExample.spec
1 spec SpecFileExample {
2   // signal_group definitions
3   group gDrive = DSO + DS7;
4   group gReceive = Q0 + Q7;
5
6   // variables
7   var Time per = 10e-06 s;
8   var Voltage vcc = 1.4 V;
9
10  // digital instrument setup: timing
11  set timingSet_1; // declaration
12  setup digInOut gDrive+gReceive {
13    wavetable wvt1 { // wavetable
14      xModes = 1 {
15        0: d1:0;
16        1: d1:1;
17        L: d1:Z r1:L;
18        H: d1:Z r1:H;
19        X: d1:Z r1:X;
20      }
21    }
22    set timing timingSet_1 { // timing
23      period = per;
24      d1 = 0.0 ns;
25      r1 = per/2;
26    }
27  }
28
29 }

// digital instrument setup: levels
30 // declaration
31 setup digInOut gDrive+gReceive {
32   connect = true;
33   set level levelSet_1 {
34     vil = vcc * 0.2;
35     vih = vcc * 0.8;
36     vol = vcc * 0.3;
37     voh = vcc * 0.7;
38   }
39
40
41
42 // setup of actions
43 action vFrc; action vMs; // declaration
44 setup digInOut gDrive {
45   action vforceImeas vFrc {
46     forceValue = 1.0 V;
47     waitTime = 1.5 ms;
48     irange = 0.1 uA;
49   }
50
51 setup digInOut gReceive {
52   action iforceVmeas vMs {
53     forceValue = 0.01 uA;
54     waitTime = 1.5 ms;
55     vexpected = 1.0 V;
56   }
57
58 }
```

# Example: User Defined Names

```

1@ spec SpecFileExample {
2
3    // signal group definitions
4    group gDrive = DSO + DS7;
5    group gReceive = Q0 + Q7;
6
7    // variables
8    var Time per = 10e-06 s;
9    var Voltage vcc = 1.4 V;
10
11   // digital instrument setup: timing
12   set timingSet_1; // declaration
13   setup digInOut gDrive+gReceive {
14     wavetable wvtl; // wavetable
15     xModes = 1 {
16       0: d1:0;
17       1: d1:1;
18       L: d1:Z r1:L;
19       H: d1:Z r1:H;
20       X: d1:Z r1:X;
21     }
22   }
23   set timing timingSet_1 { // timing
24     period = per;
25     d1 = 0.0 ns;
26     r1 = per/2;
27   }
28 }

```

```

30 // digital instrument setup: levels
31 set levelSet_1; // declaration
32 setup digInOut gDrive+gReceive {
33   connect = true;
34   set level levelSet_1 {
35     vil = vcc * 0.2;
36     vih = vcc * 0.8;
37     vol = vcc * 0.3;
38     voh = vcc * 0.7;
39   }
40
41 // setup of actions
42 action vFrc action vMs; // declaration
43 setup digInOut gDrive {
44   action vforceMeas vFrc {
45     forceValue = 1.0 V;
46     waitTime = 1.5 ms;
47     irange = 0.1 uA;
48   }
49 }
50 setup digInOut gReceive {
51   action iforceVmeas vMs {
52     forceValue = 0.01 uA;
53     waitTime = 1.5 ms;
54     vexpected = 1.0 V;
55   }
56 }
57
58

```

## Example: Details (1/2)

```

1@ spec SpecFileExample {
2
3    // signal group definitions
4    group gDrive = DSO + DS7;
5    group gReceive = Q0 + Q7;
6
7    // variables
8    var Time per = 10e-06 s;
9    var Voltage vcc = 1.4 V;
10
11   // digital instrument setup: timing
12   set timingSet_1; // declaration
13   setup digInOut gDrive+gReceive {
14     wavetable wvtl; // wavetable
15     xModes = 1 {
16       0: d1:0;
17       1: d1:1;
18       L: d1:Z r1:L;
19       H: d1:Z r1:H;
20       X: d1:Z r1:X;
21     }
22   }
23   set timing timingSet_1 { // timing
24     period = per;
25     d1 = 0.0 ns;
26     r1 = per/2;
27   }
28 }

```

*Name of the specification (file)*

*Definition of signal groups*

*Definition of variables*

*Declaration of a timing set*

*These signals are used as digital instruments.  
An assignment to another instrument is not allowed in the file*

*Definition of a wavetable for the digital instrument*

*Definition of the timing for the wavetable using a timing set*

## Example: Details (2/2)

```

30 // digital instrument setup: levels
31 set levelSet_1; // declaration
32 setup digInOut gDrive+gReceive {
33   connect = true;
34   set level levelSet_1 {
35     vil = vcc * 0.2;
36     vih = vcc * 0.8;
37     vol = vcc * 0.3;
38     voh = vcc * 0.7;
39   }
40
41 // setup of actions
42 action vFrc; action vMs; // declaration
43 setup digInOut gDrive {
44   action vforceImeas vFrc {
45     forceValue = 1.0 V;
46     waitTime = 1.5 ms;
47     irange = 0.1 UA;
48   }
49 }
50
51 setup digInOut gReceive {
52   action iforceVmeas vMs {
53     forceValue = 0.01 UA;
54     waitTime = 1.5 ms;
55     vexpected = 1.0 V;
56   }
57 }
58 }
```

*Declaration of a level set*

*Connect instrument*

*Definition of voltage levels for the digital instrument using a level set*

*Declarations of actions*

*Separate settings for the two different signal groups*

*Definition of an action*

*Actions are commands sent to an instrument during measurement execution*

*Definition of an action*

## Import Mechanism of Specification Files

In SmarTest, *specification files* can import (the content of) other *specification files*.

Thus, specification data can be split up between multiple files.

*Import statements must be placed at the top of the specification file.*

This is a modification of the first example: The group and variable definitions, the timing and the level settings are imported from three other files.

The content is the same.

```

1 import slideExamples.specFiles.GroupsAndVars;
2 import slideExamples.specFiles.Timings;
3 import slideExamples.specFiles.Levels;
4
5 spec SpecFileExampleImport {
6
7   // set values of variables
8   per = 10e-06 s;
9   vcc = 1.4 V;
10
11  // declarations and setup of actions
12  action vFrc;
13  action vMs;
14  setup digInOut gDrive {
15    action vforceImeas vFrc {
16      forceValue = 1.0 V;
17      waitTime = 1.5 ms;
18      irange = 0.1 UA;
19    }
20  }
21  setup digInOut gReceive {
22    action iforceVmeas vMs {
23      forceValue = 0.01 UA;
24      waitTime = 1.5 ms;
25      vexpected = 1.0 V;
26    }
27  }
28 }
```

# Splitting the Setup into Four Files: Source Code

```

1 import slideExamples.specFiles.GroupsAndVars;
2 import slideExamples.specFiles.Timings;
3 import slideExamples.specFiles.Levels;
4
5 spec SpecFileExampleImport {
6     // set values of variables
7 }
```

```

1 import slideExamples.specFiles.GroupsAndVars;
2
3 spec Timings {
4     // digital instrument setup: timing
5     set timingSet_1; // declaration
6     setup digInOut gDrive+gReceive {
7         wavetable wvtl { // wavetable
8             xModes = 1 {
9                 0: d1:0;
10                1: d1:1;
11                L: d1:Z r1:L;
12                H: d1:Z r1:H;
13                X: d1:Z r1:X;
14            }
15            set timing timingSet_1 { // timing
16                period = per;
17                d1 = 0.0 ns;
18                r1 = per/2;
19            }
20        }
21    }
22 }
```

```

1 import slideExamples.specFiles.GroupsAndVars;
2
3 spec Levels {
4     // digital instrument setup: levels
5     set levelSet_1; // declaration
6     setup digInOut gDrive+gReceive {
7         connect = true;
8         set level levelSet_1 {
9             vil = vcc * 0.2;
10            vih = vcc * 0.8;
11            vol = vcc * 0.3;
12            voh = vcc * 0.7;
13        }
14    }
15 }
16 }
```

```

1 spec GroupsAndVars {
2
3     // signal group definitions
4     group gDrive = D50 + DS7;
5     group gReceive = Q0 + Q7;
6     group gDigInOut = gDrive + gReceive;
7
8     // declaration of variables
9     var Time per;
10    var Voltage vcc;
11 }
```

## Example Setup Split into Four Files

The tool *setup dependency graph* visualizes the file dependencies:

### *“SpecFileExampleImport”:*

- referenced from measurement
- imports other three files
- assigns values to variables
- defines actions
- might include result handling

### *“Timings”:*

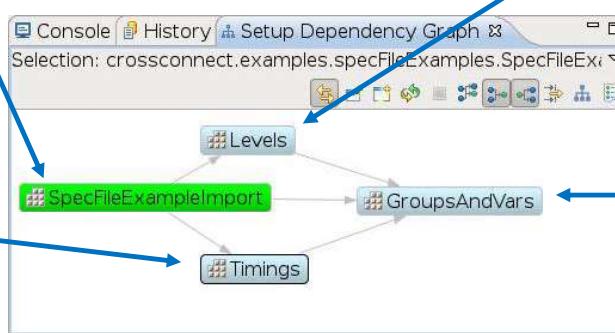
- defines wavetable and timing
- uses variables in equations

### *“Levels”:*

- defines levels
- uses variables in equations

### *“GroupsAndVars”:*

- imported by all other three files
- defines signal groups
- declares variables
- might include: signal mapping



# Import: Facilitates Reuse of Specification Files

The import mechanism facilitates reuse:

For example, the files left can be reused for other *measurements*, that use the same level settings but different settings for the timing.

The “import” statement needs as parameter the fully qualified name of the *specification file* to be imported.

*Fully qualified names are a concept of Java:  
The path is relative to the source folders,  
using “.” for folder hierarchies.  
It is without a file suffix.*

```
1@ import slideExamples.specFiles.GroupsAndVars;
2
3 spec levels {
4     / digital instrument setup: levels
5     set levelSet_1; // declaration
6     setup digInOut gDrive+gReceive {
7         connect = true;
8         set level levelSet_1 {
9             vil = vcc * 0.2;
10            vih = vcc * 0.8;
11            vol = vcc * 0.3;
12            voh = vcc * 0.7;
13        }
14    }
15 }
```

```
1@ spec GroupsAndVars {
2
3     // signal group definitions
4     group gDrive = DSO + DS7;
5     group gReceive = Q0 + Q7;
6     group gDigInOut = gDrive + gReceive;
7
8     // declaration of variables
9     var Time per;
10    var Voltage vcc;
11 }
```

# Overwriting and Evaluation of Setups

```
1@ spec OverwriteExample {
2     set timingSet_1;
3     setup digInOut DSO + DS7 {
4         set timing timingSet_1 {
5             period = 20.0 ns;
6             // d1 = 1 / 2 ns           // error
7             d1 = 1 / 2 * 1 ns;      // = 0ns
8             r1 = 1 / 2.0 * 10 ns;   // = 5ns
9             r2 = 0.5 * 20 ns;      // = 10ns
10        }
11    }
12    setup digInOut DS7 + Q7 - Q7 + Q7 {
13        set timing timingSet_1 {
14            period = 20.0ns;
15            d1 = 1 ns / 2;          // =500ps
16            r1 = 1 / 2.0 * 20 ns;   // = 10ns
17            r2 = 0.5 * 40 ns;       // = 20ns
18        }
19    }
20 }
```

*Instrument setups can overwrite other setups:  
The last setting in the file will overwrite other  
settings (includes settings via “import”) before.*

*Example:  
The settings for “DS7”  
given in lines 4-10  
are overwritten by lines 13-19.*

*Note:  
Terms to define groups are evaluated from left  
to right*

Evaluation of terms/equation:

- Terms that contain only integers are evaluated as integers (“1/2” equals “0”)
- Values with units are of type “double” (“1ns/2” equals “0.5ns”)
- “1/2ns” is evaluated as “1/(2ns)”

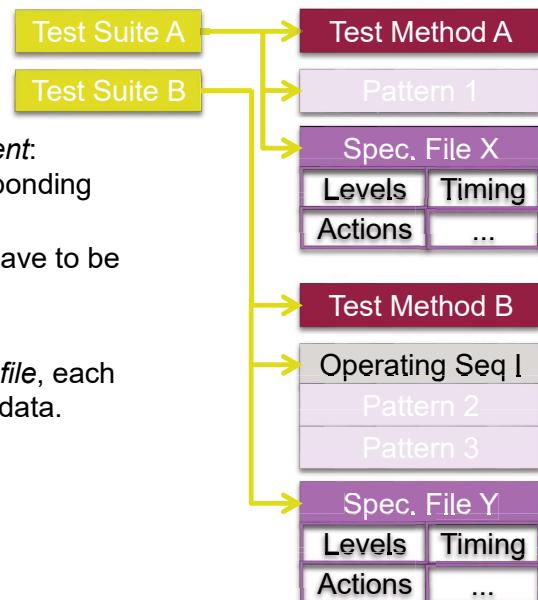
# Setup Data of Specification Files is Local

A *specification file* must be defined for any *measurement*.

The content of the *specification file* is read and added to the setup data of a *measurement* to define the following:

- Which signals are participating in the specific *measurement*: Only those that are set up as an *instrument* in the corresponding *specification file*.
- How the test head channels connected to these signals have to be configured for the *measurement*.

Even if multiple *measurements* use the same *specification file*, each setup of these *measurements* maintain its own local set of data.



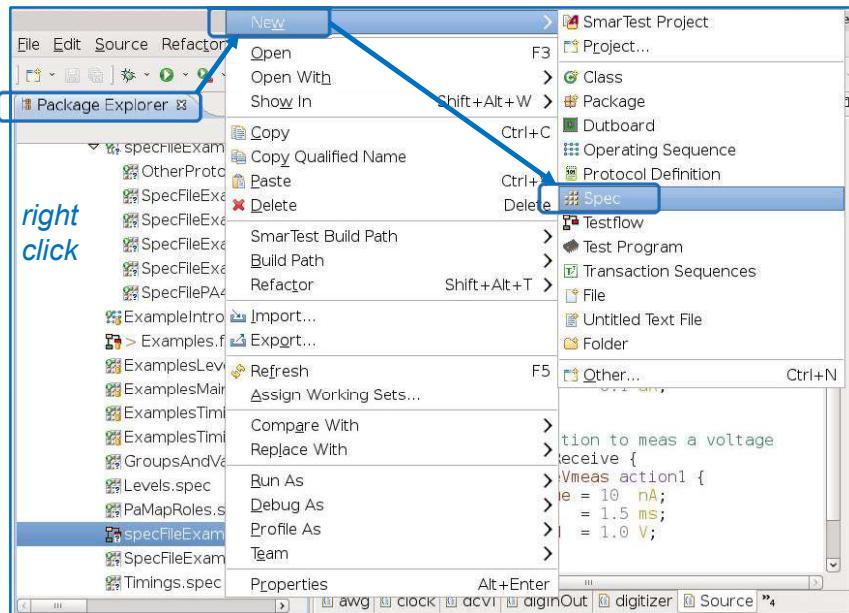
## Signal Group Editor

Signal Groups

Group members

The screenshot shows the Signal Group Editor window. On the left, a list of signal groups is displayed, including driveGroup, receiveGroup, allInputs, allOutputs, and allRs. Below this list is a detailed list of individual signals: eveninputs, oddinputs, gD00\_D03, gR00\_R03, gD04\_D07, gR04\_R07, gD08\_D11, gR08\_R11, gD12\_D15, gR12\_R15, gD00\_D07, gR00\_R07, gD08\_D15, and gR08\_R15. On the right, a grid table titled 'Signals | Groups' shows the mapping of signals to groups. The grid has columns for D00, D01, D02, D03, D04, D05, D06, D07, D08, D09, D10, D11, D12, D13, D14, D15, R00, R01, R02, R03, R04, R05, R06, R07, R08, R09, R10, R11, R12, R13, R14, and R15. The first four rows of the grid are highlighted in orange, while the rest are greyed out. At the top of the editor, there is a 'Filter Signal' input field and two checkboxes: 'Show group members only' (checked) and 'Show enabled signals only'.

# Creating a New Specification File



## Summary - What you should have learned

- The *specification files* contain the setup data of test head channels.
- In specification files, signals are assigned to a certain *instrument*.  
The setup of the *instrument* consists of defining *actions* and setting the properties.
- For the setups in the *specification files* signal groups and variables can be defined and equations can be used.
- To be able to reuse a certain part of the setup data, it can be stored in a dedicated *specification file*. Then this *specification file* can be imported by other *specification files*.
- Typically, the setup data of a digital test is split into the following *specification files*:
  - The file that is referenced from the test suite definition.  
It might contain essential settings for the test suite like assigning values to variables.
  - The file that contains the level settings.
  - The file that contains the timing settings.
  - The file that contains the wavetables if not included in the file above.Additionally, common *specification files* might be included for assigning alias names to signals, for defining signal groups and for declaring variables.



# Specification Files – Multiple Domains

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest Specification Files – Multiple Domains – 1

## Learning Objectives

- Know and understand how to define in *specification files* settings for any domain: digital, DC, MX, RF & PA.
- Know how to
  - define alias names for signals;
  - limit the set of signals participating in a test.



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest Specification Files – Multiple Domains – 2

# Agenda

- Examples of settings in the *specification files* for various *instrument types*.
- Defining alias names for signals.
- Restricting the set of signals participating in the measurement.
- The term of the *main specification file*.
- Editing content of *specification files*: source and tables.
- Editor for signal groups.

## More Examples: Result Details, Clock, RF

```
59 // setup for collected results
60 setup digInOut gReceive {
61     result.cyclePassFail.enabled = true;
62     result.maxCycles = 1000;
63 }
64
65 // setup of a clock that stays active
66 // beyond the end of the measurement
67 setup clock CP {
68     timing.period = 0.1 us;
69     shape = square;
70     keepAlive = true;
71 }
72
73 // setup and usage of waveforms
74 waveform sine sinel0MHz {
75     frequency = 10.0 MHz;
76 }
77 waveform sampleBased SampledWF {
78     dataFile = "mySampledWaveform";
79 }
80 action stimMod; // action declaration
81 setup rfstim SO {
82     action modulated stimMod {
83         frequency = 1.0 GHz;
84         power = 0.0 dBm;
85         waveform = SampledWF;
86     }
87 }
```

Amount and level of detail of collected digital results

Setup of a clock instrument  
The clock continues beyond the end of the measurement

Setup of waveforms:  
by shape or arbitrary  
by waveform-file

Setup of an RF stim instrument  
Setup of an action to generate a stimulus based on a waveform defined above

# More Examples: Signal Alias, Used Signals

```
89 // setup of a measurement execution
90 // restricted to a subset of signals
91 using tck+tdi+tdo+tms+trstn;
92
93 // renaming of signals
94 signal tck = IO1;
95 signal tdi = IO2;
96 signal tdo = IO3;
97 signal tms = IO4;
98 signal trstn = IO5;
99 group jtagGroup = tck+tdi+tdo+tms+trstn;
100
101 // setup for protocol aware test
102 protocolInterface
103     slideExamples.specFiles.Jtag JTAG {
104         // assign signals to signalRoles
105         // of the protocol
106         TCK = tck;
107         TDI = tdi;
108         TDO = tdo;
109         TMS = tms;
110         TRSTN = trstn;
111     }
112 transactSeq jtagWriteExample;
113 setup protocolInterface JTAG {
114     transactSeq
115         slideExamples.Transactions.Write
116         jtagWriteExample();
117 }
```

Setup to restrict the set of signals participating in the measurement

Assignment of alias names to existing signals

Reference to a file containing a protocol definition for „JTAG“

Specification files might contain setup parts of participate in the protocol aware software in the “JTAG” protocol interface

Details are given in the corresponding training units about protocol aware software

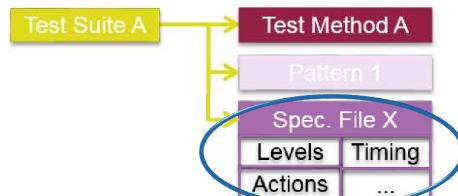
Declaration and setup of a protocol operation (“transaction sequence”)

# Main Specification Files

A test suite typically consists of three parts:

- a test method;
- a pattern or an *operating sequence*;
- a *specification file*.

The example of a testflow file shows this below:



```
1 flow SpecFileExample {
2     setup {
3         suite SpecFilePresentation calls com.advantest.itee.tml.actml.FunctionalTest {
4             measurement.pattern = setupDef(slideExamples.patterns.Small);
5             measurement.specification = setupRef(slideExamples.specFiles.SpecFileExample);
6         }
7     }
8     execute {
9         SpecFilePresentation.execute();
10    }
11 }
```

Specification file assigned to the measurement of test suite “SpecFilePresentation”

A *specification file*, that contains (probably via “import”) the full specification setup of a measurement/test suite is called a **main specification file**.

# Instruments and Specifications

The **main specification file** must define the *instrument type* for all *signals* to be used in a *measurement*.

For a *signal*, the defined *instrument type* can be switched from one *measurement* to the next one, but not during a *measurement*.

Some test head channels provide features of multiple *instrument types*.

For example, a PS1600 channel can run patterns and measure voltages.

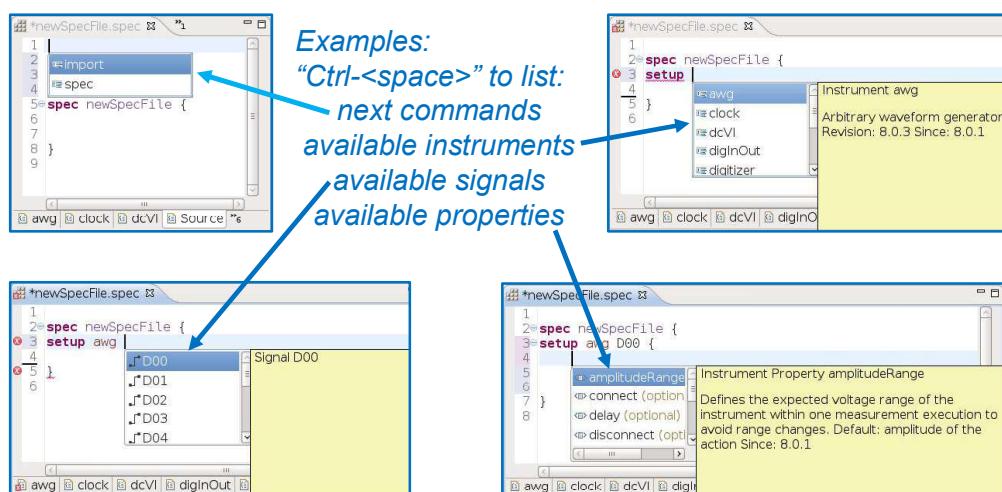
In order to support this, the *digiInOut instrument*

- can run digital patterns,
- and additionally provides the actions “vforceImeas” and “iforceVmeas” like the *dcVI instrument*.

This is an example of overlapping functionalities between different *instrument types*.

## Content Assist in Specification Files

As for other file types of SmarTest 8, content assist (via “Ctrl-<space>”) is provided for the *SmarTest Setup Format (SSF)* of the *specification files*:



# Quick Fixes in Specification Files

Hitting “Ctrl-1” at a problematic location of a file in the *SmarTest Setup Format (SSF)* triggers SmarTest to suggest quick fixes.

*Specification files* are based on *SSF*, so this works as shown below.

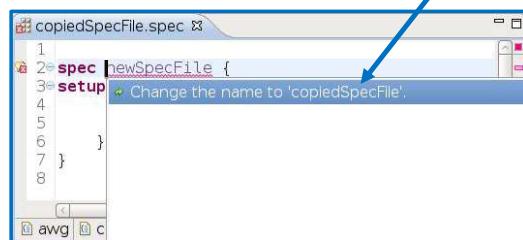
Example:

A *specification file* has been copied.

As a result, the file name does not match the name of the specification.

- Place the cursor at the incorrect name of the specification.
- Press “Ctrl-1”.
- Select the suggested quick fix.

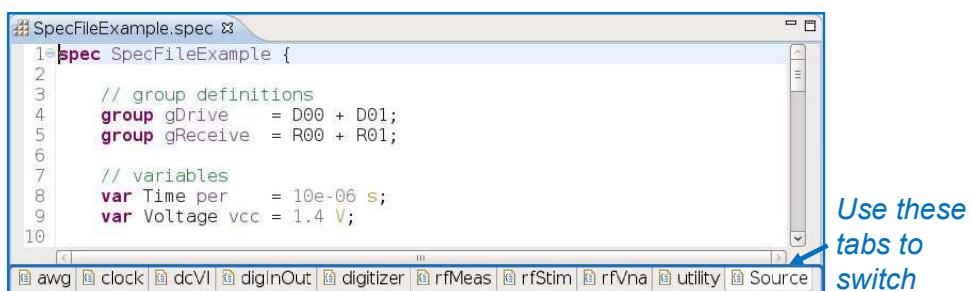
*Example: Select the suggested quick fix*



## Views and Tabs of Specification Files

Use the tabs of the SmarTest 8 editor for the *specification files*

- to work on the source code (shown below) and
  - to view and edit the settings in tables for the various *instruments* (shown on the next slide).
- SmarTest automatically aligns the content of tables and “source”.



## “digInOut” Table: Properties of the Instrument

The tables can be used to view and edit the setup of *instruments*.

It shows only the setups defined in the file and not the content of imported *specification files*.  
For example, here the table for the *digInOut* instrument is shown.

*Three tabs are available for the digInOut instrument*      *The header lists the available properties of the digInOut instrument*

Signal	connect	discon...	keepAlive	maskP...	power...	pulldown	result	capture	cyclePassFail
gDrive+gReceive									
gDrive+gReceive	true								
gDrive									
gReceive									

## “digInOut” Table: Level Sets

Example of the table for the settings of *level sets*. *Timing sets* are handled similar.

In the previous slide the header listed all available *instrument properties*.

Here it lists all available settings for the *level sets*.

This can be very helpful when looking for the appropriate properties or settings to setup an *instrument* as needed for testing a device.

*The header lists all available settings for level sets.*

Signal	Set	ioh	iol	term	vch	vcl	vlh	vll	voh	vol	vt
gDrive+gReceive											
gDrive+gReceive	levelSet_1										
gDrive											
gReceive											

# More Documentation in the TDC

The screenshot shows the SmarTest 8.2.0 Documentation interface. The left sidebar contains a tree view of documentation topics under "SmarTest 8.2.0 Documentation". The main content area is titled "Specification file reference". It includes a "Subtopics" section with links to "Resolving the used signals in the import hierarchy", "Wavetables in the specification file", "Waveforms in the specification file", "Protocol-aware setup elements", "Instrument properties and settings in the specification file", and "List of specification file examples". Below this, a note states: "The measurement specification file ("specification file" for short) contains the Measurement specification information and has the file name extension .spec.". Another note indicates: "The specification file must be saved in the source folder of a project, as described in recommended SmarTest project structure." A "Note" box specifies: "The maximum supported size for specification files is 20000 lines." The page also mentions that it contains information such as: "Declaration of the transaction sequences, sets (timing, level, or fast adjustment), and actions that are".

## Editor for Signal Groups

To open the *signal group editor*, right-click in the source code of the *specification file* and select “Open With” and “Signal Group Editor”.

This tool is useful for setting up and modifying signal groups, in particular for test programs with a large number of signals:

The tool’s capability to filter via regular expressions allows to focus on the relevant signals.

The screenshot shows the Eclipse IDE interface with the "SignalGroups.spec" file open in the editor. The left side shows the "Package Explorer" with various project files like "muder.dcc", "SetupAndDebug.flow", and "SignalGroups.sper". A context menu is open over the "SignalGroups.spec" file, with "Open With" selected. The "Signal Group Editor" option is highlighted. The main editor window displays a table of signals grouped by domain. The table has columns for D00, D01, D02, D03, D04, D05, D06, D07, D08, D09, D10, D11, D12, D13, D14, R00, R01, R02, R03, R04, R05, R06, R07, R08, R09, R10, R11, R12, R13, R14, R15, Utility\_REL1, Utility\_REL2, VDDA, VDDD, rfLoopIn, and rfLoopOut. There are checkboxes for "Filter signal", "Show group members only", and "Show enabled signals only".

# Summary - What you should have learned

- In specification files the instruments of the different domains (digital, DC, MX, RF) are all set up the same way.
- The specification file referenced from a test suite setup is called the main specification file.
- The content of specification files can be viewed and modified in tables, that are available for each instrument type.
- The signal group editor helps to manage signal groups for test programs with many signals.



# Patterns

SmarTest 8.2.5 Training

January 2020

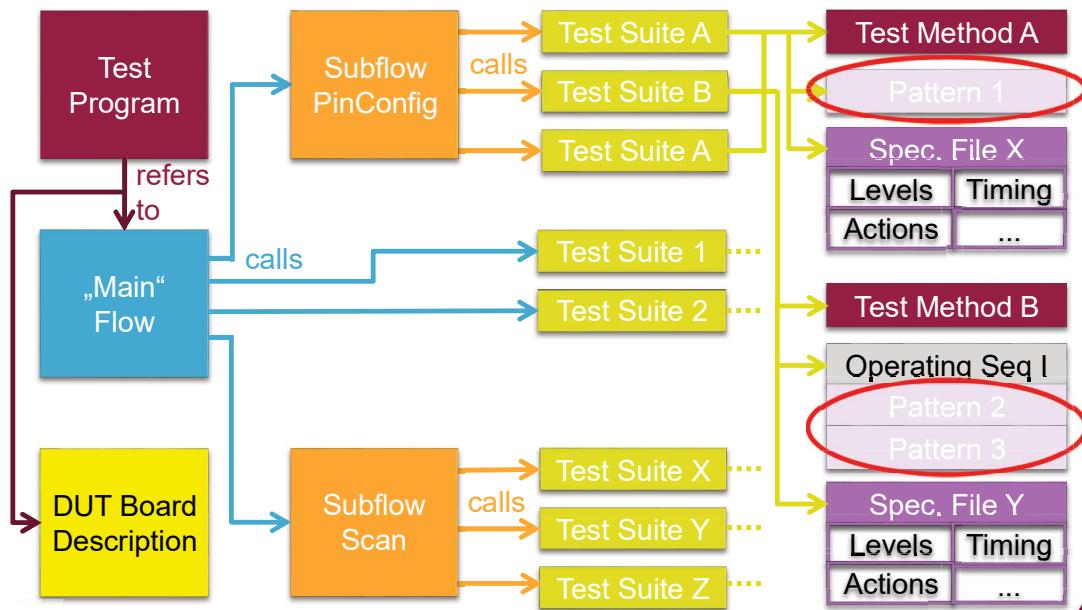
# Learning Objectives

- Understanding the purpose of patterns
- Know and understand the content of pattern files
- Learn how to edit and debug patterns

# Agenda

- Purpose and content of patterns
- Pattern representation in the *pattern editor*
- Editing patterns
- Using the *pattern editor* for debugging

# Test Program in SmarTest 8: Building Blocks



## Basics of Patterns

Patterns contain digital values that are

- applied to the digital inputs and
  - expected at the digital outputs
- of a device under test during a physical measurement.

Usually the values of digital IOs of a pattern are arranged in tables.

The columns are related to the digital IOs.

A row contains values applied or expected for a certain time interval, i.e. the period of a pattern cycle. These values form a (pattern) vector.

Additionally, patterns might contain *sequencer instructions* to be applied during their execution, for example to repeat a certain vector 10 times.

Any single vector is applied for the same time which is called a “pattern cycle”.

However, after a “repeat” or other *sequencer instructions* the vector numbers might be different from the numbers of executed pattern cycles.

# Viewing Patterns in the Pattern Editor

*Columns: Signals defined in the DUT board description file*

Vector#	Instruction	Comment	Actions	Input	Input	Output	Bidir	Input
0	RPTV 1, 10	Repeat vector		0 0 X X 0				
1				0 0 X X P				
2				1 0 H H P				
3				1 0 H H P				
4	IMeasVCC			0 1 H L P				
5	LOOP 20							
6	Repeat loop			0 1 L X P				
7	Repeat loop			0 1 H X P				
8	LOOPEND			0 0 L O P				
				0 1 H I P				

## Patterns in SmarTest 8

For all signals of a SmarTest 8 pattern, the same cycle period and timing settings are applied. A pattern contains for any signal the same number of *state characters*.

For executing a pattern, the *state characters* are mapped to waveforms representing logical values as defined in a *wavetable*.

The *wavetable* must be specified in a *specification file* which is part of the *measurement setup*. That means for correctly interpreting a pattern, the corresponding *wavetable* is always needed.

Therefore it is highly recommended to define *state characters* in a typical way as shown here:

```
wavetable typical {
    xModes = 1 {
        0: d1:0;           // drive 0
        1: d1:1;           // drive 1
        L: d1:Z r1:L;     // expect 0
        H: d1:Z r1:H;     // expect 1
        X: d1:Z r1:X;     // don't care
    }
}
```

Dynamic pattern modifications are downloaded to test head channels just before the next execution.

# Editing Patterns

Patterns can be edited in various ways in the *pattern editor*:

- Editing *state characters*
  - single *state characters*, for example find and replace;
  - all *state characters* of a vector, for example copy and paste;
  - rectangular blocks of *state characters*, for example copy and paste.
- Adding, modifying or removing
  - *sequencer instructions*;
  - *anchors of actions*;
  - comments.
- Adding or removing vectors.
- Adding or removing *signals*.

## Pattern Editor: Edit Cells

Double click into the corresponding position to edit

- single *state characters*;
- *sequencer instructions*;
- comments;
- *anchors of actions*.

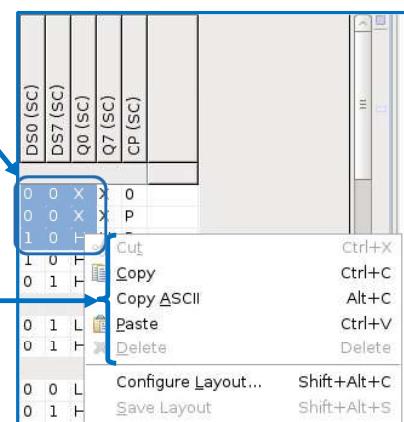
DS0 (SC)	DS7 (SC)	Q0 (SC)	Q7 (SC)	CP (SC)		
0 0 X X 0	0 0 X X P	1 0 H H P	1 0 H I	0 1 H L P	0 1 I Y P	

*Input prompt for a single state character.*

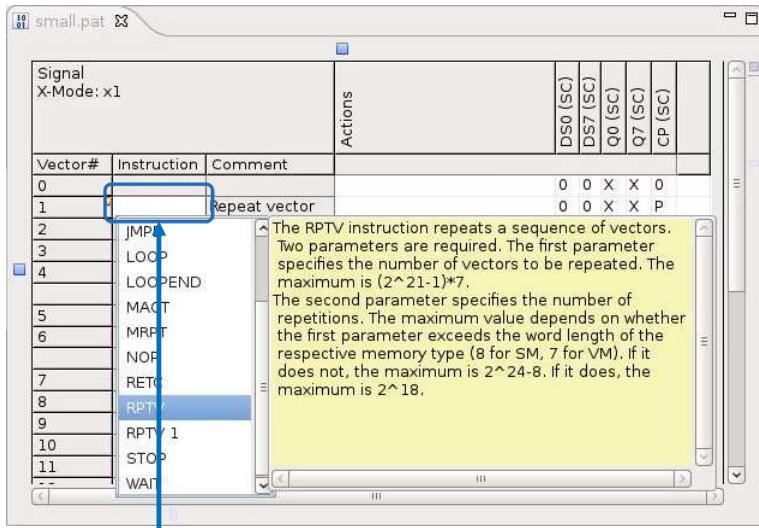
*To work with a block of state characters mark the area*

- *with a mouse click at one corner and shift + mouse click at the opposite corner.*

*Right click on a marked block brings up a menu to edit the block.*



# Pattern Editor: Sequencer Instructions

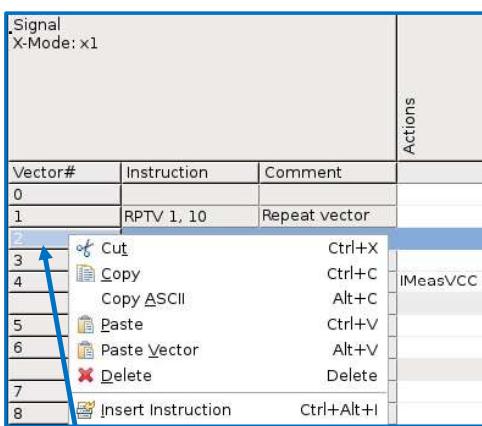


Double click into the “Instruction” column to add, edit or remove a sequencer instruction.

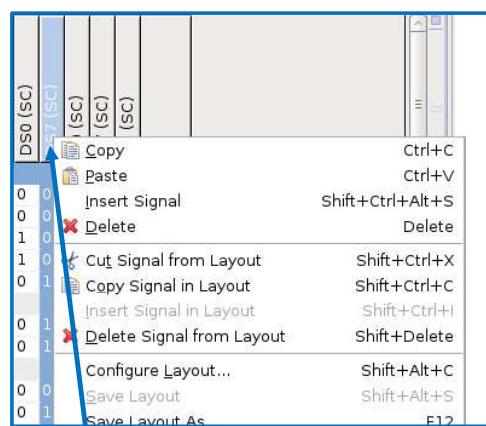
Content assist provides a list of all available sequencer instructions and their descriptions.

Use “Ctrl+Space” to get the list of sequencer instructions.

# Pattern Editor: Edit Vectors and Signals



Right click on a vector brings up a menu with special options to edit vectors.

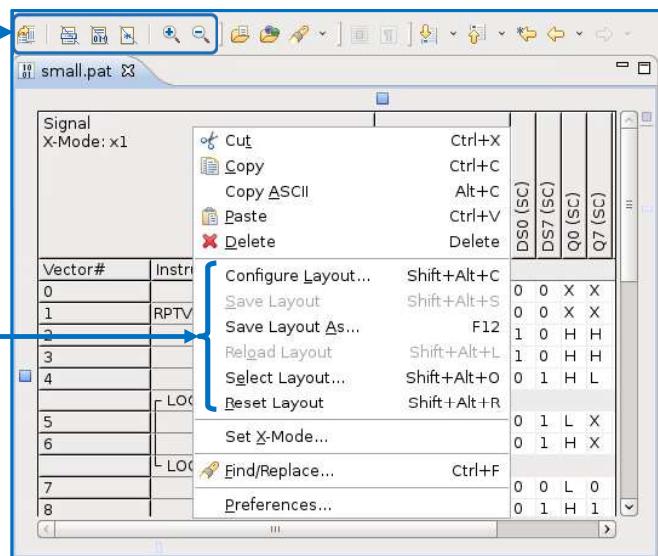


Right click on a signal brings up a menu with special options to edit signals.

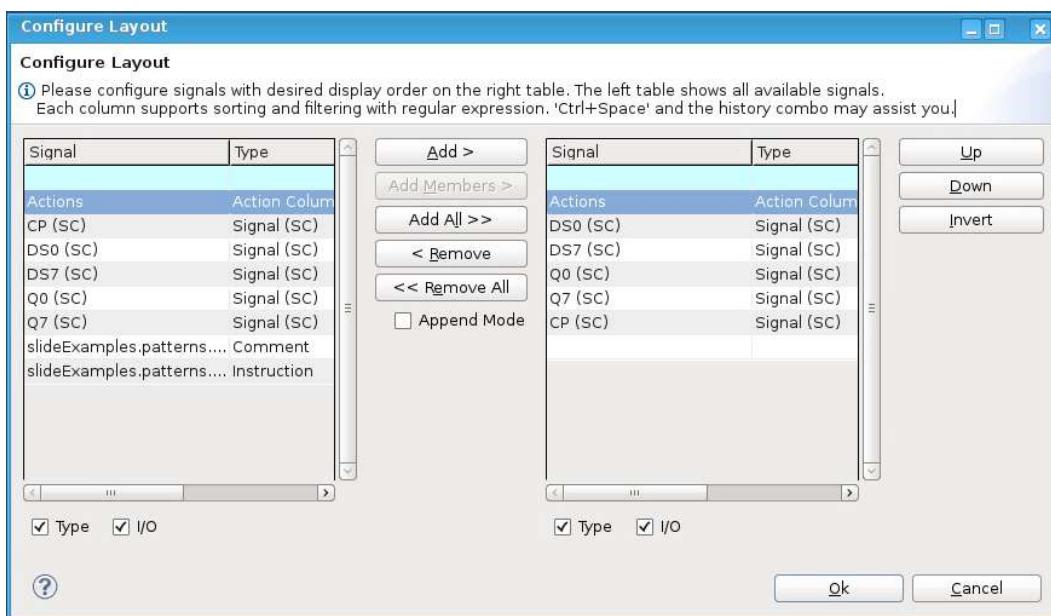
# Pattern Editor: Configure View

Buttons to zoom in or out, remove columns  
or to switch to horizontal pattern view.

Right click brings up a menu that allows to  
modify the layout.



# Pattern Editor: Configure Layout



# Basics: Higher X-Modes

So far the screenshots have shown setups for that the period of patterns cycles matched to the period of tester cycles. This is called *x1-mode*.

For higher *x-modes* multiple pattern cycles are executed in one tester cycle.

For example, in *x2-mode* the waveforms of two *state characters* are executed in one tester cycle, meaning that the usage of edges in a single tester cycle is doubled.

The **benefits** of higher *x-modes* are:

- Less memory is used in the test head cards and
- higher frequencies are feasible for running patterns.

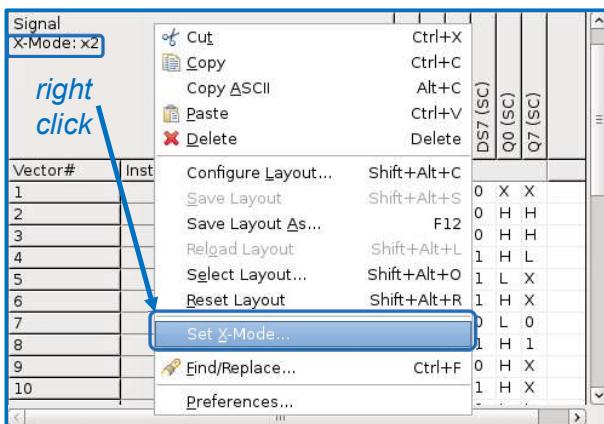
Some **constraints** must be considered for higher *x-modes*, for example:

- The number of vectors in the pattern must be a multiple of the *x-mode*.
- The number of vectors between sequencer instructions must be a multiple of the *x-mode*.
- For each edge type, the maximal number of edges is 8 per tester cycle.
- The hardware limits the number of waveforms to 256.

## Example Setup for x2-Mode

As vector data in pattern files consists of state characters, users can easily change *x-modes*, for example from *x1-mode* to *x2-mode*:

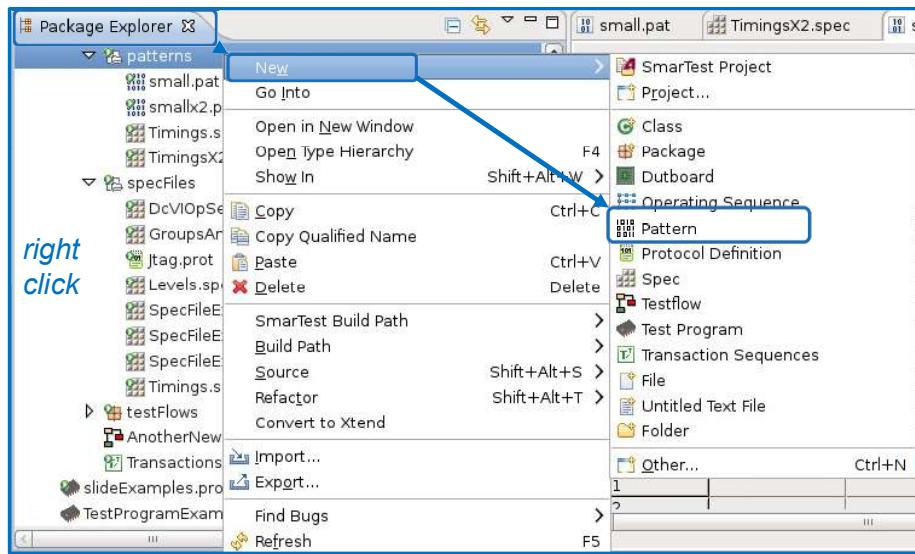
- The pattern must support *x2-mode* (refer to the constraints listed before).
- The *wavetable* of the related *specification file* must define the *x2-mode*.



```
xModes = 2{  
    0: d1:0;      // drive 0  
    1: d1:1;      // drive 1  
    L: d1:Z r1:L; // expect 0  
    H: d1:Z r1:H; // expect 1  
    X: d1:Z r1:X; // don't care  
}
```

A dedicated wavetable for a higher *x-mode* allows to specifically reduce the numbers of needed edges and waveforms in order to meet the constraints given in the previous slide.

# New Pattern



## Debug with the Pattern Editor

In the debug mode, the *pattern editor* shows cycle numbers and fails.

Ensure, that in the specification file collecting pass/fail results per cycle is enabled as follows:

```
result.cyclePassFail.enabled = true;
```

Cycle#	Vector#	Instruction	Comment	DS0 (SC)	DS7 (SC)	Q0 (SC)	Q7 (SC)	CP (SC)
0	0			0 0	X X	0		
1	1	RPTV 1, 10	Repeat vector	0 0	X X	P		
11	2			1 0	H H	P		
12	3			1 0	H H	P		
13	4			0 1	H	P		
		LOOP 20						
14	5		Repeat loop	0 1	L X	P		
15	6		Repeat loop	0 1	H	P		
54	7							
55	8			0 1	H 1	P		

Annotation at the signals show failing signals.

Cycle numbers.

Fail.

Light red fail:  
The fail did not occur in every loop execution.  
Hover the mouse pointer over it to get a (partial) list of failing cycles.

Annotation at the vectors show failing vectors.

# Debug with the Pattern Editor

More details of the *pattern editor* in debug mode.

Cycle numbers differ from vector numbers after the “repeat” sequencer instruction.

Signal X-Mode: x1				Actions	DS0 (SC)	DS7 (SC)	Q0 (SC)	Q7 (SC)	CP (SC)
Cycle#	Vector#	Instruction	Comment		0 0 X X P	0 0 X X P	1 0 H H P	1 0 H H P	0 1 H L P
0	0								
1	1	RPTV 1, 10	Repeat vector						
11	2								
12	3								
13	4								
		LOOP 20							
14	5		Repeat loop		0 L X P				
		LOOPEND							
Fail: Q0 (SC), Cycle: 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39					H X P				
54	7								
55	8								

Hover the mouse pointer over an annotation at  
- signals or  
- vectors  
to get a lists of failing  
signals and cycles.

## Summary - What you should have learned

Here is a list of items you should be able to remember after you completed this module:

- A pattern can be one of the main parts of a test suite setup.
- The *pattern editor* is used to view and edit patterns.
- Patterns can contain:
  - *sequencer instructions*;
  - *state characters*;
  - signals;
  - setting of the *x-mode*;
  - *action anchors* and
  - comments.
- In debug mode, the *pattern editor* shows additionally cycle numbers and fails.
- An *x-mode* is an operating mode in which a single tester cycle contains multiple cycles applied to the device.



# Action, Pattern and Transaction Sequence Calls

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

Pattern, Action and Transaction Sequence Calls - 1

## Learning Objective

- Understand how *actions* invoked from patterns are executed.
- Know how parallel calls of patterns, *actions* and *transaction sequences* are synchronized in *operating sequences*.



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

Pattern, Action and Transaction Sequence Calls - 2

# Agenda

- Patterns:  
Invoking *actions*
  - interruptive and
  - non-interruptive.
- *Operating sequences*:  
Synchronization of
  - patterns
  - *transaction sequences* and
  - *actions*.

## Actions invoked from Patterns

Often tests require, that a certain *action* is started aligned with the execution of a certain vector of a pattern.

This feature is provided via *anchors*, that can be assigned to vectors to call an *action*. Such a pattern might be called from an *operating sequence* or might be directly assigned to a *measurement* object.

The following example executing two patterns is used to show two different scenarios for *actions* called in patterns:

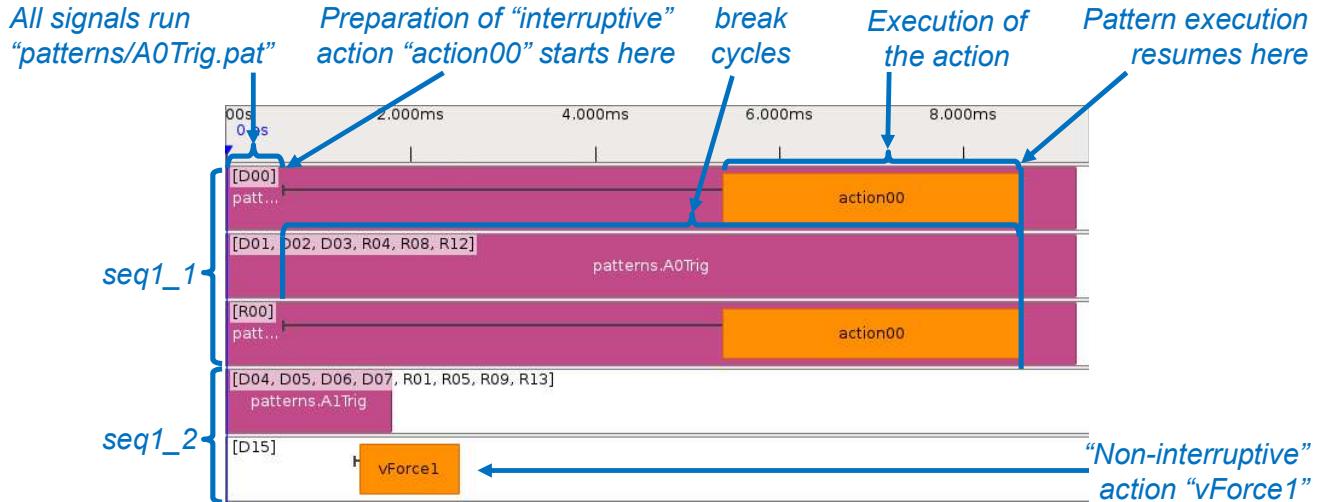
- “patterns.A0Trig” starts *actions* for two of the signals that are part of the pattern execution.
- “patterns.A1Trig” starts an *action* for a signal that is not related to the pattern.

```
1@ sequence PatternTriggerExample {
2@     parallel parGrp1 {
3@         sequential seq1_1 {
4@             patternCall patterns.A0Trig PatternLane0;
5@         }
6@         sequential seq1_2 {
7@             patternCall patterns.A1Trig PatternLane1;
8@         }
9@     }
10@ }
```

Qualified name for a pattern with file path  
“patterns/A1Trig.pat” in the source folder

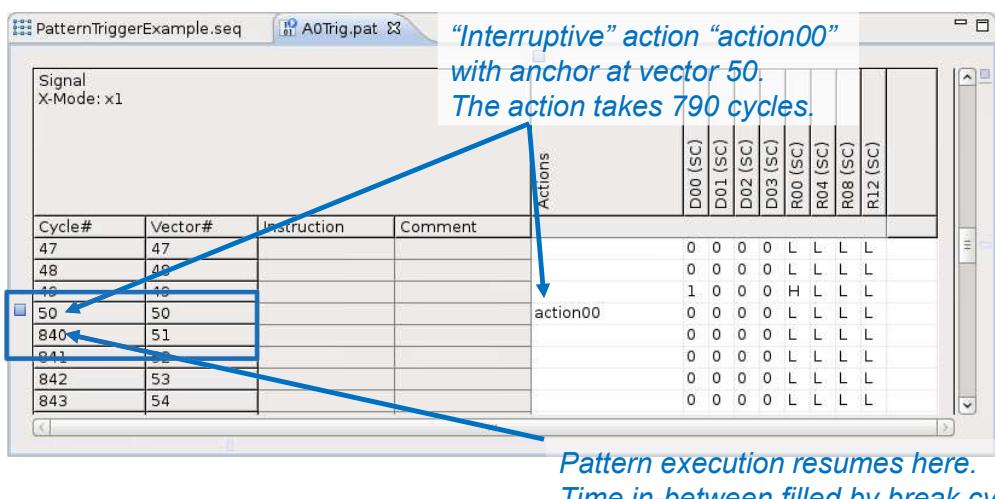
# Example of Patterns Starting Actions

The *operating sequence view* shows *actions* that have been started by *anchors* in the patterns.  
Note, there are “interruptive” and “non-interruptive” *actions*.



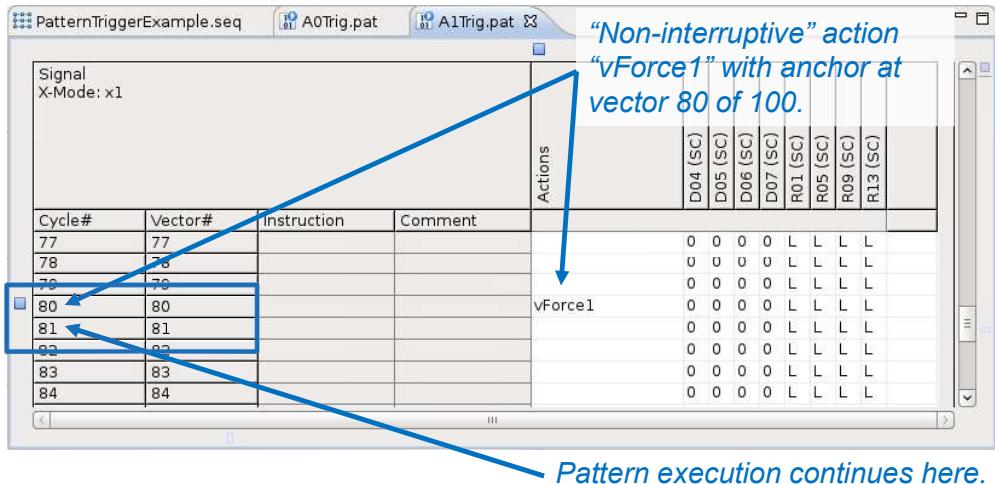
## Pattern Starting an Interruptive Action

The *pattern editor* also helps to distinguish the two cases.  
It displays how the *action* that is specified for signals of the pattern interrupts the pattern execution.



# Pattern Starting a Non-Interruptive Action

The *pattern editor* also helps to distinguish the two cases.  
It shows if the *action* does not interrupt the pattern execution.



# Operating Sequence: Synchronization

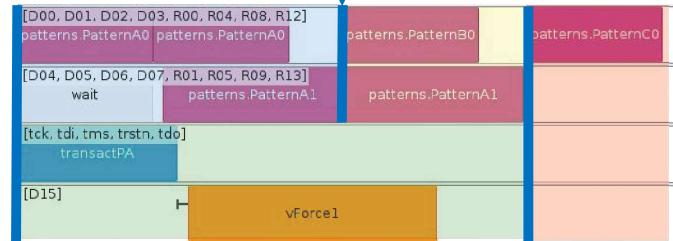
```

1 sequence OpSeqNestingExample {
2   parallel parGrp1 {
3     sequential seq1_1 {
4       parallel parSubGrp1_1 {
5         sequential seq1_1_1 {
6           patternCall patterns.PatternA0 PA0_1;
7           patternCall patterns.PatternA0 PA0_2;
8         }
9         sequential seq1_1_2 {
10           wait 0.6ms;
11           patternCall patterns.PatternA1 PA1_1;
12         }
13       }
14     }
15     parallel parSubGrp1_2 {
16       sequential seq1_2_1 {
17         patternCall patterns.PatternB0;
18       }
19       sequential seq1_2_2 {
20         patternCall patterns.PatternA1 PA1_1;
21       }
22     }
23     sequential seq1_2 {
24       transactSeqCall transactPA;
25       actionCall vForce1;
26     }
27   }
28   parallel parGrp2 {
29     sequential seq2_1 {
30       patternCall patterns.PatternC0 LastPat;
31     }
32 }
33 }
```

The execution of *sequential groups* in a *parallel group* always starts at the same time.

Make sure, that elements - for example two patterns - are started synchronously as follows:

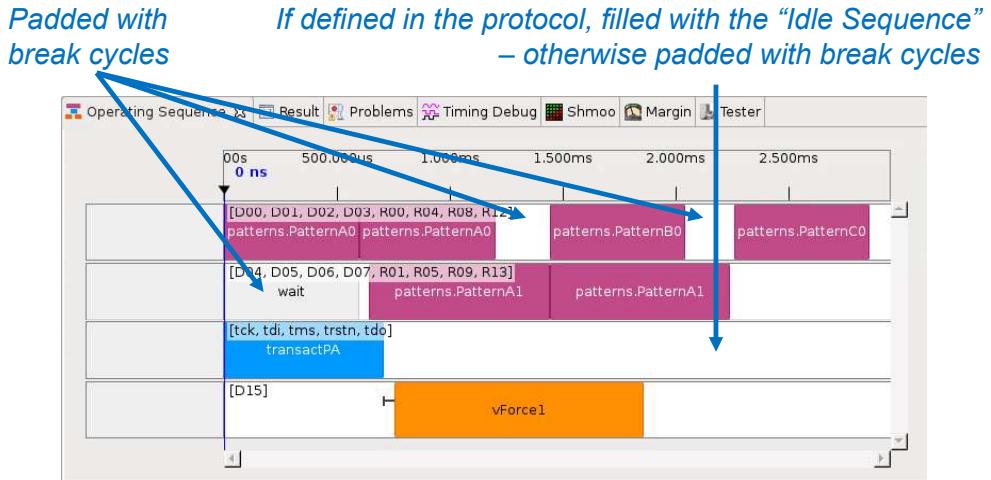
- Call each pattern in a dedicated sequential group;
- Call these sequential groups in a single parallel group.



# Padding: Digital Instruments

Often *lanes* of *operating sequences* are not fully filled.

As the sequencers must keep on running, padding cycles are inserted.

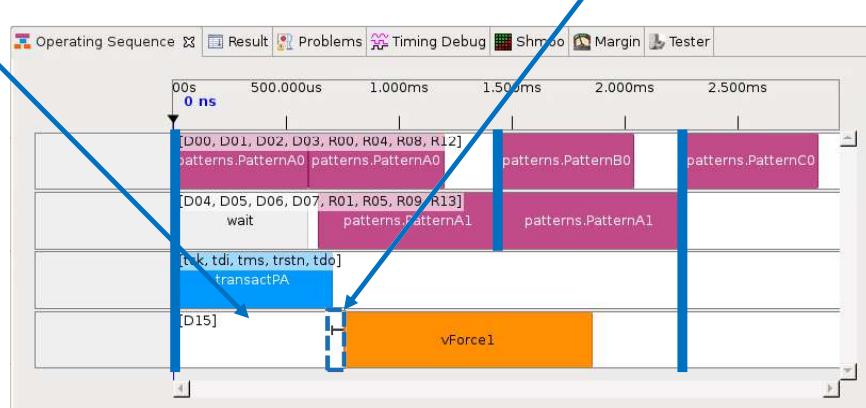


# Padding: Other Instruments and Actions

For other *instruments* padding depends on the *instrument setup*.  
The time can be used for preparing an *action*.

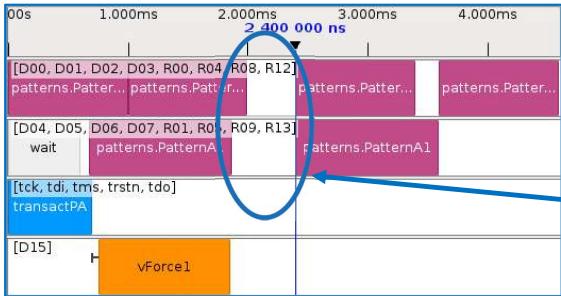
Padded according to instrument setup

Explicitly shown by the operating sequence view:  
The time required to prepare the action



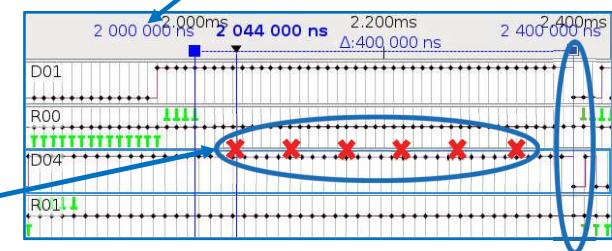
# Synchronization and Padding

The default setting for a *parallel group* is, that all patterns and *transaction sequences* at the beginning of *sequential groups* start within a time interval of 1ns.



Modified example of previous slides, changed clock periods:  
10us and 12us

The start of the pattern group is delayed by 400us due to synchronization:  
Details in the timing debug view



For a synchronized start padding cycles must be inserted until the begin of the cycle periods is aligned.

If break cycles are added, the cycle number for the next pattern start must be a multiple of 8

## Operating Sequence: Sync Accuracy

The *sync accuracy* parameter is used to tighten or relax synchronization for patterns and/or *transaction sequences*.

Relaxing accuracy can reduce the vector padding, i.e. the number of added *break cycles*. As a result, the run time of *operating sequences* is reduced.

The *sync accuracy* parameter defines the maximum difference between start times of patterns and/or *transaction sequences* that should start at the same point in time.

```
1sequence OpSeqSyncAccExample {
2    parallel parGrp1 {
3        syncAccuracy = 20us;
4        sequential seq1_1 {
5            parallel parSubGrp1_1 {
6                sequential seq1_1_1 {
7                    patternCall PatternAO PAO_1;
8                    patternCall PatternAO PAO_2;
9                }
10}
11}
12}
13}
```

# Synchronization: Other Elements

## Actions

The execution of *actions* is started as close as possible to the point of time they should start, but never before that point.

## Delay elements

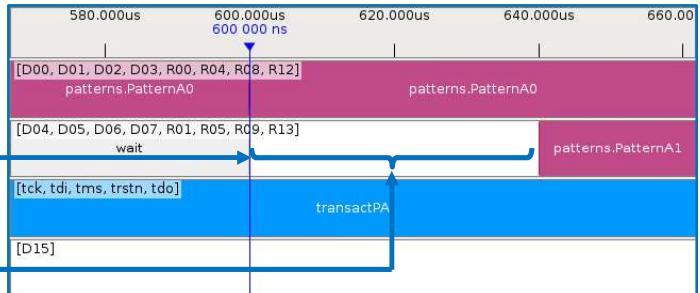
If the exact duration of the specified delay cannot be achieved, it is larger, but as close as possible.

```
sequential seq1_1_2 {
    wait 0.6ms;
    patternCall patterns.PatternA1 PA1_1;
}
```

600us/8us period = cycle 75:  
No pattern start possible

Pattern starts 40us later than specified

Added break cycles



## Summary - What you should have learned

- Anchors in patterns can be used to invoke *actions* which can be of two different categories:
  - “interruptive” and
  - “non-interruptive”.
- Interruptive *actions* are executed at signals that are part of the pattern.  
All other signals of the pattern, that do not participate in the *action*, run *break cycles*.
- A setup of a *parallel group* in the middle of an *operating sequence*, that starts the concurrent execution of multiple patterns or *transaction sequences*, might produce additional delays to wait for the right point in time for this start.
- Setting appropriately the parameter “syncAccuracy” can reduce this delay.  
The parameter specifies, what exactly “start at the time” means.
- The execution of *actions* is started at the point of time, they should start, or as close as possible after this point.
- If a delay element is specified in an *operating sequence*, then *break cycles* are executed.  
More break cycles might be executed than given by the time specified in the *operating sequence*.



# Technical Documentation Center (TDC)

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION

Technical Documentation Center - 1



## Learning Objectives

- Understand how to access the TDC
- Understand the structure of the TDC
- Understand the main components of the TDC
- Understand the structure of the Service documentation
- Learn different ways to search for information in the TDC
- Learn how to send documentation feedback



January 2020



# Accessing the TDC

- As part of the SmarTest software:  
Automatically installed with SmarTest
- TDC on the Advantest web:  
<https://inside.advantest.com/V93000/help/index.jsp>
- Easy TDC installation on your PC/workstation:  
Download TDC package from  
<https://www.advantest.com/service-support/ic-test-systems/technical-documentation/v93000-technical-documentation>

## TDC Structure

The screenshot shows the V93000 - ADVANTEST Technical Documentation Center website. At the top, there is a search bar with the placeholder "Enter search term" and a "Search" button. Below the search bar is a navigation menu with links to version 8.3.2, 8.3.1, 8.3.0, 8.2.5, 8.2.3, 8.1.5, 7.5.3, 7.5.2, 7.4.5, and 7.3.4. A blue arrow points to the "8.2.5" link with the text "Select your SmarTest version". Below the menu, there is a section titled "SmarTest 8.2.5 Documentation" with a publication date of 26-Jul-2019 and an update date of 5-Nov-2019. This section includes links to "Release Information", "Quickstart videos and tutorials", and "Examples". Another blue arrow points to the "Examples" link with the text "Complete example projects". The main content area is divided into three columns: "Concepts and basics" (containing "SmarTest introduction", "SmarTest concepts", "SmarTest workstation administration", "SmarTest Work Center basics", "First steps with SmarTest", "Quick reference cards", and "Glossary of terms"), "Typical Test Engineer tasks" (containing "Developing test programs", "Creating and modifying test programs", "Developing test methods", "Logging test results", "Using SmarTest tools", "Test Cell Access", and "Application papers, presentations and tips"), and "More info section..." (containing "Reference", "Technical specifications", "System reference", "Instrument reference", "SmarTest Setup Format reference", "API and TML reference", and "How to use the TDC"). Arrows point from the labels "Concepts and basics", "Typical Test Engineer tasks", and "More info section..." to their respective sections in the documentation.

# SMT 8 end user documentation

## • Stronger focus on code examples

- Centralized in example section
- Validated with each release
- Code snippets reused in documentation
- Organized in projects covering all test segments

## Further new elements

- SMT8 Glossary of Terms
- Concept Section to explain SMT8 concepts
- Revised DUT Board Design Guide

## • Reference Documentation

- Step 1: Automated import from various sources e.g. API, DS API, instruments, ...  
=> ensures completeness and correctness
- Step 2: Detailed description and code examples are added in post processing step

## Application Layer

- Using instruments
- Specifying setup data
- List of examples
- DC programming guidelines
- RF programming guidelines

# Service documentation

- Product life cycle
  - Site planning and preparation
  - Installation
  - Operation
  - Preventive/corrective
  - Decommissioning (installation)

V93000 - ADVANTEST Technical Documentation Center

Slot Based Infrastructure Service 8.2.5 | More ▾ Search

8.2.5

Slot Based Infrastructure Service 8.2.5 Documentation - 2019-05-20 Preview Published on 20-May-2019  
Release information · Infrastructure specification · Safety information

**Test system**  
Site planning and preparation guide  
Installation guide  
Operation guide  
Preventive maintenance guide  
Repair guide

**Ancillary equipment**  
System controller software installation guide  
Cooling system guide  
Calibration robot guide  
Purge Kit Guide  
Engineering cart and manipulator guides

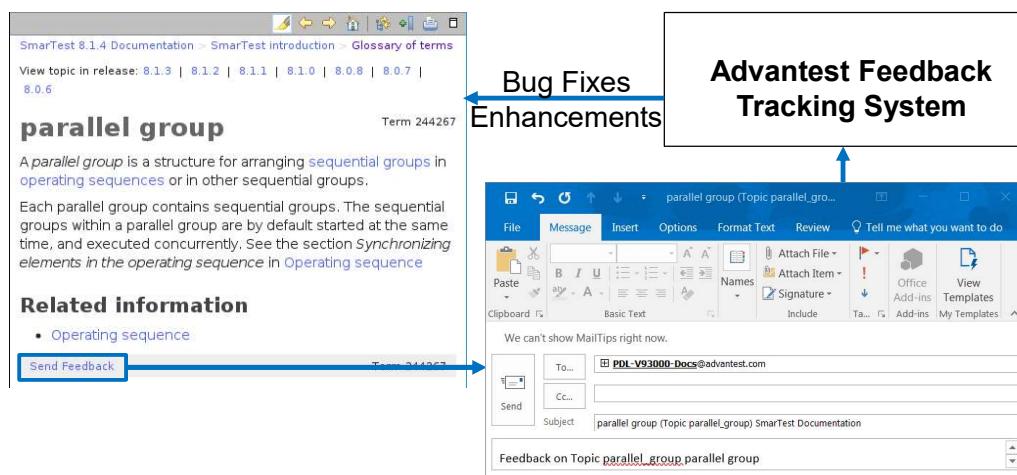
Send Feedback · Send Link

# TDC Search Features

- Auto-completion of search terms
- Search as you type
- Topic preview when hovering over instant search results
- Searches are restricted to the selected SmarTest version
- Search in sections of interest directly
- Advanced search query

**TDC Search helps you find relevant information faster.**

## Send Documentation Feedback



## Summary - What You Should Have Learned

- How to install and access the Technical Documentation Center
- How to use the search function in the TDC
- How to send documentation feedbacks
- How to download and use the application examples



January 2020



## Testflow Files

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest Testflow Files - 1

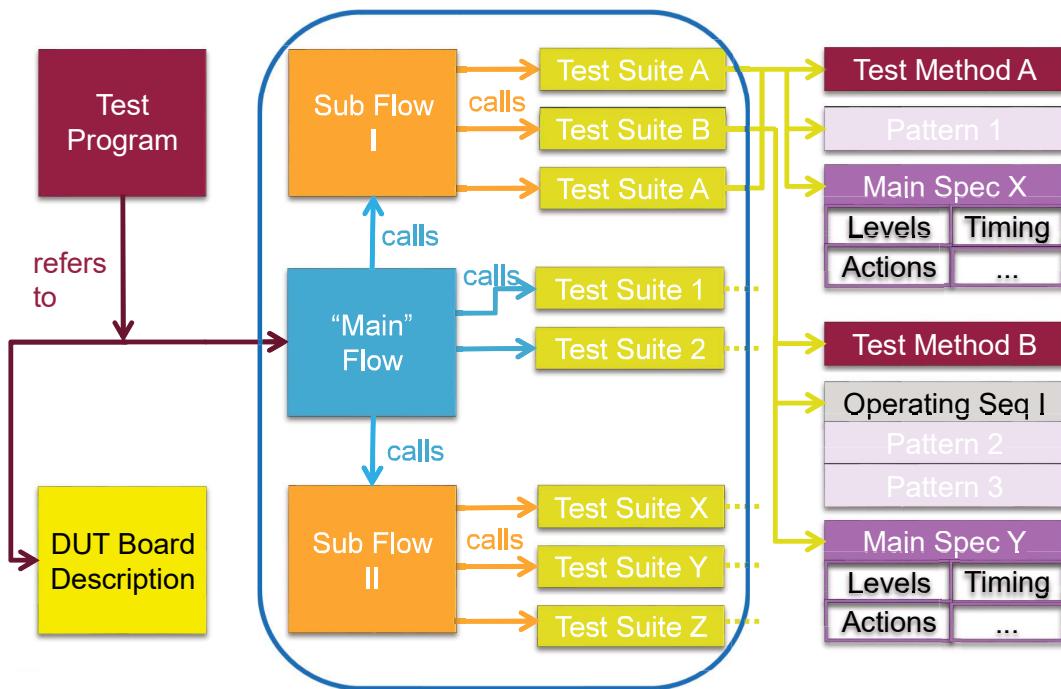
# Learning Objective

- Learn how to control the sequence of test executions in testflows.
- Know how to make use of *auxiliary flows*.

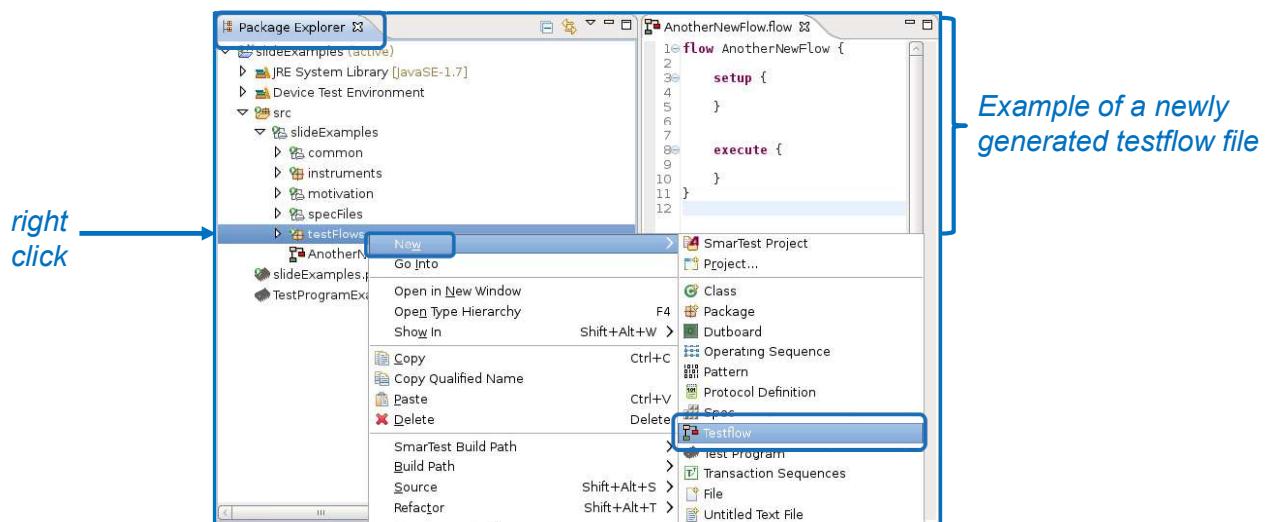
# Agenda

- Controlling the execution sequence in testflows
- Creation of testflow files
- Parameters, variables and binning in testflows
- Auxiliary testflows
- Shmoo and margin tools in testflows

# Building Blocks: Testflows and Test Suites



## Creating a New Testflow File



Editing testflow files:

As for other files in the Smartest Setup Format, content assist is provided.

# Execute Part in Testflow Files – Source Code

```
1 flow Flow_I {
2     in flow_I_Parameter = 0;
3     setup {}
4
5     execute {
6         Test_Suite_A.execute();
7         print("Test suite A ");
8         if (Test_Suite_A.pass) {
9             println("passed.");
10            Test_Suite_B.execute();
11        } else {
12            println("failed.");
13            stop();
14        }
15        for (counter: 0..3) {
16            Sub_Flow_K.flow_K_Parameter = counter;
17            Sub_Flow_K.execute();
18        }
19        waitForAllResults();
20        messageLogLevel = 10;
21        message (10, "Subflow done.");
22    }
23}
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40 }
```

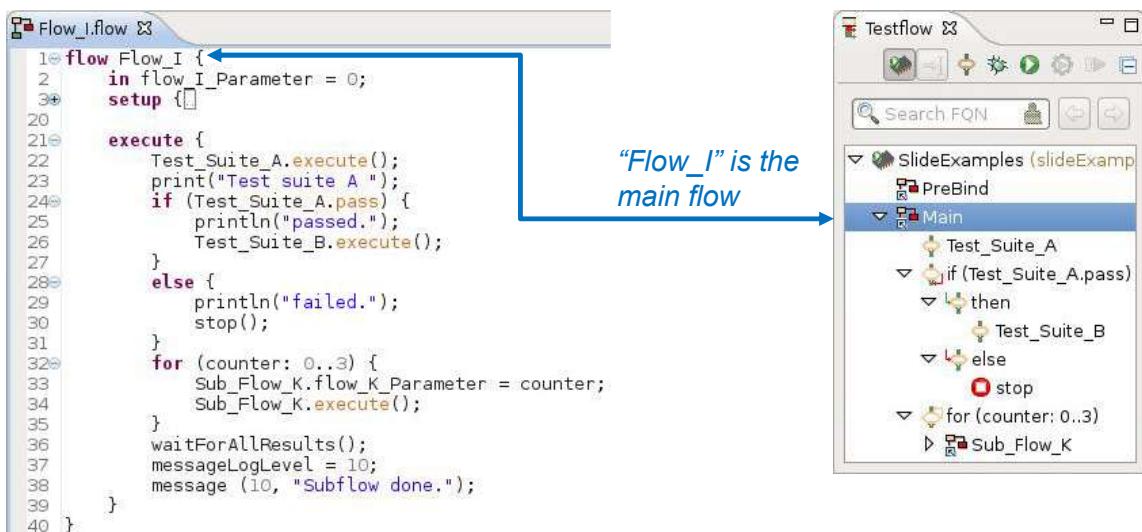
The execution part defines, which of the test suites and the sub flows, that are defined in the setup part, are executed in what sequence.

The sequence can be controlled by

- conditional branching;
- stopping execution;
- waiting for all results of processes running in the background;
- loops.

Parameters of testflows and test suites can be set in the execute part, potentially overwriting the settings of the setup part.  
Settings must be provided before execution.

## Graphical Testflow View: Details



The testflow view shows the sequence of executing testflows and test suites and also commands that control this sequence.

# Parameters of Testflow Files

```

1 flow Flow_K {
2     in flow_K_Parameter = 0;
3     out failingTests = 0;
4
5     setup {
6         flow Sub_Flow_L calls Flow_L {
7             flow_L_Parameter = 2;
8         }
9         suite Test_Suite_C calls Test_Method_A {
10            testSignals = "R01"; // test method parameter
11            measurement.specification = setupRef(Main_Spec);
12            measurement.pattern = setupRef(Pattern_II);
13        }
14    }
15
16    execute {
17        Sub_Flow_L.execute();
18        failingTests = Sub_Flow_L.failingTests;
19        Test_Suite_C.execute();
20        if (!Test_Suite_C.pass) {
21            failingTests = failingTests + 1;
22        }
23    }
24 }

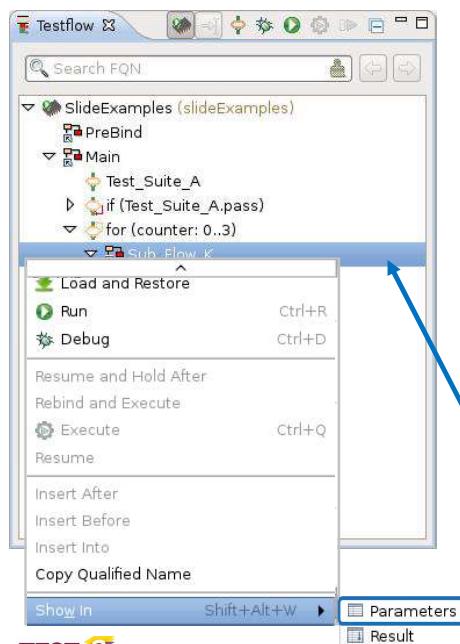
```

Testflow parameters are always defined and used multi-site aware.

The returned parameter value of the called sub flow is stored in the testflow parameter of the calling testflow.

Here “failingTests” is incremented only for the failing sites.

## Editing Parameters in the Parameters View



Switch between tabular and tree view

Parameter	Value	Type
flow_K_Parameter	0	MultiSiteLong
failingTests	0	MultiSiteLong
byPass		Boolean
messageLogLevel		MultiSiteLong

Test Suite	Test Method	Measurements			
name	operating...	pattern	spec		
Test_Suite_A	Test_Method_A	mea...	patter...	Main	
Test_Suite_D	Test_Method_B	mea...	slideExam...	Main	

Right click on a testflow or test suite and select “Show In” and “Parameters”.

# Parameters View - Overview

The screenshot shows a table with columns: Test Suite, Test Method, Parameters, Name1, Value1, Name2, Value2, and Name. The table lists several rows corresponding to different test cases and their parameters.

Open selected testflow file or test method file

Select settings shown in the table:

- System flags
- Measurements
- Parameters
- Parameter Groups

Tabular view

Tree view

The screenshot shows a hierarchical tree structure of parameters. At the top level, there is a measurement node. Under measurement, there are specification, pattern, operatingSequence, myItd, messageLogLevel, bypass, logOnPassOnly, and overridePass nodes. Each node has its value and type listed below it.

Clear parameter and assign the default value

Button shows/hides default parameters

# Variables and Binning

```

1 flow Flow_L {
2     in flow_L_Parameter = 0;
3     out failingTests = 0;
4
5     setup {}
6
7     execute {
8         var debugModeVar = 0;
9         Test_Suite_A.execute();
10        if (!Test_Suite_A.pass) {
11            addBin(4);
12            stop();
13        }
14        Test_Suite_D.execute();
15        if (!Test_Suite_D.pass) {
16            debugModeVar = 10;
17            failingTests = failingTests + 1;
18        }
19        Test_Suite_E.debugMode = debugModeVar;
20        Test_Suite_E.execute();
21    }
22 }
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38 }
```

Testflow variables allow to store intermediate results. The testflow variables are always multi-site aware and can only be used in the execute part of testflow files.

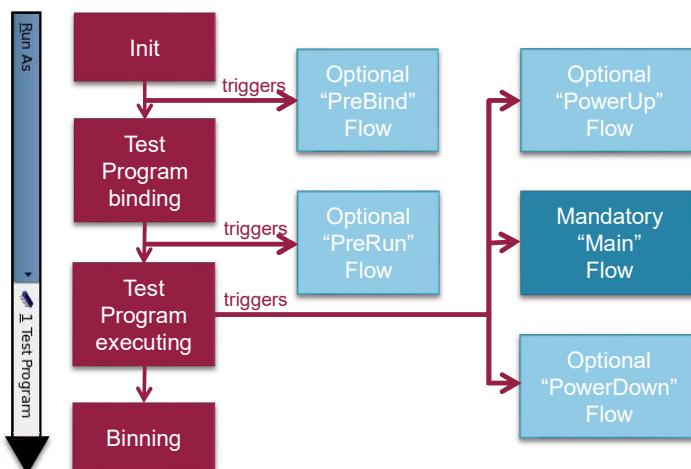
In the execute part of the testflow the command “addBin()” allows to set binning information for devices during test execution.

# Auxiliary Flows

When SmarTest starts testing, at certain stages special activities can be triggered via the optional *auxiliary flows*.

Like the *main flow*, testflows are specified as *auxiliary flows* in the *test program file*.

These are defined in testflow files like any other testflows.



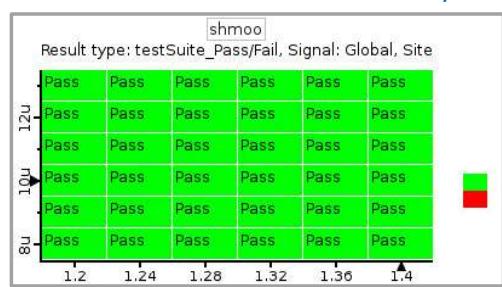
## Shmoo and Margin Tools in Testflows

```
1 flow CharFlow {
2     setup {
3         suite Test_Suite_A calls Test_Method_A {
4             testSignals = "ROI"; // test method parameter
5             measurement.specification = setupRef(Main_Spec);
6             measurement.pattern = setupRef(Pattern_II);
7         }
8         shmoo shmooOverTest_Suite_A {
9             target = Test_Suite_A;
10            axis[Voltage] = {
11                resourceType = specVariable;
12                resourceName = "testFlows.Main_Spec.vcc";
13                range.start = 1.2;
14                range.stop = 1.4;
15                range.steps = 5;
16            };
17            axis[Timing] = {
18                resourceType = specVariable;
19                resourceName = "testFlows.Main_Spec.per";
20                range.start = 8e-06;
21                range.stop = 13e-06;
22                range.steps = 5;
23            };
24        }
25    }
26
27    execute {
28        Test_Suite_A.execute(); // run the test suite
29        shmooOverTest_Suite_A.execute(); // run shmoo
30    }
31 }
```

*Shmoo* test suites and *margin* test suites are setup and executed in testflow files.

*Example:*  
*Shmoo over*  
*a test suite*  
*with*  
*setup and*  
*execution.*

*Shmoo plot*



# Summary - What you should have learned

- In the execute part of a testflow you can implement
  - conditional branches and loops;
  - waits until all background processes have been completed;
  - assignments of bins;
  - stops of the whole test program execution;
  - define and use variables to store intermediate results;
  - modify input parameters of subflows and test suites and access their output parameters.
- A testflow can also define and execute shmoo and margin tests.
- The *parameters* view allows to view and edit parameters and system flags of test suites and testflows in tables and tree views.
- Testflows specified as *auxiliary flows* are also defined in regular testflow files and they are called at certain stages of the test program execution.



# Test Methods: Introduction

SmarTest 8.2.5 Training

January 2020

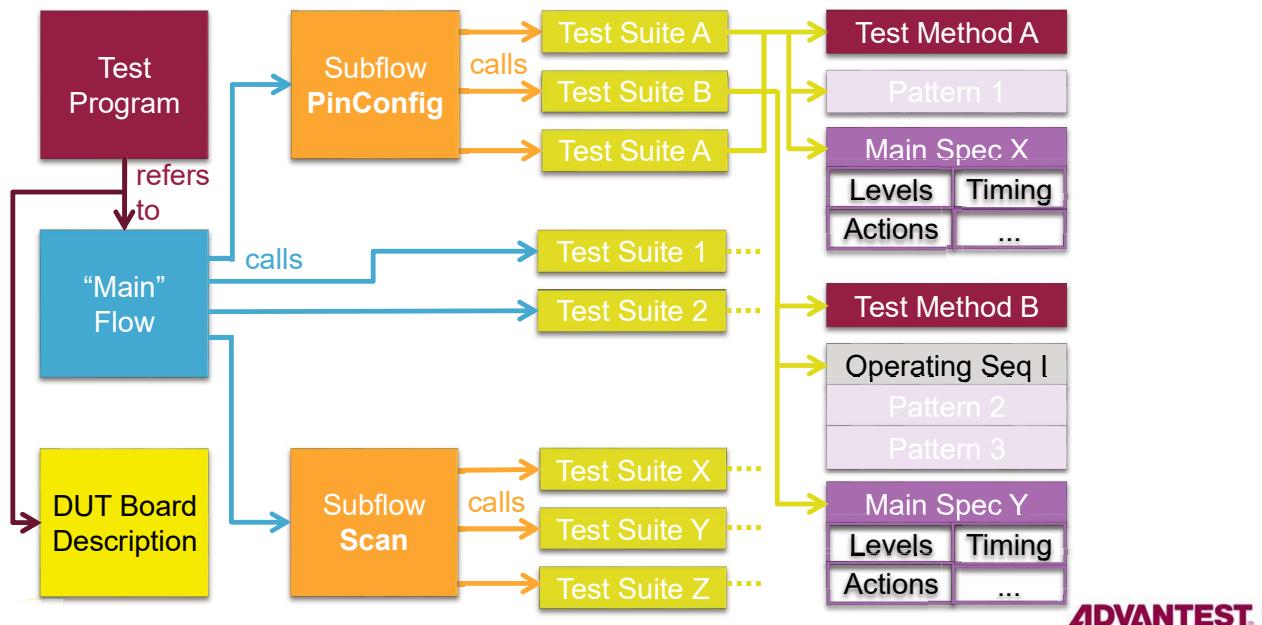
# Learning Objective

- Understand the purpose and basic structure of test methods.

# Agenda

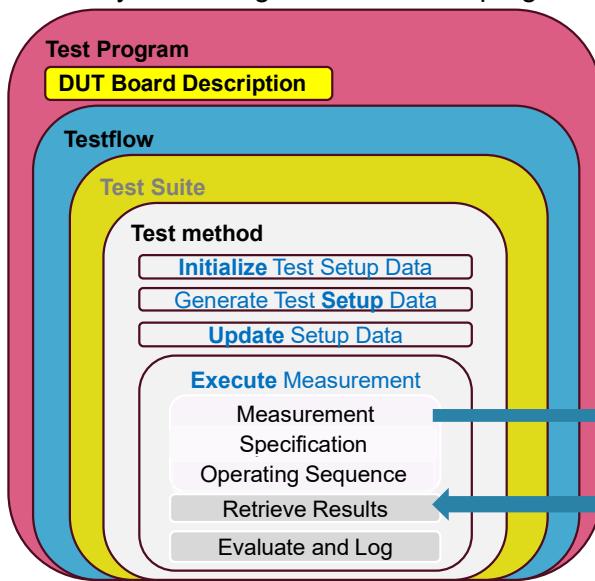
- Test methods: hierarchy in a test program
- Creation of a test method
- Structure of test methods
- Accessing tester hardware: First steps

# Test Program in SmarTest: Building Blocks



## Test Methods: Hierarchy in a Test Program

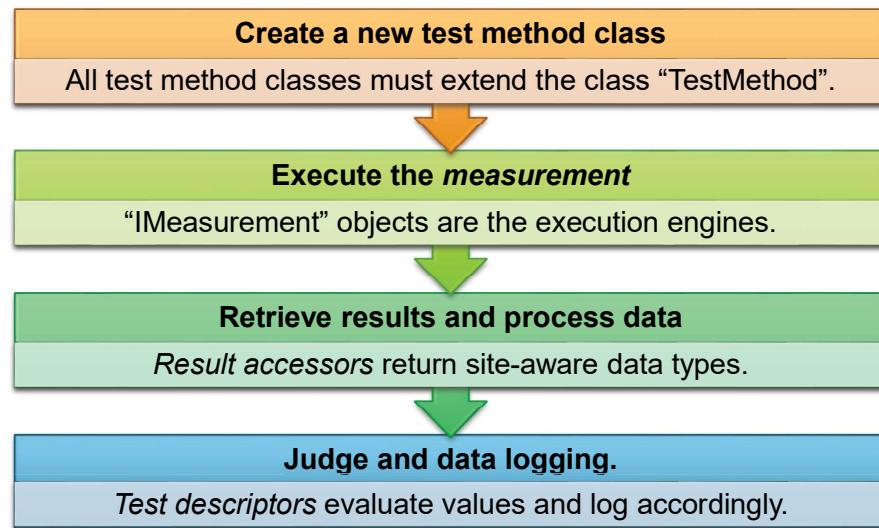
Hierarchy of building blocks of a test program



Test head channels of the V93000 tester

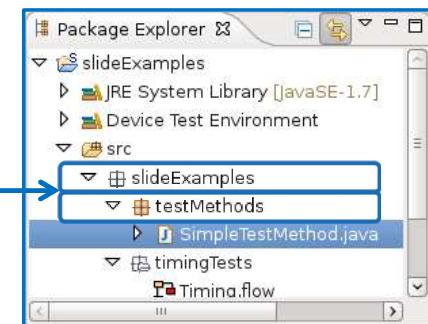


# Building a Test Method



## Structure of Test Methods

```
package slideExamples.testMethods;  
  
import xoc.dta.TestMethod;  
  
public class SimpleTestMethod extends TestMethod {  
    @Override  
    public void initialize() { // Optional }  
  
    @Override  
    public void setup() { // Optional }  
  
    @Override  
    public void update() { // Optional }  
  
    @Override  
    public void execute() { // Mandatory }  
}
```



# Minimized Test Method

```
package slideExamples.testMethods;
import xoc.dta.TestMethod;

public class SimpleTestMethod extends TestMethod {

    @Override
    public void execute() {
        // Mandatory
    }
}
```

The shown minimized test method is executable:  
It can be called and executed from a testflow; however, it does nothing.

Now one more specific element will be added that is used to setup and execute a test on the tester.

The specific *measurement* element is of type “IMeasurement”.

## First Test Method Accessing Tester Hardware

```
package slideExamples.testMethods;
import xoc.dta.TestMethod;

public class SimpleTestMethod extends TestMethod {
    public IMeasurement measurement;
    @Override
    public void execute() {
        measurement.execute();
    }
} Test method

setup {
    suite SimpleSuite calls SimpleTestMethod {
        measurement.specification = setupRef(simpleSpec);
        measurement.operatingSequence = setupRef(simpleOpSeq);
    }
}
execute { SimpleSuite.execute(); } Testflow
```

# Summary - What you should have learned

- A test suite must call a test method.
- A test method is a Java class that must inherit the base class “TestMethod”
- The class method “execute()” must be implemented, the class methods “initialize()”, “setup()” and “update()” are optional.
- A test method uses
  - “IMeasurement” objects to setup test;
  - “IMeasurement” to execute physically tests;
  - *result accessors* to retrieve results from the “IMeasurement” object and
  - *test descriptors* to evaluate results and perform data logging.The last three items are only performed in the class method “execute()”.
- For setting up a test suite, a *specification file* and an *operating sequence file* (or pattern file) are assigned to an “IMeasurement” object of the test method called by the test suite.  
The corresponding *measurement* is physically executed according to the settings read from these files.

## Backup

Details of the code of a simple test method.

## Simple Example of a Complete Test Method

```
package slideExamples.testMethods;

import xoc.dta.TestMethod;

public class SimpleTestMethod extends TestMethod {

    @Override
    public void initialize() { // Optional }

    @Override
    public void setup() { // Optional }

    @Override
    public void update() { // Optional }

    @Override
    public void execute() { // Mandatory }
}
```

## Simple Example of a Complete Test Method

```
package slideExamples.testMethods;

import xoc.dta.TestMethod;

public class SimpleTestMethod extends TestMethod {

    @Override
    public void initialize() { // Optional }

    @Override
    public void setup() { // Optional }

    @Override
    public void update() { // Optional }

    @Override
    public void execute() { // Mandatory }
}
```

Java groups classes in packages.

The package name, a fully qualified name, refers to the folder of the test method file.

The path is relative to the source folder ("src") of the SmarTest project.

Conventionally, in Java the name of the class starts with a capital letter.

# Simple Example of a Complete Test Method

```
package slideExamples.testMethods;

import xoc.dta.TestMethod;

public class SimpleTestMethod extends TestMethod {

    @Override
    public void initialize() { // Optional }

    @Override
    public void setup() { // Optional }

    @Override
    public void update() { // Optional }

    @Override
    public void execute() { // Mandatory }
}
```

The “import” statement refers to the package that contains the class “TestMethod”.

The class “TestMethod” must be inherited by any test method class.

The package is an essential part of Smartest, that has been developed and is maintained by Advantest.

# Simple Example of a Complete Test Method

```
package slideExamples.testMethods;

import xoc.dta.TestMethod;

public class SimpleTestMethod extends TestMethod {

    @Override
    public void initialize() { // Optional }

    @Override
    public void setup() { // Optional }

    @Override
    public void update() { // Optional }

    @Override
    public void execute() { // Mandatory }
}
```

In order to have access to the tester, any test method classes must extend the class “TestMethod”.

## The class

- gives access to setups and results of the tester hardware and to datalog results;
- provides the “context” interface, which is about the environment in that the test method is embedded.

# Simple Example of a Complete Test Method

```
package slideExamples.testMethods;

import xoc.dta.TestMethod;

public class SimpleTestMethod extends TestMethod {

    @Override
    public void initialize() { // Optional }

    @Override
    public void setup() { // Optional }

    @Override
    public void update() { // Optional }

    @Override
    public void execute() { // Mandatory }
}
```

*As a test method class extends the class “TestMethod”, it inherits the four methods “initialize()”, “setup()”, “update()” and “execute()”.*

*These methods are called at run time at different stages of the execution engine and define the structure of any test method class.*

*“execute” must be implemented, “initialize()”, “setup()”, “update()” can be omitted.*



# Basics of Test Methods

SmarTest 8.2.5 Training

January 2020

# Learning Objective

- Understand the basic structure of test methods.
- Know how to implement simple test methods that
  - Run a test on the tester.
  - Modify a test setup.

# Agenda

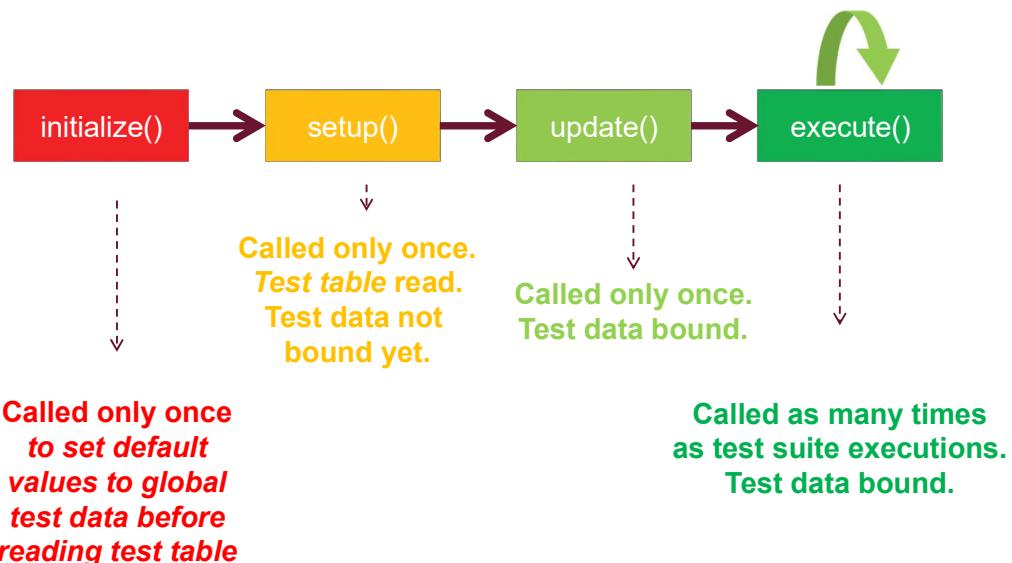
- Structure of test methods
- Modifying setups with the *Instrument Specific API*

# Structure of Test Methods

Each test method has four major methods: "initialize()", "setup()", "update()" and "execute()" which are inherited from the class "TestMethod".

```
public class SimpleTestMethod extends TestMethod {  
  
    @Override  
    public void initialize() { // Optional }  
  
    @Override  
    public void setup() { // Optional }  
  
    @Override  
    public void update() { // Optional }  
  
    @Override  
    public void execute() { // Mandatory }  
}
```

## Execution Stages of Test Methods



## “initialize()” and “setup()”

### “initialize()”:

- Optional - empty implementation is provided by the base class “TestMethod”.
- Rarely used, typically for setting default values for limits, fail bins, etc.
- Called first before the “PreBind” flow and only once per test program execution – in subsequent iterations, it will not be called again.

### “setup()”:

- Optional - empty implementation is provided by the base class “TestMethod”.
- *Measurement* objects are empty – do not yet contain any setup data.
- Typically used to programmatically generate setup files such as *specification files* and *operating sequences files*.
- Called after executing the “PreBind” flow (reading the *test table*).
- Called only once per test program execution – in subsequent iterations, it will not be called again.

## “update()” and “execute()”

### “update()”:

- Optional - empty implementation provided by the base class “TestMethod”.
- *Measurement* objects already contain setup data.
- Typically used to access and modify the setup data of *measurements*, for example to set site-specific setup data.
- Cannot be used to execute a *measurement*.
- Called only once.

### “execute()”:

- Mandatory - must be implemented but can be left empty.
- *Measurements* objects already contain setup data and can be executed.
- Called after “update()” and every time a testflow and its corresponding test suites are executed.

# Execution of Measurements

```
public class SimpleTestMethod extends TestMethod {  
  
    public IMeasurement measurement;  
  
    @Override  
    public void execute() {  
        measurement.execute();  
    }  
}
```

*The measurement object is the central access point to the tester hardware:*

- Setup a test.
- Execute a test.
- Retrieve test results.

## “measurement.execute()”:

- Implicitly binds if not yet bound or something has changed:  
Acquires the resources, downloads *measurement* setup data to the tester hardware and then starts the execution.
- Can only be called in the “execute()” method of the test method.
- An exception is thrown if a problem occurs.

# Accessing Measurement Setup Data

In test methods, the setup (and also the result) data of a *measurement* is always accessed using the corresponding “IMeasurement” object.

Here is an example of how to access the data of an *rfMeas* instrument:

```
measurement.rfMeas("rfOut1").modPower("cap1").setSamples(8192);
```

*Measurement* setup data can be accessed in the “update()” and “execute()” part of a test method.

Modifications of *measurement* setup data (for example, specification variables) affect only the currently used *measurement* object.

Thus the modifications are **local**.

*Measurements* of other test suites will not see the modification, even if these test suites call the same test method.

Exception:

Modifications of patterns are **global**, a test suite does not maintain a dedicated, local copy of a pattern.

# Measurement Setup in Specification Files

*Specification file based on the SmarTest Setup Format (SSF)*

```
setup rfMeas rfOut1 {
    config.mode = highResolution;

    action modPower cap1 {
        frequency = 2e9 Hz;
        sampleRate = 250e6 Hz;
        samples = 4096;
        expectMaxPower = 5 dBm;
    }
}
```



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest Test Methods: Basics - 10

## Dynamic Modification of Instrument Setups

*Specification file based on the SmarTest Setup Format (SSF)*

```
setup rfMeas rfOut1 {
    config.mode = highResolution;

    action modPower cap1 {
        frequency = 2e9 Hz;
        sampleRate = 250e6 Hz;
        samples = 4096;
        expectMaxPower = 5 dBm;
    }
}
```

*Test method based on the Instrument Specific API*

```
public class SimpleTestMethod
    extends TestMethod {
    public IMeasurement measurement;

    @Override
    public void update() {
        measurement.rfMeas("rfOut1")
            .modPower("cap1")
            .setSamples(8192);
    }

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



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest Test Methods: Basics - 11

# Instrument Specific API

The *Instrument Specific API* is the interface in test methods to access *instrument* setups.

- Part of the *Device Test API*.
- It can be accessed from an object of type “IMeasurement”:  
It accesses the setup data of the *measurement*.
- For each *instrument* a method with the *instrument*’s name exists.  
*A signal* name or a *signal group* name as the argument is mandatory.
- Uses the same keywords as used in the *specification files*.  
→ Consistent and intuitive.

The previous example was based on the *rfMeas instrument* - the same scheme applies to other instruments like

*dcVI, awg, digitizer, digInOut, rfStim, ....*

## Instrument Specific API – Details

To develop an API that allows to access in a test method *instruments* settings of *measurement* setups, all the possible properties, sets, and actions have been translated to functions.  
All these functions are now part of the *Instrument Specific API*.

Example:

Assume, the measurement object of the test method is named “measurement”:

```
ILevel vddLevel = measurement.dcVI("VDD").level();
vddLevel.setVforce(1.3);
```

This reflects the property group “level” of the *dcVI instrument*, as it is specified in a *specification file*:

```
setup dcVI VDD {
    level.vforce = ...;
    level.irange = ...;
    ...
}
```

# Instrument Specific API – “Setter” and “Getter” Methods

The API provides methods to get and set values of properties.

Previous example continued:

Set the force voltage of the level property group in a test method:

```
measurement.dcVI("VDD").level().setVforce(3.0);
```

Get the force voltage of the level property group in a test method:

```
voltages = measurement.dcVI("VDD").level().getVforce();
```

This function returns a *multisite data type*.

Syntax of the corresponding setup in the *specification file*:

```
setup dcVI VDD {  
    level.vforce = 3.0 V;  
    ...  
}
```

## Summary - What you should have learned

- The execution order of the different class methods:
  1. “initialize()”: Executed once, no test table read so far, purpose: set default values, optional.
  2. “setup()”: Executed once, test table applied, purpose: generate setup files before bind, optional.
  3. “update()”: Executed once, test setup bound, purpose: set site specific values, optional.
  4. “execute()”: Executed multiple times if needed, purpose: execute test + retrieve results + pass/fail decision + datalog, mandatory.
- “measurement.execute()” executes physically a test with the test head channels according to the setup defined in the measurement object “measurement”.
- The *Instrument Specific API* is the interface in test methods to access *instrument* setups.
- The names and structure of the functions provided by the *Instrument Specific API* match the keywords of the corresponding setups in the *specification files*.
- The “setter” and “getter” methods allow to read and modify values of the *instrument* settings.



# Test Methods: MultiSite Data Types & Context

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

SmarTest Test Methods: MultiSite Data Types & Context - 1

## Learning Objective

- Know how to implement simple test methods that
  - handle data per site correctly for a single site setup and for various multi site setups.
  - access properties of the environment in that a test is running.



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

SmarTest Test Methods: MultiSite Data Types & Context - 2

# Agenda

- Usage of *multisite data types*
- Interacting with the context in that a test and its test method is executed
- Error handling

## Instrument Specific API – “Setter” and “Getter” Methods

The API provides methods to get and set values of properties.

Example:

Set the force voltage of the level property group in a test method:

```
measurement.dcVI("VDD").level().setVforce(3.0);
```

Get the force voltage of the level property group in a test method:

```
voltages = measurement.dcVI("VDD").level().getVforce();
```

The function “getVforce()” returns a *multisite data type*.

Syntax of the corresponding setup in the *specification file*:

```
setup dcVI VDD {  
    level.vforce = 3.0 V;  
    ...  
}
```

# Type of Result Returned by “Getter” Methods

```
MultiSiteDouble voltages = measurement.dcVI("VDD").level().getVforce();
```

“MultiSiteDouble” is a *multisite data type* provided by SmarTest, to work in an easy and flexible way with one or more sites:

- *Multisite data types* contain data for all configured sites in the *DUT board description file*.
- *Multisite data types* are aware of which sites are active.  
For a site, which is configured, but not active, the value is “**null**” for test results.
- Using *multisite data types*, a test method works with any number of configured sites.
- No need to change the source code of test methods if the list of
  - configured,
  - ignored,
  - enabled or
  - disabledsites changes.

## MultiSite Data Types: Names and Functions

The name of *multisite data types* is composed as follows:

“MultiSite” + data type.

Examples:

- “MultiSiteDouble” suggests that the value for a site is a “Double”.
- “MultiSiteWaveComplex” suggests that the value for a site is an object of type “Wave<Complex>”.

*Multisite data types* provide the same variants of the functions

- “get(..)”,
- “getData()” and
- “set(..)” functions

to give access to the content of the *multisite data types*

# “Getter” Methods of MultiSite Data Types

Example:

```
MultiSiteBoolean msBool = new MultiSiteBoolean();
// get the Boolean value of site 3:
Boolean v = msBool.get(3);

// get all data as an array;:
Boolean[] boolArray = msBool.getData();
```

Get the value of a specific site: “Type get(int siteNumber)”

- Example: “msBool.get(3)” returns a Boolean for site 3.  
For a site, that is not active, “null” is returned.

Get the values for all the sites as an array: “Type[] getData()”

- Example: “msBool.getData()” returns an array of type “Boolean[]”.

# “Setter” Methods of MultiSite Data Types

Example continued, showing all variants of the “Boolean.set(..)” function:

```
// set all sites to "true":
msBool.set(true);

// set only site 1 to "false":
msBool.set(1, false);

MultiSiteBoolean msBool2 = new MultiSiteBoolean(true);

// copy the values of active sites from "msBool":
msBool2.set(msBool);

// copy values of an array of type "Boolean[]":
msBool2.set(boolArray);
```

# Arithmetic Operations for MultiSiteDouble

Java does not allow to overload arithmetic operators like “+”, “-”, “/”, “\*”...

Instead, the type “MultiSiteDouble” supports (for example, consider variables “a”, “b”, “c”):

```
a.add(b); b.subtract(c); a.divide(c); a.multiply(b);
```

These arithmetic methods can be “concatenated”.

They will always be evaluated from left to right:

```
a.subtract(b).divide(c); // this is (a-b)/c  
a.add(b.multiply(c)); // this is a+b*c
```

Complex operations require iterating over the active sites using 3 steps:

```
// Calculating b=1-cos(a):  
for (int site : context.getActiveSites()) {  
    b.set(site, (1 - Math.cos(a.get(site))));  
}
```

1. Get single site value
2. Calculate
3. Store result

# Getting Information from the Context

All test methods inherit from the class “TestMethod”.

The corresponding package includes a protected variable “context” which is about interacting with the environment in which the test method is embedded.

Examples of use cases:

- Get the number of configured sites:

```
context.getNumberOfConfiguredSites();
```

- Get the list of active sites:

```
context.getActiveSites();
```

- Get the fully qualified name of the test suite calling the test method:

```
context.getTestSuiteName();
```

- Read the identification string of the DUT board:

```
context.dutBoard().readIdString();
```

# Broadcasting Specification Changes

The variable “context” also provides a *specification publisher* interface.

```
ISpecPublisher global = context.spec("example.SomeSpecName");
```

This *specification publisher* of type “ISpecPublisher” provides functions to set specification variables, *instrument* configurations and *action* properties.

```
global.setVariable("someVariableName", 3.3);
```

The new value “3.3” set by the *specification publisher* will be broadcasted to all *measurement* objects which refer to the specification “SomeSpecName”.

“context” provides access to various aspects of the environment:

- Alarm configuration
- Binning and *bin table*
- Datalog format
- DutBoard ID
- Patterns
- Tester: 10 MHz in/out
- Tester: Utility power
- Test program setup
- ...

For more examples and information with respect to “context”, refer to the [TDC](#), topic [“xoc.dta.ITestContext”](#).

## Error Handling in Test Methods

There are many scenarios, for that a test method should stop execution and dump an error message. For example, if a test method receives invalid values for the input parameters.

Then the test method should throw an exception to stop the current testflow execution:

```
if (somethingWentWrong) {  
    throw new UncheckedDTAEException("Hey, this is what went  
        wrong and why...");  
}
```

Example with test method name and line number in the error message:

```
if (somethingWentWrong) {  
    throw new UncheckedDTAEException("[  
        + this.getClass().getName() + ": line "  
        + new Exception().getStackTrace()[0].getLineNumber()  
        + "] Hey, this is what went wrong and why...");  
}
```

# Summary - What you should have learned

- The purpose of *multisite data types*:  
To work in an easy and flexible way with any number of configured sites.
- The usage of *multisite data types* with respect to
  - naming;
  - instantiation;
  - setting values;
  - getting values and
  - basic arithmetic operations.
- The implementation of a loop over all active sites.
- The implementation of complex calculations on *multisite data types*.
- The purpose and usage of the “context” variable:  
Interacting with the environment in which the test method is embedded.



# Result Access in Test Methods

SmarTest 8.2.5 Training

January 2020

# Learning Objective

- Know how to implement simple test methods that
  - retrieve test results,
  - make a pass/fail decision and
  - push results to the data log.
- Learn more about how to use SmarTest's software interfaces for these tasks:
  - *result accessors* and
  - *test descriptors*.

# Agenda

- Result accessors
- Test descriptors

# Result Accessors – Typical Scenario

Example of a typical use case:

After the execution of a *measurement*, that calls *actions* to measure voltages, the measured values are uploaded.

```
25  @Override
26  public void execute() {
27      // Execute the Measurement
28      measurement.execute();
29
30      // Reserve the result
31      IDcVIResults measuredResultsHandle =
32          measurement.dcVI("D01+D02").preserveResults();
33
34      // Release tester resources -> next test suite can start
35      releaseTester();
36
37      // Retrieve results for the voltage measure action
38      Map<String, MultiSiteDoubleArray> measuredVoltages =
39          measuredResultsHandle.vmeas("").getVoltage();
```

## Result Accessors

- *Result accessors* have to be requested before the next execution of any *measurement*, because this might overwrite the stored results.
- Result preservation automatically ends, if the “execute()” class method of its test method has completely been processed.
- Result preservation is done per *instrument* type.  
A single *instrument* might support different types of results.  
Results can be preserved for several signals/groups at the same time.
- Typically, accessing results is based on *actions*.  
*Actions* have multi-site results based on specialized types, for example:  
double, array of doubles, waveform, complex waveform,...
- Some *result accessors* are not based on action, for example:
  - Pass / fail results from digital patterns or from protocol transactions
  - Captured values from digital patterns or from protocol transactions
- The “getter” methods to retrieve the results are used in a similar way for all the various *result accessor* types.

# Result Accessors for Actions

For *result accessors*, the name of the method, that accesses the results of an *action*, is its type.

As different *actions* of the same type might be set up and executed, optionally the name of the *action* can be given as a parameter of the method instead of an empty string.

```
30 // Reserve the result for multiple signals "D01" and "D02"
31 IDcVIResults measuredResultsHandle =
32     measurement.dcVI("D01+D02").preserveResults();
33
34 // Release tester resources -> next test suite can start
35 releaseTester();
36
37 // Multiple signals: Retrieve results of measured voltages
38 Map<String, MultiSiteDoubleArray> measuredVoltages =
39     measuredResultsHandle.vmeas("voltMeas1").getVoltage();
40
41 // Retrieve results of measured voltages only for signal "D01"
42 MultiSiteDoubleArray measuredD01Voltages =
43     measuredResultsHandle.vmeas("voltMeas1").getVoltage("D01");
```

# Example Usage of Test Descriptors

*Test descriptors* are used in SmarTest to make a pass/fail decision and to control datalogging.

```
1 public class NewLabTest extends TestMethod {
2     public IMeasurement measurement;
3     public IParametricTestDescriptor testDescriptor;
4
5     @Override
6     public void execute() {
7         measurement.execute();
8         IDcVIResults measuredResultsHandle =
9             measurement.dcVI("D01+D02").preserveResults();
10        releaseTester();
11        Map<String, MultiSiteDoubleArray> measuredVoltages =
12            measuredResultsHandle.vmeas("").getVoltage();
13        // pass/fail judgement + data logging
14        testDescriptor.evaluate(measuredVoltages, 0);
15    }
16 }
```

# Test Descriptors: Overview

The “evaluate()” function of a *test descriptor* is used to:

- Determine the pass/fail state.
- Publish details of the test (results, limits,...) to the data log.
- Assign a software bin to sites that failed the test.

The input to the “evaluate()” function are results to be tested.

*Test descriptors* objects are similar to *measurement* objects:

Both need to be declared as member variables of the test method class.

Dedicated *test descriptors* are available for these three types of tests:

- Parametric test
- Functional test
- Scan Test

## Usage Examples of “evaluate()”

Examples of “evaluate()” methods that are called with different types of parameters.

```
42 IDcVIResults measuredResultsHandle =
43         measurement.dcVI("D01+D02").preserveResults();
44
45 MultiSiteDoubleArray measuredD01VoltMeas1 =
46     measuredResultsHandle.vmeas("voltMeas1").getVoltage("D01");
47 // Evaluate results of first "voltMeas1" meas. at signal "D01"
48 testDescriptor.evaluate(measuredD01VoltMeas1, 0);
49
50 MultiSiteDouble secondMeasuredD01VoltMeas1 =
51     measuredD01VoltMeas1.getElement(1);
52 // Evaluate results of 2nd "voltMeas1" meas. at signal "D01"
53 testDescriptor.evaluate(secondMeasuredD01VoltMeas1);
54
55 Map<String, MultiSiteDoubleArray> measuredVoltages =
56     measuredResultsHandle.vmeas("").getVoltage();
57 // Evaluate results of third voltage meas. at "D01" and "D02"
58 testDescriptor.evaluate(measuredVoltages, 2);
```

# Measurement Results and Alarms

```
27 // Execute the Measurement
28 measurement.execute();
29
30 // Reserve the global measurement results and alarms
31 IMeasurementResults measurementResults =
32     measurement.preserveResults();
33
34 // Release tester resources -> next test suite can start
35 releaseTester();
36
37 // pass/fail judgement + data logging
38 functionalTestDescriptor.evaluate(measurementResults);
```

“measurementResults” provides for each active site access to a single pass/fail result of the last execution of the *measurement* object – without any details.

Additionally, it provides data about alarms that might have occurred.

A *functional test descriptor* is used to publish the single pass/fail results and alarms to the data log.

## Summary - What you should have learned

- The purpose of *result accessors*:  
Preserve results and provide a handle to upload the results.
- “releaseTester()” allows to upload test results, postprocess results, make pass/fail decisions and log data in the background.
- These five basics steps of the class method “execute()” should be executed in the following order:
  1. First execute the *measurement*,
  2. then preserve results,
  3. then release tester resources,
  4. then upload results and
  5. finally evaluate pass/fail results and publish results to the datalog.
- The purpose of *test descriptors* is:
  - Store an identification number, name, description, limits, fail bin and more for a pass/fail test.
  - Determine the pass/fail state.
  - Publish details of the test (results, limits,...) to the data log.
  - Assign a software bin to sites that failed the test.

# Backup

More details on the previously shown examples

- for *result accessors of actions*;
- for the usage of “evaluate()”.

## Result Accessors for Actions

For *result accessors*, the name of the method, that accesses the results of an *action*, is its type.

As different *actions* of the same type might be set up and executed, optionally the name of the *action* can be given as a parameter of the method instead of an empty string.

```
30 // Reserve the result for multiple signals "D01" and "D02"
31 IDCVIResults measuredResultsHandle =
32     measurement.dcVI("D01+D02").preserveResults();
33
34 // Release tester resources -> next test suite can start
35 releaseTester();
36
37 // Multiple signals: Retrieve results of measured voltages
38 Map<String, MultiSiteDoubleArray> measuredVoltages =
39     measuredResultsHandle.vmeas("voltMeas1").getVoltage();
40
41 // Retrieve results of measured voltages only for signal "D01"
42 MultiSiteDoubleArray measuredD01Voltages =
43     measuredResultsHandle.vmeas("voltMeas1").getVoltage("D01");
```

# Result Accessors for Actions

Lines 37 – 39:

The retrieved voltage values for multiple signals are stored in a map of type “Map<String, MultiSiteDoubleArray>”

- The keys of the map are the names of the signals (of type “String”).
  - The values of the map are of type “MultiSiteDoubleArray”:
- Per site and per action call of the action “voltMeas1” of type “vmeas” a value is stored, which is of type “Double”.  
The stored value is the measured voltage value.
- ```
30 // Multiple signals: Retrieve results of measured voltages
31 Map<String, MultiSiteDoubleArray> measuredVoltages =
32     measuredResultsHandle.vmeas("voltMeas1").getVoltage();
33
34 // Retrieve results of measured voltages only for signal "D01"
35 MultiSiteDoubleArray measuredD01Voltages =
36     measuredResultsHandle.vmeas("voltMeas1").getVoltage("D01");
37
38 // Multiple signals: Retrieve results of measured voltages
39 Map<String, MultiSiteDoubleArray> measuredVoltages =
40     measuredResultsHandle.vmeas("voltMeas1").getVoltage();
41
42 // Retrieve results of measured voltages only for signal "D01"
43 MultiSiteDoubleArray measuredD01Voltages =
44     measuredResultsHandle.vmeas("voltMeas1").getVoltage("D01");
```

# Result Accessors for Actions

Lines 41 – 43:

Note, that an *operating sequence* or a pattern can call a certain action with a certain name multiple times.

Therefore, the results of a certain action for a single signal (here for example “D01”) are always stored in a data type which is an array of a *multisite data type*.

- ```
32         measurement.dcVI("D01+D02").preserveResults();
33
34 // Release tester resources -> next test suite can start
35 releaseTester();
36
37 // Multiple signals: Retrieve results of measured voltages
38 Map<String, MultiSiteDoubleArray> measuredVoltages =
39     measuredResultsHandle.vmeas("voltMeas1").getVoltage();
40
41 // Retrieve results of measured voltages only for signal "D01"
42 MultiSiteDoubleArray measuredD01Voltages =
43     measuredResultsHandle.vmeas("voltMeas1").getVoltage("D01");
```

# Usage Examples of “evaluate()”

Examples of “evaluate()” methods that are called with different types of parameters.

```
42 IDcVIResults measuredResultsHandle =
43     measurement.dcVI("D01+D02").preserveResults();
44
45 MultiSiteDoubleArray measuredD01VoltMeas1 =
46     measuredResultsHandle.vmeas("voltMeas1").getVoltage("D01");
47 // Evaluate results of first "voltMeas1" meas. at signal "D01"
48 testDescriptor.evaluate(measuredD01VoltMeas1, 0);
49
50 MultiSiteDouble secondMeasuredD01VoltMeas1 =
51     measuredD01VoltMeas1.getElement(1);
52 // Evaluate results of 2nd "voltMeas1" meas. at signal "D01"
53 testDescriptor.evaluate(secondMeasuredD01VoltMeas1);
54
55 Map<String, MultiSiteDoubleArray> measuredVoltages =
56     measuredResultsHandle.vmeas("").getVoltage();
57 // Evaluate results of third voltage meas. at "D01" and "D02"
58 testDescriptor.evaluate(measuredVoltages, 2);
```

# Usage Examples of “evaluate()”

Examples of “evaluate()” methods that are called with different types of parameters.

```
42 IDcVIResults measuredResultsHandle =
43     measurement.dcVI("D01+D02").preserveResults();
44
45 MultiSiteDoubleArray measuredD01VoltMeas1 =
46     measuredResultsHandle.vmeas("voltMeas1").getVoltage("D01");
47 // Evaluate results of first "voltMeas1" meas. at signal "D01"
48 testDescriptor.evaluate(measuredD01VoltMeas1, 0);
```

Lines 45 – 48: `MultiSiteDouble secondMeasuredD01VoltMeas1 =`

This evaluates the element at index 0 of “measuredD01VoltMeas1”, which contains results of all voltage measurements “voltMeas1” at “D01”.

- This means, for each site SmarTest evaluates the result of the first voltage measurement “voltMeas1” (of type “vmeas”) at the signal “D01”.  
`Map<String, MultiSiteDoubleArray> measuredVoltages = measuredResultsHandle.vmeas("").getVoltage();`
- SmarTest writes the results to the datalog.  
If the format of the datalog is STDF, SmarTest generates a “Parametric Test Record” (PTR), that contains the results of the single signal “D01”.  
`testDescriptor.evaluate(measuredVoltages, 2);`

## Usage Examples of “evaluate()”

Examples of “evaluate()” methods that are called with different types of parameters.

```
42 IDcVIResults measuredResultsHandle =
43     measurement.dcVI("D01+D02").preserveResults();
44
45 MultiSiteDoubleArray measuredD01VoltMeas1 =
46     measuredResultsHandle.vmeas("voltMeas1").getVoltage("D01");
Lines 50 – 53: measuredD01VoltMeas1, 0);
47
48 MultiSiteDouble secondMeasuredD01VoltMeas1 =
49     measuredD01VoltMeas1.getElement(1);
50 // Evaluate results of 2nd "voltMeas1" meas. at signal "D01"
51 testDescriptor.evaluate(secondMeasuredD01VoltMeas1);
52
53
```

- This means, for each site SmarTest evaluates the result of the second voltage measurement "voltMeas1" (of type "vmeas") at the signal "D01".
- SmarTest writes the results to the datalog. If it is in STDF format, it generates a “Parametric Test Record” (PTR), that contains the results of the single signal “D01”.

## Usage Examples of “evaluate()”

Examples of “evaluate()” methods that are called with different types of parameters.

```
42 IDcVIResults measuredResultsHandle =
43     measurement.dcVI("D01+D02").preserveResults();
44
45 MultiSiteDoubleArray measuredD01VoltMeas1 =
46     measuredResultsHandle.vmeas("voltMeas1").getVoltage("D01");
47 This evaluates the element at index 2 of "measuredVoltages", which contains results of all voltage
48 measurements of type "vmeas" at "D01" and "D02".
49 meas. at signal "D01"
```

- This means, for each site SmarTest evaluates the results of the third voltage measurements of type "vmeas" at signals "D01" and "D02".
- SmarTest writes the results to the datalog.

If the format of the datalog is STDF, SmarTest generates a “Multiple-Result Parametric Record” (MPR), that contains the results of multiple signals: “D01” and “D02”.

```
50 MultiSiteDouble secondMeasuredD01VoltMeas1 =
51     measuredD01VoltMeas1.getElement(1);
52
53 Map<String, MultiSiteDoubleArray> measuredVoltages =
54     measuredResultsHandle.vmeas("").getVoltage();
55 measuredVoltages = measuredResultsHandle.vmeas("").getVoltage();
56 measuredResultsHandle.vmeas("").getVoltage();
57 // Evaluate results of third voltage meas. at "D01" and "D02"
58 testDescriptor.evaluate(measuredVoltages, 2);
```



# Data Logging

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

SmarTest 8: Data Logging - 1

## Learning Objectives

- Understand the mechanism of logging data in SmarTest 8.
- Learn and understand how to implement and control data logging.



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

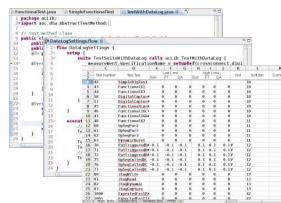
SmarTest 8: Data Logging - 2

# Agenda

- Introduction to data logging.
- Implementation of data logging in test methods: Usage of *test descriptors*.
- Viewing results and data log.

## Basics: Introduction to Data Logging

Test Program



Execution & Results



Data Log



A semiconductor device is usually tested according to a test plan:

- The test program provides the list and the setups of tests.
- The execution of the tests by a tester produces test results.
- Test results are uploaded from the tester channel memory to the workstation memory.
- The test results are evaluated ("pass"/"fail") and written to a datalog file for debug, characterization, correlation, fault analysis, yield learning, etc.
- STDF is widely used as format for storing data logs.
- Results can be viewed directly with the built-in tools of the tester software or with external tools.

# Setup of Data Logging in the Test Program

**Test Table**

The **test table** of a test program lists the pass/fail tests with test limits and data log settings.

*Test tables* are in the „OpenDocument Spreadsheet“ (ODS) format.

**Testflows**



Testflows determine the flow of execution of test suites and their tests. Only settings in the “execute” part of a testflow can overwrite what is given by a *test table*.

**Test Methods**



Test methods contain result retrieval, evaluation and writing results to data log. Any pass/fail test, that should generate dedicated result records in the data log, is implemented by a **test descriptor**.

Note: The *test program file* and the software to control the test cell can contain settings for data logging as well.

## Introduction to Test Descriptors

The *test descriptor* of a pass/fail test is used to:

- Assign to a test an identification number and name.
- Determine the pass/fail state.
- Publish details of the test (results, limits,...) to the data log.
- Optionally determine a software bin to be assigned if the test failed.

The pass/fail state of a single test can depend on

- the overall pass/fail state of a *measurement*, or
- an individual test with specific limits for a *measurement*.

Specific limits can be compared with

- raw test results from the *instruments*, or
- calculated test results.

# Example: Data Logging for Functional Tests

```
1 package slideExamples.dataLogging;
2
3 import xoc.dta.TestMethod;
4
5 /**
6  * Basic test method class to run a digital pattern.
7 */
8 public class FuncTestWithDataLog extends TestMethod {
9     public IMeasurement measurement; // measurement object
10    public IFunctionalTestDescriptor myFtd; // data log object
11    @In public String testSignals = "allIos"; // test signals
12
13    @Override public void update() {
14        // Record cycle pass/fail data when log level is >= 30
15        if (myFtd.getLogLevel() >= 30) {
16            measurement.digInOut(testSignals).result()
17                .cyclePassFail().setEnabled(true);
18        }
19    }
20
21    @Override public void execute() {
22        measurement.execute();
23        IDigInOutResults passFailResults =
24            measurement.digInOut(testSignals).preserveResults(myFtd);
25        releaseTester();
26        myFtd.evaluate(passFailResults); // results to data log
27    }
28
29 }
30 }
```

Mandatory for data logging:  
Declaration of the test descriptor

Test descriptor types:

- functional
- parametric
- scan

Evaluation using a test descriptor:  
Make pass/fail decision and write  
results to data log

Example of a test method for a simple digital functional test.

## Settings of Test Descriptors

It is recommended to set all properties of a *test descriptor* using a *test table*.

The settings can be read via “get..()” functions (and changed via “set..()” functions, which is not recommended) in the test method:

```
// get settings of the test descriptor myFtd
testID   = myFtd.getTestNumber(); // ID
testName = myFtd.getTestName(); // name
testText = myFtd.getTestText(); // description
testFailBin= myFtd.getSoftBinId(); // fail bin
logLevel  = myFtd.getLogLevel(); // data log level
myFtd.evaluate(passFailResults); // results to data log
testResult = myFtd.getPassFail(); // pass/fail
```

Use  
setTestName ("...")  
to control which test  
name is written to  
datalog

The “evaluate()” function operates on test results per site as follows:

- It performs the pass/fail judgement – for *parametric test descriptors* the results are evaluated against associated limits.
- It adds a bin to the bin list if a “fail” was evaluated.
- It pushes results to the data log according to the “logLevel” setting.

# Log Level given by Test Descriptors

Specifies how much information is to be logged.

The screenshot shows a Java code editor with the following code snippet:

```
15@ Override public void update() {  
16    myFtd.  
17    // Re  
18    if (m  
19        setSoftBin() : ITestDescriptor ->  
20        setLogLevel(int logLevel) : TestDescriptor ->  
21        setLogPerCall(boolean value) : IDigitalTestDescriptor  
22    }  
23    @Override  
24    void measure()  
25    IDIn  
26    setTestNumber(int testNumber) : ITestDescriptor ->  
27    IDIn  
28    setTestNumberIncrement(int value) : TestDescriptor ->  
29    IDIn  
30    setTestText(String testText) : ITestDescriptor ->  
31    IDIn  
32    toString() : String -> Object  
33 }  
34 }
```

A tooltip window titled "Press 'Ctrl+Space' to show Template Proposals" is open over the code editor, listing several methods related to log levels. A blue arrow points from the text "Here 'Ctrl+Space' helps to find and understand the properties of test descriptors." to the tooltip window.

**Detailed description of the log levels of test descriptors.**

Log level	Log level text	Description
0	Off	No test result is logged.
10	PassFail	The overall pass/fail result is logged.
20	PerSignalPassFail	Per signal pass/fail result is logged.
30	FailValue	Per signal pass/fail result and failed test values are logged.
40	Value	Per signal pass/fail result and all failing device cycles per signal.

Note: If you set a higher log level in your test method than it is set in the calling testflow, the log level of the testflow is used.

## Input Parameters of “evaluate()”

A parametric test descriptor provides many different evaluate functions. Their input is always a result container that contains multi-site results.

The screenshot shows a Java code editor with the following code snippet:

```
measurement.execute(); // execute measurement  
IDigitResults passFailResults = // retrieve digital results  
    measurement.digInOut(testSignals).preserveResults(myFtd);  
// retrieve results of DC / voltage measurements  
IDCVIResults result = measurement.dcv("") .preserveResults();  
Map<String, MultiSiteDoubleArray> vMeasResults  
    result.iforceVmeas("").getVoltage();  
releaseTester(); // start next test suite  
myFtd.evaluate(vMeasResults, 0);  
// get se  
testID  
testName  
testText  
testFailB  
logLevel
```

A tooltip window is open over the line `myFtd.evaluate(vMeasResults, 0);`, providing detailed information about the `evaluate` method. The tooltip includes the following text:

**result container**  
“index=0”: results of first action are evaluated

Evaluates the values per signal at the specified index and sets the overall pass/fail state. The overall pass/fail state is the boolean AND of the pass/fail states of all sites.

This writes the results to the data log depending on specified log level.

**Parameters:**

- valuesPerSignal** The value map, that maps the site-specific values to the corresponding signal.
- index** The zero-based index of the site-specific values in the array.

Press 'Tab' from proposal table or click for focus

## Variants of “evaluate()”

All the different variants of the “evaluate()” function of *parametric test descriptors* work on the same two principles:

- Variants with respect to the generated STDF records:
  - If the type of the input parameter is a *MultiSite data type*, then data for a single signal is provided:  
In the STDF data log a “Parametric Test Record” (PTR) is generated.
  - If the type of the input parameter is a “Map”, then the keys are signals and for each signal the “Map” contains results:  
In the STDF data log a “Multiple-Result Parametric Record” (MPR) is generated.  
This is the default, if the STDF configuration setting “forcePtr” is “false”.
- Variants that handle an array (or in a “Map” multiple arrays) of a *MultiSite data type*:
  - As input is additionally needed the index of the element to evaluate.

## Viewing Results

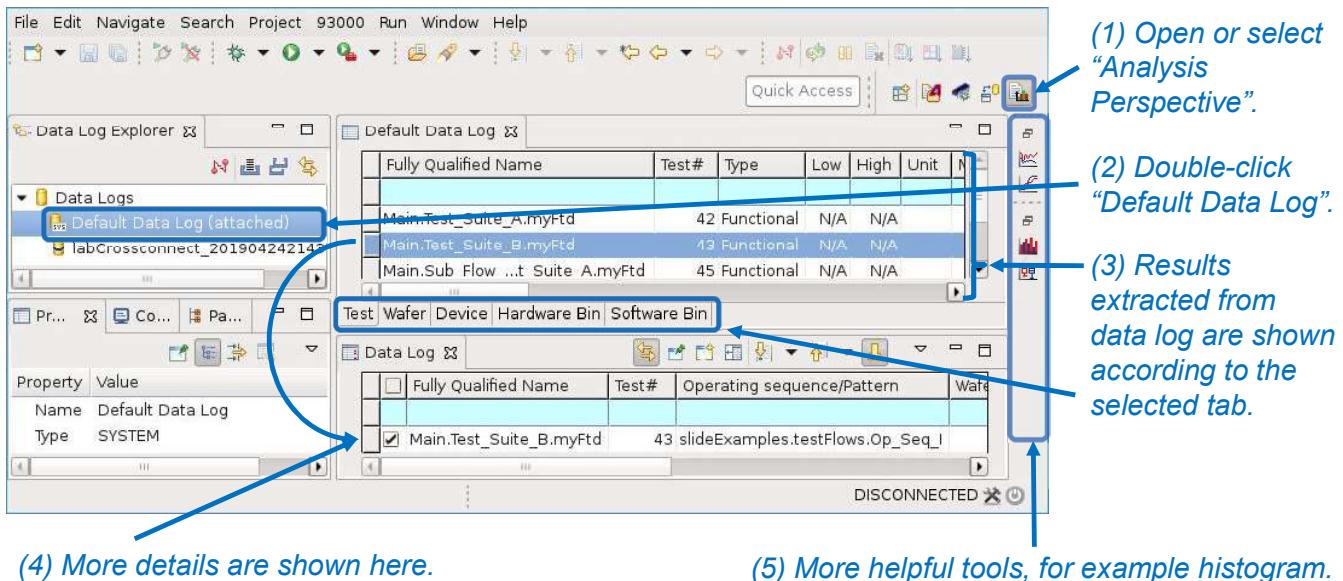
The *result* view shows logged data.

Fully Qualified Name	Site	Test#	Type	Operating sequence/Pattern	P/F
Main.Test_Suite_A.myFtd	1	42	Fun...	slideExamples.testFlows.Pat...	Pass
Main.Test_Suite_B.myFtd	1	43	Fun...	slideExamples.testFlows.Op...	Pass
Main.Sub_Flow...Suite_A.myFtd	1	45	Fun...	slideExamples.testFlows.Pat...	Pass
Main.Sub_Flow...Suite_D.myFtd	1	46	Fun...	slideExamples.testFlows.Op...	Pass
Main.Sub_Flow...Suite_E.myFtd	1	47	Fun...	slideExamples.testFlows.Op...	Pass
Main.Sub_Flow...Suite_C.myFtd	1	44	Fun...	slideExamples.testFlows.Pat...	Pass
Main.Sub_Flow...Suite_A.myFtd	1	45	Fun...	slideExamples.testFlows.Pat...	Pass

These tabs allow to select the granularity of shown results.  
Here tab “Test” is selected.

This column provides the fully qualified name of test descriptors:  
The calling hierarchy (with respect to testflows and test suites) and the name of the test descriptor.

# Viewing Data from Data Log



## Data Log Formatters

In SmarTest, test results are firstly captured in the “event data log” or EDL. It is a binary format of the raw test-event data stream from instruments. An EDL file can be loaded and its data reviewed in SmarTest.

SmarTest 8 provides various data log formatters converting the test data:

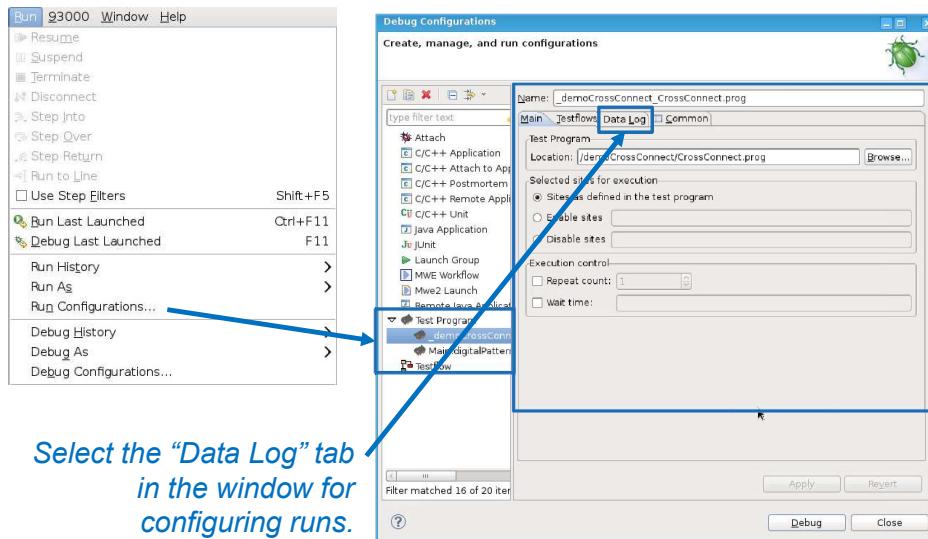
- STDF: Standard data log formatter that converts data to STDF.
- ASCII: Converts the test-event data to text data log, similar to the output in the data log console view.
- SUMMARY: A summary of the test-event data in text form.

Additionally, SmarTest 8 supports that user can write their own data log formatter for custom formats:

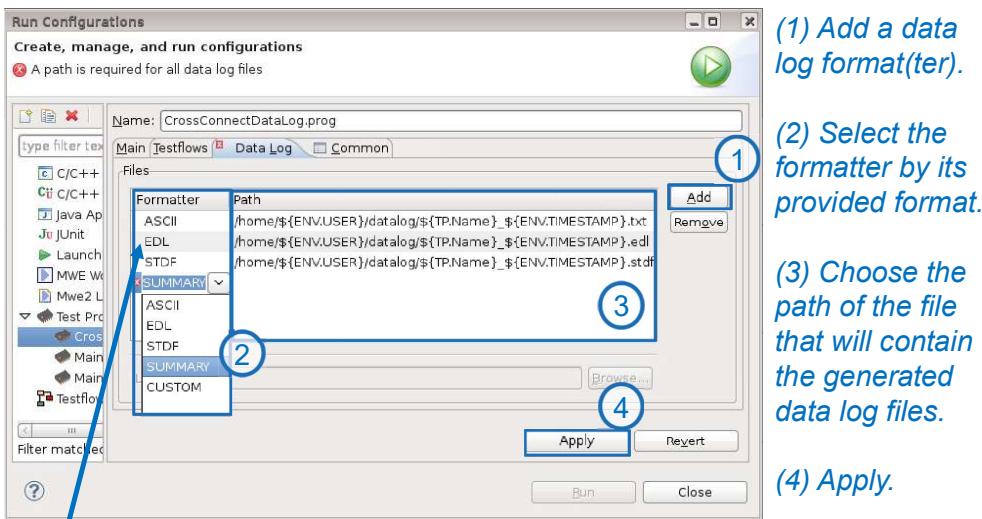
- CUSTOM: Data log formatter provided by the user.

# Run Configurations and Data Logging

The run configuration includes settings for the formatter of data logs.



## Setting Up the Formatter for Data Logging



# Summary - What you should have learned

- *Test descriptors* are used to store the various properties of a test and by calling their “evaluate()” function a pass/fail decision is made, a fail bin might be assigned and test results are pushed to the datalog.
- Properties of test descriptors can be set in test tables, test methods and testflows – in particular to determine the level of detail for logged result data.  
The recommendation is to assemble all settings in test tables.
- The result data pushed by the “evaluate()” function is a stream of raw binary data in the EDL format.
- According to the setup in the run configuration, this stream can be processed by datalog formatters to bring the result data into a specific format and store it in a file with an user defined name.
- SmarTest provides datalog formatters for EDL, STDF, SUMMARY and ASCII and supports customized datalog formatters.
- The purpose of the analysis perspective is to facilitate viewing and analysing the test results stored in datalog files.



# Binning

SmarTest 8.2.5 Training

January 2020

# Learning Objectives

- Understand the binning mechanism in SmarTest 8.
- Learn how to implement the binning.

# Agenda

- Introduction
- SmarTest 8 binning concept
- Setting up a list of available bins (*bin table*)
- Assigning bins to devices while executing the tests of a test program
- Selecting the *final bin*
- Summary

# Introduction

- Binning is about giving handlers the data to correctly separate the good from various forms of bad devices at the end of a test program.
- A good binning strategy can provide a way to quickly identify and focus on production issues without analyzing large amount of data.
- SmarTest 8 provides a flexible binning concept with a simple implementation which enable users to strategize their binning.
- It is recommended to define bins in a *test table*.
- If a device fails a test, then typically a setting within the corresponding *test descriptor* will control which bin is assigned to the device.

## Binning Concept

### **Bin table:**

A table listing available bins and their properties.  
It is generated in the *auxiliary flow* "PreBind"  
which is executed only once.

### **Bin list** (per site & execution / device):

A list of bins that have been assigned to a certain site/device during execution.

- Execution begins with an empty *bin list*.
- During execution the test results are evaluated.
- The *bin list* is subsequently filled with software bins that are defined in the *bin table*.

### **Selection of the final bin** (per site & execution / device):

After all required tests have been executed, the *final bin* is selected from the *bin list*.

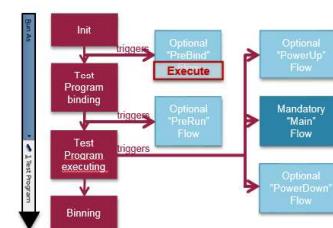
The associated hardware bin of the *final bin* is sent to the handling equipment.

The selection algorithm for the *final bin* can be customized.

### **Execution: "PreBind" Flow**

The optional "PreBind" flow is setup and then executed.

Typically this flow is used to read *test tables* that are used to set test limits, levels of data logging, hardware and software bins, etc.



#### **Execution steps**

0. Load
1. Initialization.
2. **Exec "PreBind" flow**
3. "setup()" method
4. Link.
5. Bind.
6. "update()" method
7. "execute()" method
  - a) "PreRun" flow
  - b) "PowerUp" flow
  - c) "Main" flow
  - d) "PowerDown" flow
8. Binning.

# Bin Table

The *bin table* of a test program defines all available bins with the associated properties:

- Hardware bins properties:  
“name”, “number”, “pass/fail”.
- Software bin properties:  
“name”, “number”, “pass/fail”, “color”, “priority” and corresponding hardware bin.

The *bin table* is usually generated from a *test table* as follows:

A test suite in the *auxiliary flow* “PreBind” reads the *test table*, which is stored in the “Open Document Spreadsheet” format with suffix “.ods”.  
It applies the settings given in the *test table*.

User defined test method code can also generate a *bin table*, but it must be executed by a test suite that is defined and executed in the *auxiliary flow* “PreBind”.

## Default Bins

SmarTest provides three default bins:

- **Overall default bin**  
The *overall default bin* is assigned to a device if a test fails and no fail bin is defined.  
The software and hardware bin numbers are 0. It is a fail bin of red color.
- **Default pass bin**  
If specified and the *bin list* is also empty at the end of the execution then the *default pass bin* is added.  
If not specified, the *overall default bin* is used for this purpose.
- **Default alarm bin**  
If specified, the *default alarm bin* is added to the *bin list* of a device
  - If an alarm is raised during the test of the device.
  - If the alarm configuration enables binning on this type of alarm.If not specified, the *overall default bin* is used for this purpose.

# Bins Defined in Test Table Sheets

Software bins:

1	Software Bin Name	Software Bin	Hardware Bin	Result	Color	Priority	Comment
2	DEFAULT	0	0	FAIL	MAGENTA	1	Definition of "overall default bin"
3	DEFAULT_PASS	2	1	PASS	CYAN	3	Definition of "default pass bin"
4	DEFAULT_ALARM	3	0	FAIL	YELLOW	15	Definition of "default alarm bin"
5	goodBin	1	1	PASS	GREEN	10	
6	passContinuity	9	2	PASS	CYAN	2	
7	failContinuity	10	2	FAIL	RED	40	
8	failDC	12	3	FAIL	RED	30	

Hardware bins:

1	Hardware Bin	Hardware Bin Name	Result	Comment
2	0hardDefault	FAIL		
3	1hardPass	PASS		
4	2hardMisc	FAIL		
5	3hardDC	FAIL		

## Bin List

The *bin list* contains the software bins that were assigned to the device while the testflows and test suites of the program were executed.

It is always empty at the beginning of the execution of the *main flow*.

It is filled while executing the *main flow* in the following manner:

- Pass/fail tests based on *test descriptors*.  
In the case of a failing test, a software bin is added to the *bin list*.  
Its *test descriptor* allows to define the software bin or otherwise the *overall default bin* is added.
- In the “execute” section of testflows, the command “`addBin(binNumber)`” explicitly adds a software bin to the *bin list*. Example:  
`“if (!Test_Suite_A.pass) { addBin(42); stop(); }”`
- In test methods, “`context.binning().add(site,binNumber);`” adds a bin to the *bin list* for the given site.
- One of the special *default bins* can be automatically added.

# Setting Bins for Failing Tests

A *test table* can be used to consolidate the setup for binning in one file.

It allows to define a software bin to be added to the *bin list* for every failing test.

These settings are displayed below in the “Soft Bin” column of the “Tests” tab.

In general, based on *test descriptors*, the *test table* allows to define the settings of all tests .

	B	C	D	E	F	G	H	I	J	K	L
1	Test	Test Number	Test Text	Low Limit			High Limit			Unit	Soft Bin
2				Sort	FT	QA	Sort	FT	QA		
3	funcTestDescriptor	42	Functional test								10
4	parametricTestDescriptor1	421	Voltage mV	0.9	0.9	0.95	1.1	1.1	1.05 V		11
5	parametricTestDescriptor2	422	Voltage mV	0.8	0.8	0.85	1	1	0.95 V		11
6	parametricTestDescriptor3	423	Voltage mV	2.4	2.4	2.45	2.6	2.6	2.55 V		11

Note:

SmarTest provides the flag **"suppressBinning"** for test suites.

If it is set to “true” then for all the *test descriptors* of a test suite no software bin will be added to the *bin list* even if a test fails.

## Selecting the Final Bin

Binning takes place after all tests have been executed.

- For each site the *final bin* is selected from its *bin list*.
- The associated *hardware bin* is determined from the *final bin*.
- These bins are *logged* and *communicated* to the handling equipment.

A customized *final bin* selection algorithm may also be implemented within a test method:  
A software bin can be explicitly set as “final”.

Otherwise the following default algorithm is used to determine the *final bin*:

- If the *bin list* is empty, then the *final bin* is the *default pass bin*.
- If the *bin list* contains one bin, then this bin is the *final bin*.
- If the *bin list* contains multiple bins, then the bin with the highest priority value is the *final bin*.  
If multiple bins share the highest priority value, then the *final bin* is the bin with the highest priority value, that was firstly added to the *bin list*.

# Customized Algorithm to Select the Final Bin

In test methods a software bin can be explicitly set to “final” as follows:

```
"context.binning().setFinalBin(site,binNumber);"
```

It is recommended that a test suite, executed after all other tests have been executed, calls the test method that selects and sets a software bin as “final”.

If multiple software bins were set to “final” within test methods, then the last bin to be set as “final” during test program execution, is considered the *final bin*.

```
"context.binning().getBinList(site);"
```

gives access to the *bin list* of a specific site.

This might be needed, for example, if the customized algorithm determines the *final bin* from the software bins collected during the execution of the *main flow*.

## Summary - What you should have learned

- The *bin table*, generated in the “PreBind” *auxiliary flow*, defines all expected bins to be used. It is recommended to use a *test table* for this step.
- For each site the dedicated *bin list* is filled during the execution of the *main flow* of a test program, using one of these four mechanisms:
  1. Bins are implicitly added to the list if a test fails and the “SoftBinId” property of the corresponding *test descriptor* is set.  
It is recommended to use a *test table* for these settings.
  2. Bins can be explicitly added for all active sites in a *testflow file*.
  3. Bins can be explicitly added for specified sites in a test method.
  4. SmarTest adds a special default bin. For example, it might happen if an alarm occurs.
- The *final bin* is selected after all required tests have been executed:
  1. If bins are explicitly set as “final”, the *final bin* is the bin of the last setting.
  2. Otherwise, the default algorithm applies:  
The bin with the highest priority, set before other bins with the same priority, is the *final bin*.
  3. If the *bin list* is empty, then the *final bin* is the *default pass bin*.



# Usage of Test Tables

SmarTest 8.2.5 Training

January 2020

## Learning Objectives

- Understanding the purpose of *test tables*.
- Know how to use a *test table*

# Agenda

- Purpose and contents of *test tables*
- Generation and usage of *test tables*

## Purpose of Test Tables

When a test program is executed,

- one or several testflows are called;
- the testflows define the order of execution of various test suites;
- the executed test suites perform measurements that produce results;
- the results must be judged, data logged and bins are assigned.

The purpose of *test tables* is:

**A single, centralized setup entity that consolidates all settings for limits, binning and data logging of a test program or of an IP block.**

Consolidating all the settings in a single spreadsheet document facilitates the process of working on a test program through its various stages:

Development, debugging, characterization, qualification, release for production (wafer sort / final test), maintenance,...

# Usage of Test Tables: Overview

*Test tables* are not mandatory, but recommended:

Instead of using *test tables*, the corresponding settings could be also implemented in testflows and test methods.

But then these settings are spread over many files in various folders.

If SmarTest executes a test program with a *test table*, then the test program contains a test suite that reads it.

The test suite is implemented in the *auxiliary flow* "PreBind".

After reading, SmarTest applies the settings of the *test table*.

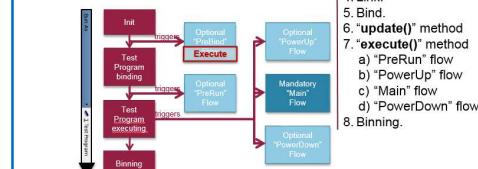
Recommendation: Do not overwrite settings of a *test table* in test methods.

- It invalidates the concept of a centralized table for all these settings.
- It leads to wrong assumptions how pass/fail tests are executed/evaluated.
- It might result in errors in test programs that can cause high costs.

## Execution: "PreBind" Flow

The optional "PreBind" flow is setup and then executed.

Typically this flow is used to read *test tables* that are used to set test limits, levels of data logging, hardware and software bins, etc.



- Execution steps
0. Load
  1. Initialization.
  2. Exec "PreBind" flow
  3. "setup()" method
  4. Link.
  5. Bind.
  6. "update()" method
  7. "execute()" method
    - a) "PreRun" flow
    - b) "PowerUp" flow
    - c) "Main" flow
    - d) "PowerDown" flow
  8. Binning.

## Overview continued: Content of Test Tables

A *test table* is a spreadsheet document in the "Open Document Spreadsheet" format (.ods), that contains multiple sheets:

	B	C	D	E	F	G	H	I	J	K	L
1											
2	Test	Test Number	Test Text		Low Limit		High Limit				
3	funcTestDescriptor			Sort	FT	QA	Sort	FT	QA	Uni	Soft Bin
4	parametricTestDescriptor1	421	Voltage m	0.9	0.9	0.95	1.1	1.1	1.05	V	10
5	parametricTestDescriptor2	422	Voltage m	0.8	0.8	0.85	1	1	0.95	V	11
6	parametricTestDescriptor3	423	Voltage m	2.4	2.4	2.45	2.6	2.6	2.55	V	11

- The optional "Master\_File" sheet allows to reference other test tables.
- The two optional "Bins" sheets define the bin table.
- The mandatory "Tests" sheet contains setups for pass/fail tests.
- The optional "Profile" sheet to set parameters of test suites and testflows.
- The optional "Alarm\_Config", "STDF\_Config" and "TPVariables" sheets.
- The optional "Specs" sheet to set specification variables in "PreStart" flow.

# Usage of Test Tables: Test Program File

To read a *test table*, the *test program file* must define the *auxiliary flow* of type “PreBind”.

```
1 testprogram TestTableExamples {
2     dutboard = crossconnect.common.CrossConnect4Site;
3
4     // optional auxiliary testflow to read test tables
5     testflow PreBind {
6         flow = crossconnect.common.ReadTestTable;
7     }
8
9     // mandatory main testflow
10    testflow Main {
11        flow = crossconnect.common.Demo;
12    }
13
14 }
15 }
```

Call of the “PreBind” flow in the test program file

Note:

1. When triggering the execution of a test program or a subflow, then as one of the initial steps SmarTest always executes the *auxiliary flow* “PreBind” if defined. This is also true for debug.
2. If the *test table* contains settings for specification variables, the *auxiliary flow* “PreStart” must be defined as well to read the *test table* a second time after *specification files* have been loaded.

## Usage of Test Tables: “PreBind” Testflow

The testflow must contain a test suite that reads the *test table*.

SmarTest provides a special test method for this:

```
1 flow ReadTestTable {
2
3     setup {
4         suite importTestTable calls
5             com.advantest.itee.testtable.testmethod.TestTableReader {
6                 testTablePath = "crossconnect/common/MasterTestTable.ods";
7                 limitCategory = "QA";
8                 logLevelCategory = "QA";
9                 errorLogPath = "/tmp";
10                forceImport = true;
11                useCache = true;
12            }
13
14     }
15
16     execute {
17         importTestTable.execute();
18     }
19 }
```

“TestTableReader”:  
The test method to read a test table

Path to the test table relative to its  
source folder

Various categories reflect different  
stages of the test program

Store test table  
data in a cache

This setting forces to continue the  
execution even if a problem occurs

If problems occur, a commented test  
table will be written in this folder

# The Test Method “TestTableReader”

The test method “TestTableReader” checks, that the *test table*

1. includes settings for each test (i.e. *test descriptor*) in the test program;
2. contains no two settings that are overlapping;
3. lists only settings that can be mapped to the test program;
4. contains only settings that are complete and consistent.

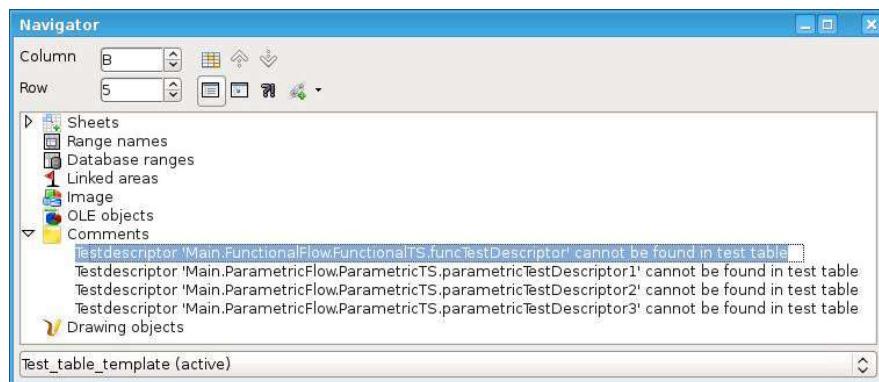
If any of the checks fails, then the “TestTableReader”

- dumps warnings to the console;
- generates a new *test table* with additional comments and warning messages for each failed check, for example for check (1):
  - lists all *test descriptors* missing the mandatory settings;
- stores the new *test table* with additional comments in the folder specified by the “errorLogPath” parameter;
- if the parameter “forceImport” is “false”: Stops test program execution.

## Warnings of the Commented Test Table

To view the detailed warning messages in the commented *test table* in “LibreOffice”, press “F5” to start its “Navigator” and check the comments.

Double-click on a comment to bring the focus to the cell that is related to the warning message.



## “TestTableWriter”: Generation of a Test Table

An (initial) test table as shown below can be generated by a test suite, that calls the test method “TestTableWriter” as shown below.

Its “Tests” sheet contains a list of the test descriptors of all pass/fail tests found in the test program.

In SmarTest 8.2.0, this test method works only in the *auxiliary flow* “PreStart”.

In later revisions it will also work in the *main flow* and in other *auxiliary flows*.

The screenshot illustrates the process of generating a test table. On the left, a code editor shows a SmarTest flow script named "WriteTestTable.flow". The script defines a flow with a setup section containing a call to "com.advantest.itee.testtable.testmethod.TestTablewriter". The "testsSheetByFlow" parameter is set to "false". An annotation with a blue arrow points to this line, stating "Set to ‘true’ to get dedicated ‘Tests’ sheets per testflow". On the right, a LibreOffice Calc window titled "ExportedTestTable.ods" displays a table with six rows of test descriptors. The table has columns for "Test Suite", "Test", and "Test Number". The data is as follows:

Test Suite	Test	Test Number
Main.FunctionalFlow.FunctionalTS	funcTestDescriptor	1
Main.ParametricFlow.ParametricTS	parametricTestDescriptor1	1
Main.ParametricFlow.ParametricTS	parametricTestDescriptor2	1
Main.ParametricFlow.ParametricTS	parametricTestDescriptor3	1

Annotations with blue arrows point to the "Test Suite" column and the "Test" column, both labeled "Fully qualified names of test descriptors found in the test program".

## Summary - What you should have learned

- The benefit of the *test table* is that it is a centralized, single file that consolidates various settings. Otherwise, these settings would be spread over many files.
- The *test table* is optional, therefore the usage must be explicitly implemented.
- The *test table* is read in the *auxiliary flow* “PreBind”, that executes a test suite calling a specific test method provided by SmarTest.
- This test method performs checks in order to help to assemble a correct, complete and consistent *test table*.
- The *test table* supports the definition of categories reflecting the different stages of test program development and their specific settings for tests and data logging.
- You can write your own test method to read a *test table* in a customized format.



# Content of Test Tables

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest 8: Content of Test Tables - 1

## Learning Objectives

- Know how a *test table* defines
  - the list of available software and hardware bins;
  - the limits and fail bins for tests;
  - the settings for data logging and verbosity of messages dumped by test methods;
  - the settings of input parameters of test suites and testflows;
  - the configuration of alarm handling and STDF format;
  - the setting of test program variables and variables in *specification files*.
- Learn how a *test table* can refer to other test tables
  - to facilitate reuse;
  - to facilitate collaborative development.



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest 8: Content of Test Tables - 2

# Agenda

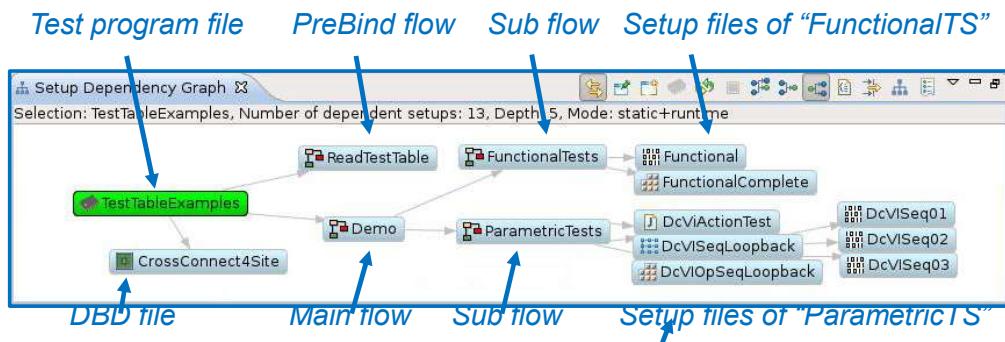
- Contents of *test tables*:
  - Hardware and software bins.
  - Setup to evaluate pass/fail tests, i.e. tests with limits, fail bin, test ID number, etc.
  - Settings for data log and other test suite parameters.
  - Configuration of alarm handling and STDF format.
  - Setting of test program variables and variables of *specification files*.
  - References to other *test tables*.
- Recommendations for modular test programs

## Example Setup to Show Content of Test Tables

The following sheets are shown in order to illustrate the various settings stored in a test table. The contents of the sheets are based on an example of a small test program.

Its *main flow* calls two subflows “FunctionalTests” and “ParametricTests”.

- The subflow “FunctionalTests” executes the test suite “FunctionalTS” with one pass/fail test.
- The subflow “ParametricTests” executes the test suite “ParametricTS” with three pass/fail tests.



## Content: Definition of Bins

- The names of the **optional sheets** in the test table are “Software\_Bins” and “Hardware\_Bins”.
- The sheets define bins that are added to the *bin table*.
- The test method “TestTableReader” checks, if the bin settings of the two sheets and the “Tests” sheet are consistent.

The screenshot shows two LibreOffice Calc windows side-by-side. Both windows have the title "SingleTestTable.ods - LibreOffice Calc".

**Software\_Bins Sheet:**

	A	B	C	D	E	F
1	Software Bin Name	Software Bin	Hardware Bin	Result	Color	Priority
2	goodBin		1	1 PASS	GREEN	10
3	functionalFail		10	2 FAIL	RED	30
4	parametricFail		11	3 FAIL	RED	50
5						

**Hardware\_Bins Sheet:**

	A	B	C
1	Hardware Bin Name	Hardware Bin	Result
2	hardPASS		1 PASS
3	hardFuncFAIL		2 FAIL
4	hardParamFAIL		3 FAIL
5			

## Content: Definition of Pass/Fail Tests

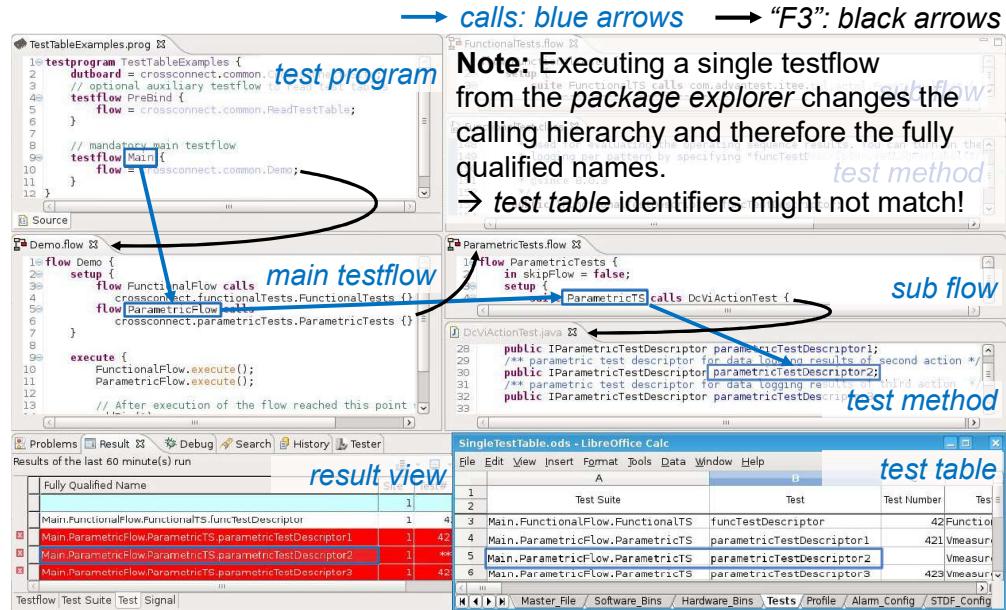
- The name of the **mandatory sheet** in the *test table* is “Tests” or “<UserDefinedPrefix>\_Tests”.
- The sheet defines settings of pass/fail test, i.e. the properties of the associated *test descriptors*.
- The columns allow to set for a *test descriptor* a unique test number, a test text, low & high limits for parametric tests with a unit, a software bin for a failing test and a comment.
- Test descriptors* are identified by fully qualified names that are derived from the call hierarchy used when executing the test program.
- If the sheet’s name is “<UserDefinedPrefix>\_Tests”, then “<UserDefinedPrefix>” is added as prefix to the fully qualified names.

The screenshot shows a LibreOffice Calc window with the title "SingleTestTable.ods - LibreOffice Calc".

**Tests Sheet:**

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Test Suite	Test	Test Number	Test Text	Low Limit			High Limit			Unit	Soft Bin	Comment
2					Sort	FT	QA	Sort	FT	QA			
3	Main.FunctionalFlow.FunctionalTS	funcTestDescriptor	42	Functional test									10
4	Main.ParametricFlow.ParametricTS	parametricTestDescriptor1	421	Vmeasurement 1	0.9	0.9	0.95	1.1	1.1	1.05V			11
5	Main.ParametricFlow.ParametricTS	parametricTestDescriptor2	422	Vmeasurement 2	0.0	0.0	0.05	1	1	0.95V			11
6	Main.ParametricFlow.ParametricTS	parametricTestDescriptor3	423	Vmeasurement 3	2.4	2.4	2.45	2.6	2.6	2.55V			11

# Fully Qualified Names for Test Descriptors



## Identifiers for Fully Qualified Names of Test Descriptors (1)

The entries of optional columns left of the "Test" column are added as prefix to the entry of "Test" to form the identifiers.

If the sheet's name has a prefix, then it is added as prefix to all identifiers.

Here "Main" is added.

The explicit wildcard "\*" covers multiple levels of the execution hierarchy.

*This identifier matches all test descriptors of the test program, that are named "parametricTestDescriptor2".*

1		Test
2		
5	Main.ParametricFlow.ParametricTS.parametricTestDescriptor2	
1		Test Suite
2		
5	Main.ParametricFlow.ParametricTS	parametricTestDescriptor2
1	Testflow	Test Suite
2		
5	Main.ParametricFlow	ParametricTS parametricTestDescriptor2
1	Testflow	Test Suite
2		
5	ParametricFlow	ParametricTS parametricTestDescriptor2
1		Test Suite
2		
5	**.ParametricTS	parametricTestDescriptor2
1		Test Suite
2		
5	**	parametricTestDescriptor2

## Specifying Fully Qualified Names (2)

The wildcard “\*” covers only a single level of the execution hierarchy.

1	Test Suite	Test
2		
5	Main.*.ParametricTS	parametricTestDescriptor2

Master\_File Software\_Bins Hardware\_Bins Tests Profile

This identifier matches all test descriptors named “parametricTestDescriptor2”, that are called by a test suite “ParametricTS” of a testflow that is directly called by the main flow.

If an identifier is incomplete, then the settings of its row are applied to all *test descriptors* with a fully qualified name, that contains the incomplete identifier as suffix.

1	Test Suite	Test
2		
5		parametricTestDescriptor2

Master\_File Software\_Bins Hardware\_Bins Tests Profile

Note:

If multiple *test descriptors* in the test program match an incomplete identifier, then the test method “TestTableReader” throws a warning and writes out a commented *test table*.

## “Tests” Sheet: Different Categories for Limits

The *test table* works with user defined categories which allows for limit settings of wafer sort, final test or other scenarios.

The input parameter “limitCategory” of the test method “TestTableReader” is used to select a category from which the corresponding limits will be set.

The names of the categories can be freely selected.

“Sort”, “FT” and “QA” are examples of the *test table* template of the TDC.

SingleTestTable.ods - LibreOffice Calc													
File Edit View Insert Format Tools Data Window Help													
1	A	B	C	D	E	F	G	H	I	J	K	L	M
	Test Suite	Test	Test Number	Test Text	Low Limit		High Limit		Unit	Soft Bin	Comment		
2					Sort	FT	QA	Sort	FT	QA			
3	Main.Fu	fur		42 Functional test									10
4	Main.Pa	par		421 Vmeasurement 1	NA	0.9	0.95	1.1	1.1	1.05V			11
5	Main.Pa	par		422 Vmeasurement 2	0.8	0.8	0.85	1	1	0.95V			11
6	Main.Pa	par		423 Vmeasurement 3	2.4	2.4	2.45	NA	2.6	2.55V			11

Use “NA” to set no low or no high limit

# Content: Test Suite and Testflow Parameters

- The name of the **optional sheet** in the *test table* is “Profile” (+optional prefix).
- It allows the setting of *test descriptor* log levels, and parameters of test suites and test flows.
- The parameters are identified by fully qualified names relative to the source folders, optionally with wildcards, a prefix to the sheet’s name and additional columns on the left – as in the “Tests” sheet.
- The input parameter “logLevelCategory” of the test method “TestTableReader” determines what “Value” column is valid.

	A	B	C	D	E
1	Parameter	Value			Comment
2		Sort	FT	QA	
3	FunctionalFlow.FunctionalTS.funcTestDescriptor.loglevel	10	10	40	Affects only specified test descriptor
4	ParametricFlow.**.loglevel	10	10	40	Affects all test descriptors of ‘ParametricFlow’
5	**.messageLogLevel	5	5	30	Affects all test suites and testflows
6	FunctionalFlow.FunctionalTS.messageLogLevel	8	8	30	Overwrites settings above for specified test suite
7	FunctionalFlow.FunctionalTS.bypass	FALSE	TRUE	TRUE	Sets parameter of specified test suite
8	ParametricFlow.skipFlow	FALSE	TRUE	TRUE	Sets input parameter of specified testflow
9	ParametricFlow.ParametricTS.testSignals	R00+R01	R00	R00	Sets input parameter of test method of test suite

Master\_File / Software\_Bins / Hardware\_Bins / Tests / Profile / Alarm\_Config / STDF\_Config / Specs / TPVariables / +

# Content: Alarm, STDF Format, TP Variables

The **optional sheet** named “Alarm\_Config” allows to configure how SmarTest handles alarms.

1	Alarm Name	Enable	Force Fail	Software Bin	Enable Binning	Comment
2	LIMIT_EXCEEDED	Y	Y	3	Y	
3	CURRENT_CLAMP	Y	Y	3	Y	
4	VOLTAGE_DROP	Y	Y	3	Y	

Tests / Profile / Alarm\_Config / STDF\_Config / Specs / TPVariables / +

Alarms of given type If this alarm occurs, it is a fail and are not suppressed. bin “3” will be added to bin lists.

The **optional sheet** named “STDF\_Config” allows the format configuration of STDF files.

1	Configuration	Value	Comment
2	stdfVersion	V4-2007.1	Latest and greatest STDF version
3	testText	TESTSUITE:TEST_TXT	
4			

Tests / Profile / Alarm\_Config / STDF\_Config / Specs / TPVariables / +

The **optional sheet** named “TPVariables” allows the setting of variables values which are defined in the *test program file*.

1	Test Program Variables	Value	Comment
2	Sort	FT	QA
3	ConfigName	demoBasicSort	demoBasicFT
4	SYS.LOT_TYPE	PACKAGE_TEST	WAFER_TEST

Tests / Profile / Alarm\_Config / STDF\_Config / Specs / TPVariables / +

# Content: Specification Variables

- The name of the **optional sheet** in the *test table* is “Specs” (+optional prefix).
- Overwrites values of variables assigned in *specification files*.
- A variable is identified by the fully qualified name of the corresponding *specification file* and the variable name delimited by a dot.  
Note, this fully qualified name is not about the call hierarchy; it is about the path of the *specification file* relative to the **source folders** (without suffix “.spec”)
- The input parameter “specCategory” of the test method “TestTableReader” determines what “Value” column is valid.

If values of this sheet should be assigned to specification variables, then the *test table* must be read in the *auxiliary flow* “PreBind” and again in the *auxiliary flow* “PreStart”.

1	Spec Variables	Value			Unit	Comment
		Sort	FT	QA		
3	crossconnect.simpleTests.Functional.vcc	1.4	1.2	1.3 V		
4	crossconnect.simpleTests.Functional.IV_Swing	1.4	1.2	1.3 V		
5	crossconnect.simpleTests.Functional.OV_Swing	0.4	0.2	0.3 V		

Sets variable “vcc”  
in specification file  
“Functional.spec”

# Content: References to Other Test Tables

- The name of the **optional sheet** in the *test table* is “Master\_File”.
- Contains in column “Files” references (relative to the source folders) to other *test tables* that will be read as well.
- Using a *test table* that refers to multiple, specific *test tables* instead of one single *test table* supports modular test programs:
  - It facilitates collaborative development;
  - It allows for a dedicated IP block to setup a *test table* that can easily be reused.

MasterTestTable.ods - LibreOffice Calc	
File	Edit
A	B
1	Files
2	crossconnect/functionalTests/FunctionalTestTable.ods
3	crossconnect/parametricTests/ParametricTestTable.ods

FunctionalTestTable.ods - LibreOffice Calc			
A	B	C	D
1	Test Suite		
2		Test	↑↓
3	FunctionalLTS	funcTestDescriptor	42 Functional test

# Recommendations: Modular Test Programs

- **Test program setup:**

Each module has its own *test table* referenced by a “master” *test table*.

- **Test program development:**

To make changes in the execution hierarchy simple, identifiers should be specified without wildcards as prefix, only relative to the module level and not from the top level.

Example:

Instead of the fully qualified name

```
Main.ModuleMainFlow.ModuleSubflow.TestSuite.TestDescriptor  
rely on the implicit mechanism to match incomplete identifiers and use  
ModuleSubflow.TestSuite.TestDescriptor
```

- **Test program release:**

- Use the test method “*TestTableWriter*” to generate a *test table* with fully qualified names without wildcards for the final check of correctness.
- Review this new *test table* and then use it.

Set the input parameter “*forceImport*” to “*false*” for the test method “*TestTableReader*”.

## Summary - What you should have learned

- Various settings can be consolidated in the *test table*:
  - Software and hardware bins.
  - Setting of tests stored in *test descriptors*.
  - Level of detail for data logging of tests.
  - Input parameters and system flags for testflows and test suites.
  - Configuration of alarms.
  - Configuration of the STDF file format.
  - Values for test program variables and specification variables.
- Only values can be set for these test program variables, that are defined in the *test program file*.
- To set specification variables, the *test table* must be read in again in the *auxiliary flow* “PreStart”.
- For some types of settings different categories for the values can be defined.
- Test descriptors, testflows and test suites are identified by their fully qualified name with respect to the calling hierarchy.



# Test Program Execution and Debug

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

SmarTest Test Program Execution and Debug - 1

## Learning Objective

- Get a brief overview of navigating through test programs in SmarTest
- Learn how to execute test programs and how to efficiently debug test programs
- Know how SmarTest interacts in a production environment



January 2020

All Rights Reserved - ADVANTEST CORPORATION

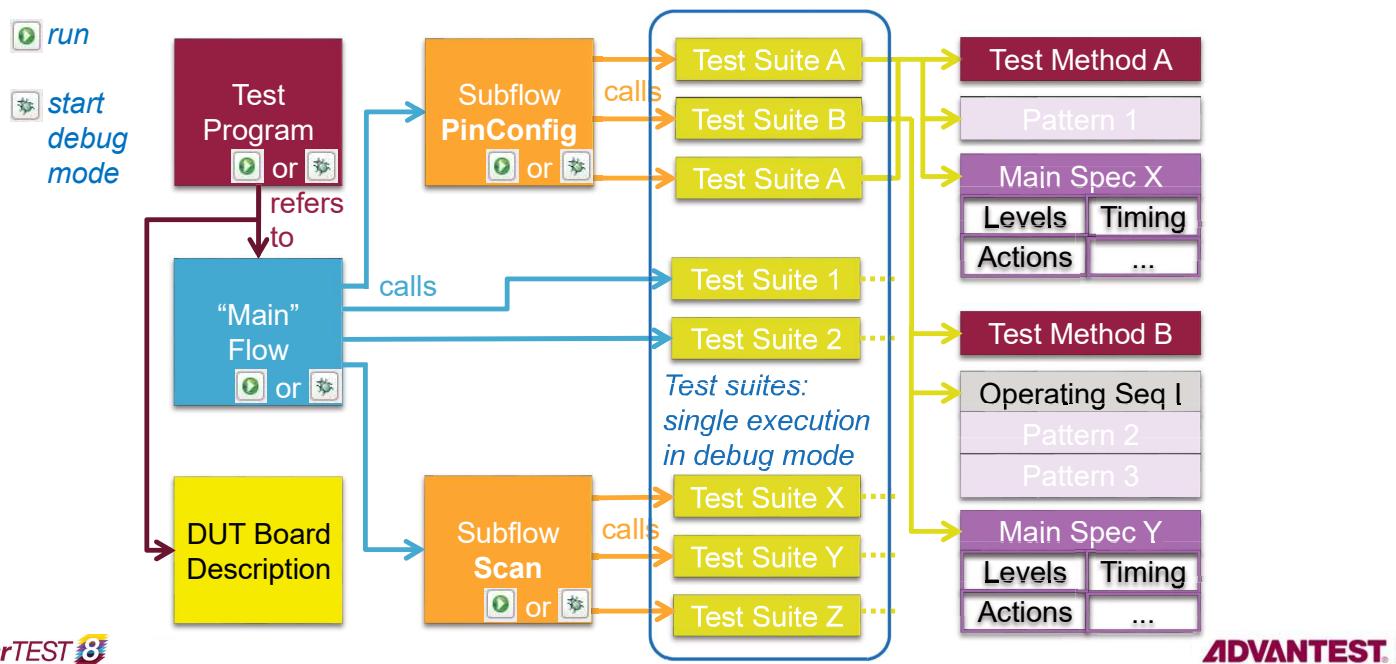
**ADVANTEST**

SmarTest Test Program Execution and Debug - 2

# Agenda

- Navigation through the setup data of a test.
- Test execution and configuration.
- Starting debug of a test.
- Execution of a test program with its *auxiliary flows* in a production environment.

## Test Program in SmarTest 8: Building Blocks

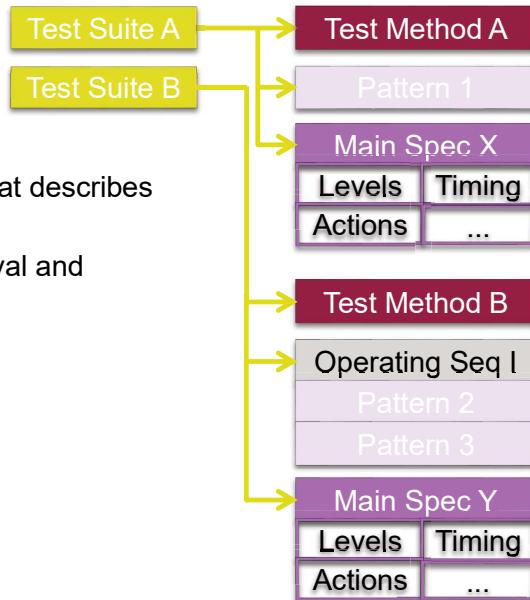


# Setup Data of Test Suites

A test suite is the atomic element of a testflow.

A test suite is described by 3 setup elements:

- One or multiple **specification files**, that configure the *instruments*.
- A single **pattern** or an **operating sequence**, that describes the sequence of executing basic test steps.
- A **test method**, that typically implements retrieval and processing of the test results.



**ADVANTEST**

## Navigating Through Testflow Setups via “F3”

Teamwork requires setup data distributed over a modular hierarchy with many files.

```

CrossConnect.prog
5@ /* testprogram CrossConnect {
6
7     dutboard = crossconnect.common.CrossConnect4Site;
8     // licensing = crossconnect.common.CrossConnect;
9
10    var printToConsole = false;
11
12    // optional auxiliary testflow to read and write test tables
13    testflow PreBind {
14        flow = crossconnect.common.PreBind;
15    }
16
17    // optional auxiliary testflow to power up the DUT
18    testflow PowerUp {
19        flow = crossconnect.common.PowerUp;
20    }
21
22    // mandatory main testflow
23    testflow Main {
24        flow = crossconnect.common.Demo;
25    }
}

DigitalPattern.flow
8
9
10    // testflow input parameters, types are auto-recognized
11    in subflowDebugMode = 1;
12    /** enable 'releaseTester' for background processing */
13    in backgroundProcessing = true;
14
15    setup {
16        /* Digital functional pattern executed with a simple test method */
17        suite SimpleFunctional calls com.advantest.itee.tml.actml.FunctionalTest {
18            measurement specification;
19            setupRe(crossconnect.digitalPattern.mainSpecs.FunctionalComplete);
20            measurement specification;
21            setupRe(crossconnect.digitalPattern.patterns.Functional);
22            signals = "ackUpdate";
23        }
24    }

CrossConnect.prog
1@ /**
2 * Common testflow referencing all other testflows.
3 *
4 * @see "Testflow file reference in TDC (Topic 244398)"
5 */
6 flow Demo {
7     in debugMode=2;
8
9     setup {
10         flow digitalPatternFlow calls crossconnect.digitalPattern.DigitalPattern {
11             subflowDebugMode = debugMode,
12         }
13         flow fastDigitalPatternFlow calls crossconnect.digitalPattern.FastDigitalPattern {
14             subflowDebugMode = debugMode;
15             runShmooOverTestsuite = 0;
16     }
17 }

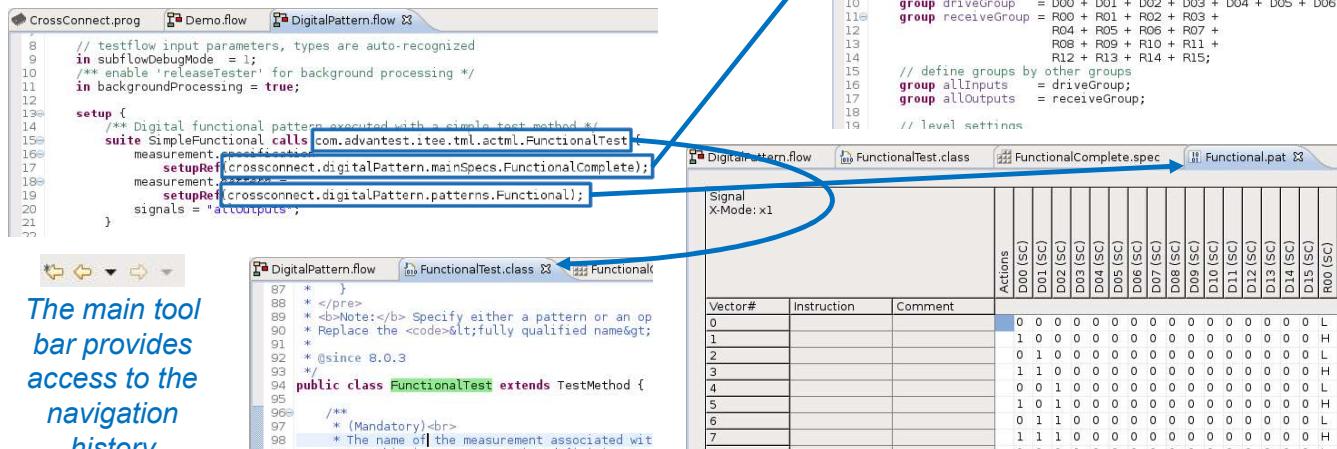
```

For every setup item, it is easy to navigate to its definition: right-click + “Open Declaration” or press “F3” after selecting it.

**ADVANTEST**

# Navigating Through Test Suite Setups via “F3”

Usage of “F3” (“Open Declaration”) works across all setup files including patterns, operating sequences, specification files and test method files.



SmarTEST 8

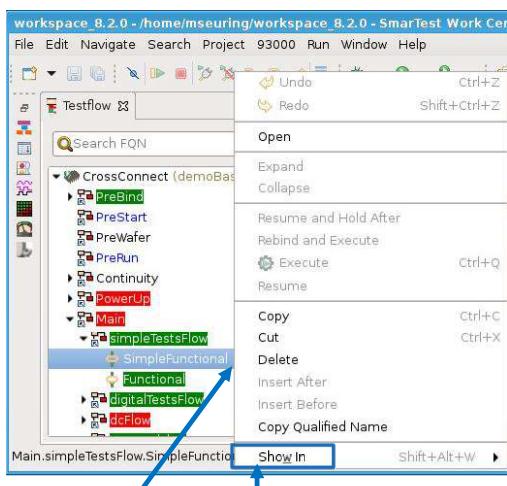
January 2020

All Rights Reserved - ADVANTEST CORPORATION

ADVANTEST

SmarTest Test Program Execution and Debug - 7

## Testflow View – Overview



Use “Show In” + “Parameters” to view and edit testflow and test suite parameters in tables

- The graphical testflow view shows the full execution hierarchy of a test program. Testflows and test suites are color coded according to the results of the last run.
- The testflow view allows to start execution or debug of complete test programs or just testflows.
- While in debug mode, it allows to execute single test suites or testflows in an arbitrary order.
- The execution order can be modified: Test suites can be copied, cut, deleted and inserted.

SmarTEST 8

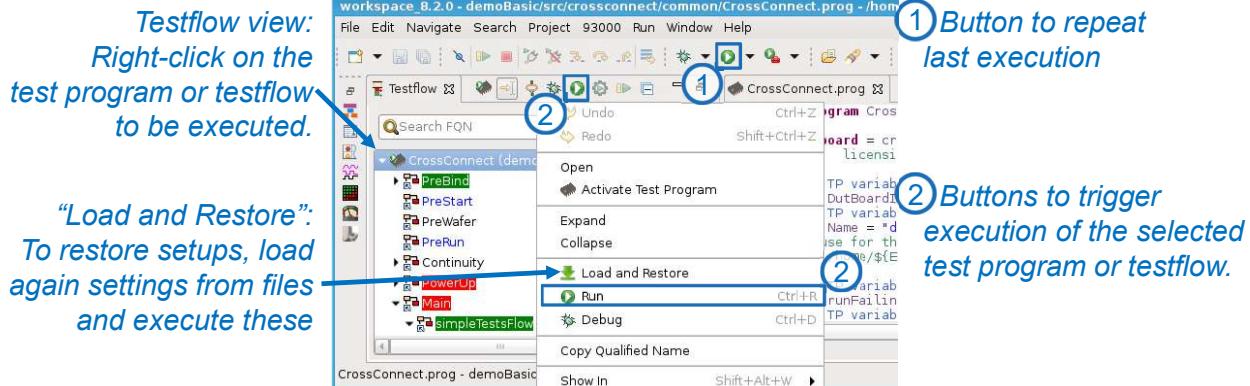
January 2020

All Rights Reserved - ADVANTEST CORPORATION

ADVANTEST

SmarTest Test Program Execution and Debug - 8

# Execution of Test Programs or Testflows



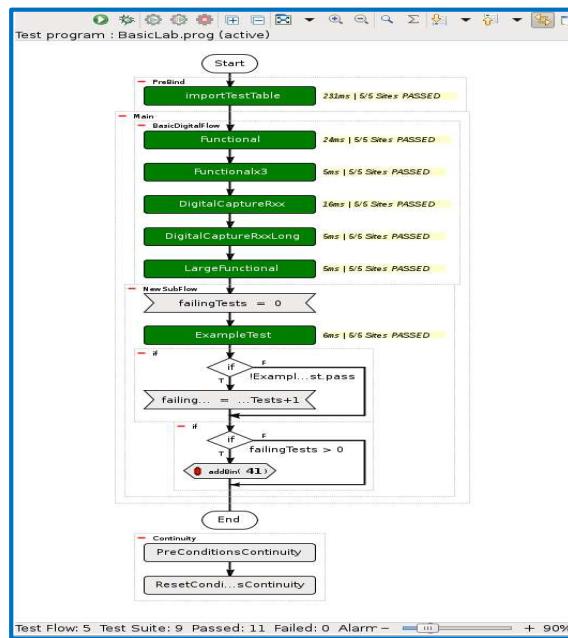
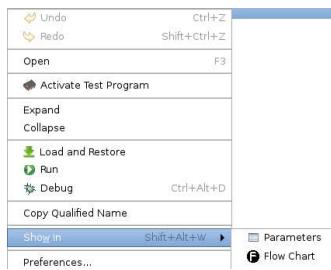
When executing a selected testflow, SmarTest temporarily sets the selected testflow as *main flow*. Then the *main flow* and the auxiliary flows "PreBind", "PreStart", "PreRun", "PowerUp", "PowerDown" and "PreStop" will be executed in the order as listed in the *testflow view*.

Executions can be started as well in the *package explorer* in a similar way:  
Typically it is used for testflow files that are not called by a test program.

# Execution of Test Programs or Testflows from Flow Chart view

The Flow Chart view allows you to debug a test program or testflows without setting manual breakpoints.

Navigate from the Testflow view to the Flow Chart View using **Show In**

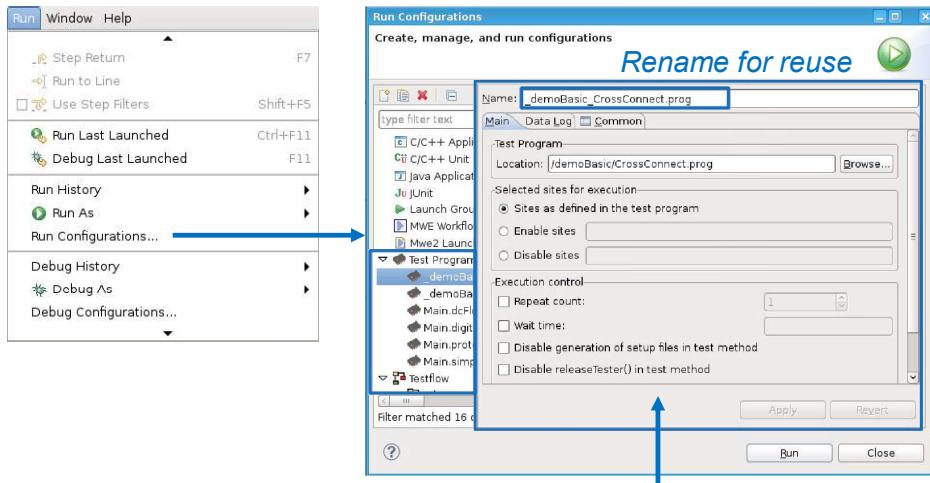


# Run Configurations

A *run configuration* is a collection of user-defined settings to configure the run of an associated test program or testflow.

For reuse, assign a dedicated name to a *run configuration*.

*Run configurations* are automatically saved in the workspace.



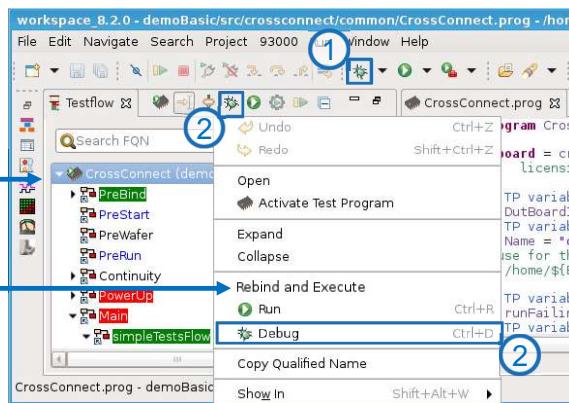
Additional parameters for execution and data logging can be set up here.

# Start Debug of Test Programs or Testflows

Execution in debug mode is started in a similar way as a regular run.

Select testflow or test program for debug and right-click

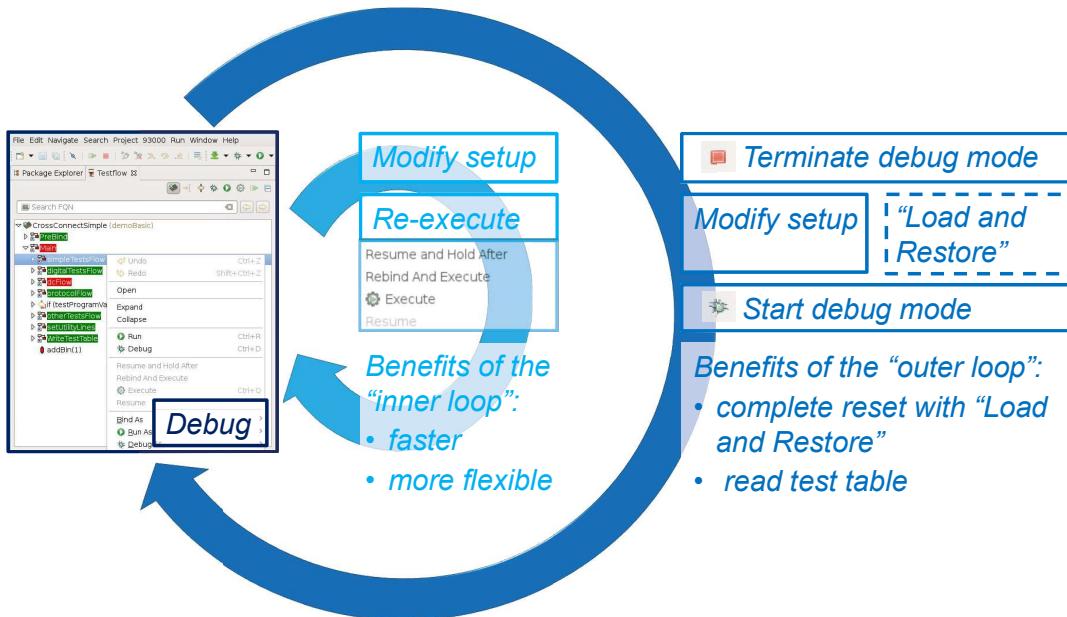
Resets setups to the content of file



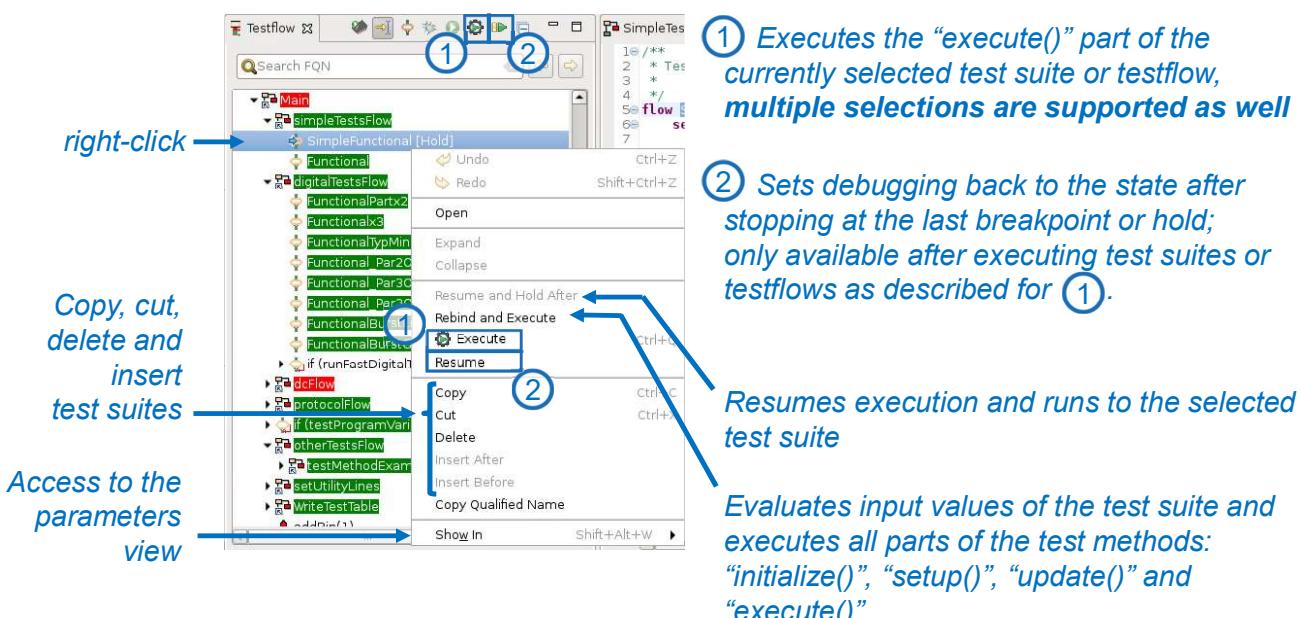
① Trigger to repeat last execution in debug mode. Stops at breakpoints.

② Enters debug mode for the selected test program or testflow, but holds just before running the first test suite. Stops also at breakpoints.

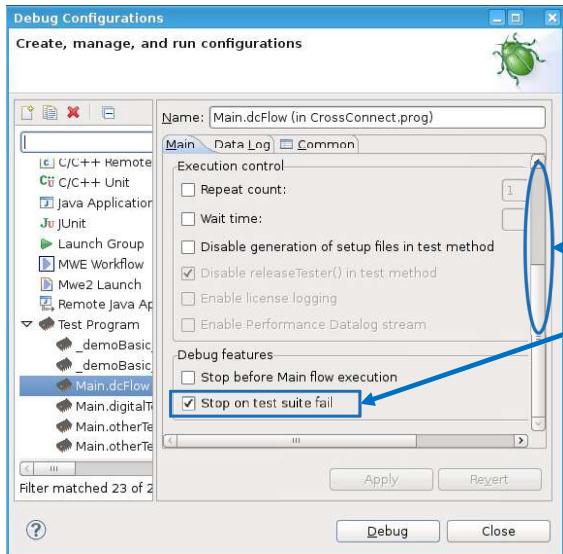
# Debug: Outer Loop versus Inner Loop



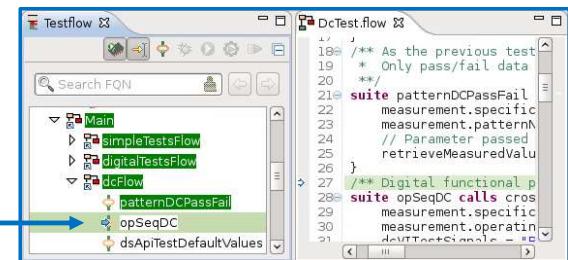
## Inner Loop: Operations on Selected Test Suites



# Debug Configurations



*Debug configurations are very similar to run configurations:  
They are collections of user-defined settings for debugging an associated test program or testflow.*

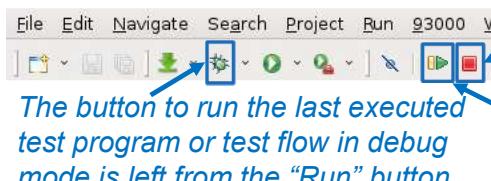
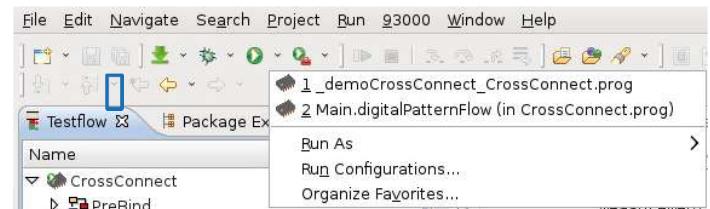


*For debug features scroll down  
If enabled, execution is stopped  
at fails in test suites.*

## Re-run and Terminate Execution



*The “Run” button executes the last executed flow/program, which is shown if the mouse hovers over it.*



*The red square allows to terminate an execution and disconnect.  
The “Resume” button allows to resume execution at a breakpoint or at hold.*

# Execution in a Test Cell

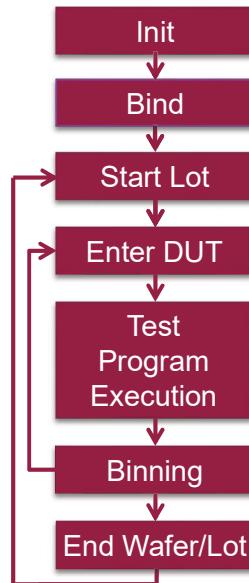
Material flow:

A test cell must interact with the tester (software) to test in an automated way.

Key functionality is the communication with the test cell control software

- to perform the tests of the devices under test and
- to obtain binning information and data logs
- aligned with the test cell.

The diagram gives a simplified overview to show next how the *main flow* and the *auxiliary flows* provided by SmarTest are used to support the material flow.

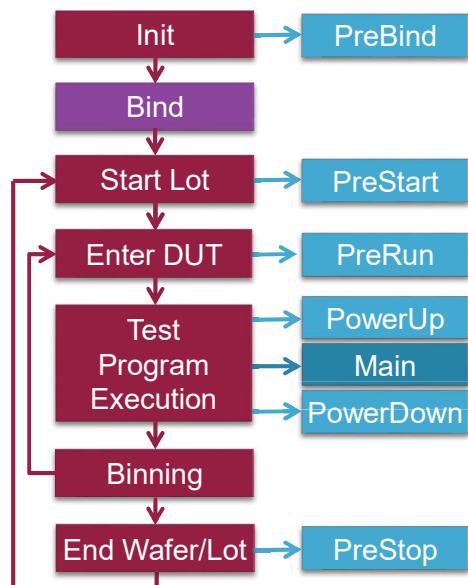


## Auxiliary Flows

Automated test runs through various stages. Some stages might need information or activities which can be set up in the *test program file* using *auxiliary flows*.

These are listed here with usage examples:

- PreBind: Read in test table
- PreStart: Power up DUT board
- PreRun: Set up global variables
- PowerUp: Perform continuity tests and run power up sequence of DUT
- Main (mandatory): Run tests
- PowerDown: Run power down sequence of DUT
- PreStop: Power down DUT board, close datalog file



# Summary - What you should have learned

- Smartest allows to easily navigate through the setup files of test programs.
- The *testflow view* is the tool be used for starting execution and debug:
  - It allows to start execution or debug for the complete test program or just testflows.
  - It shows testflows and test suites in the order as they are executed.
  - It uses color coding to visualize results.
  - It allows to execute single test suites or testflows in an arbitrary order while in debug mode.
  - It can be used to modify the execution order: Test suites can be copied, cut, deleted and inserted.
- SmarTest supports the “outer loop” and the “inner loop” for debugging.
- The main tool bar allows to start, resume and terminate execution and debug.
- *Run configurations* and *debug configurations* allow to change the settings like enabled or disabled sites
- *Auxiliary flows* are used to interact in an automatic test environment.

## Backup

## Table: “Execute” vs. “Rebind and Execute” in the Testflow View

	Testflow: setup block	Testflow: execute block	Test method: initialize(), setup(), update()	Test method: execute()
Testflow selected: <b>Execute</b>				
Test suite selected: <b>Execute</b>				
Testflow selected: <b>Rebind and execute</b>				
Test suite selected: <b>Rebind and execute</b>				

**Resume and hold** runs the “execute” block and “execute()” class method of the next test flows and test methods and stops, after the run of the selected testflow/test suite has been completed.



## Steps of Test Program Execution

SmarTest 8.2.5 Training

January 2020

# Learning Objective

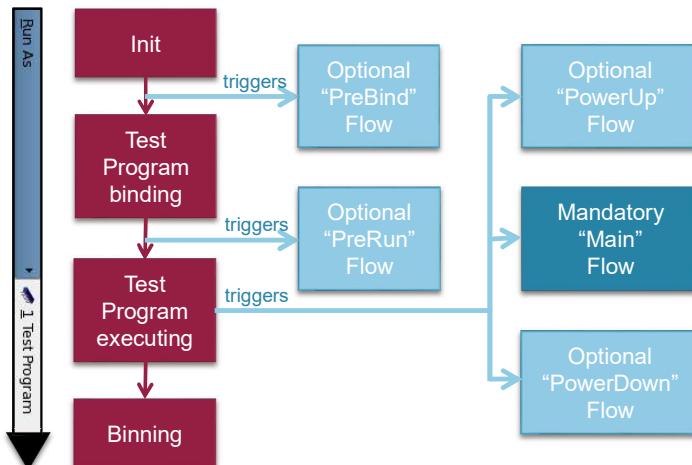
- Learn about the various steps of executing a test program

# Agenda

- Detailed description of the steps of test program execution.

# Sequence of Steps: Execution of Auxiliary Flows

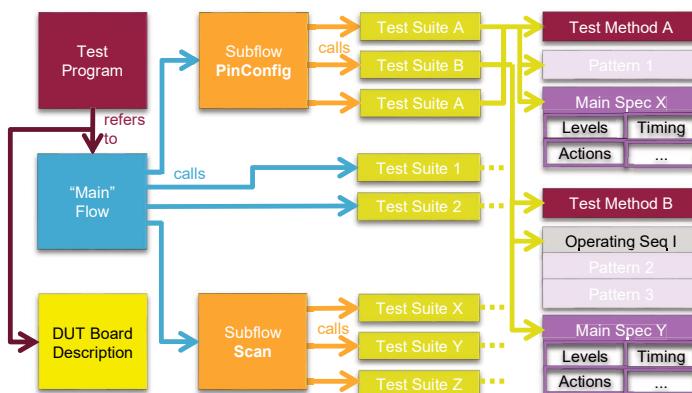
A test program is executed in several steps which will be explained in the following.  
In particular, beside running the *main flow*, it triggers the execution of the *auxiliary flows*.  
For simplicity, the *auxiliary flows* “PreStart”, “PreStop” and “Continuity” are not shown.



# Sequence of Steps for Execution: Overview

When executing a test program, what happens behind the scenes?

Details on how the setup elements affect the execution is explained using the list on the right and using the schematic of the building blocks of a test program setup below.



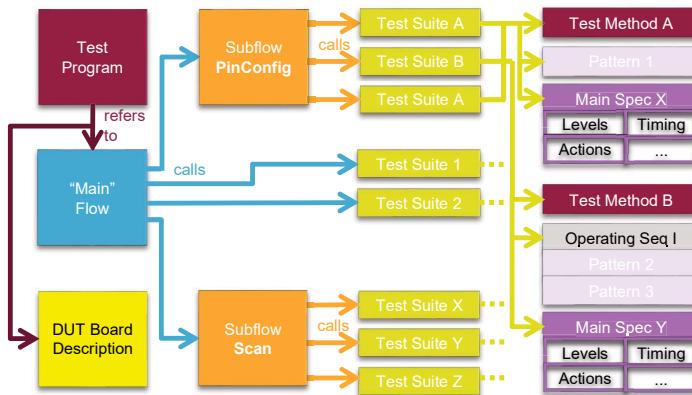
## Execution steps

0. Load
1. Initialization.
2. Execute “PreBind” flow
3. **“setup()” method**
4. Link.
5. Bind.
6. **“update()” method**
7. **“execute()” method**
  - a) “PreRun” flow
  - b) “PowerUp” flow
  - c) “Main” flow
  - d) “PowerDown” flow
8. Binning.

# Execution: Load Test Program

For the first execution, all setup files of the active test program are read and the necessary data structures are created.

After an execution, the generated data structures might be reused for the next execution.



## Execution steps

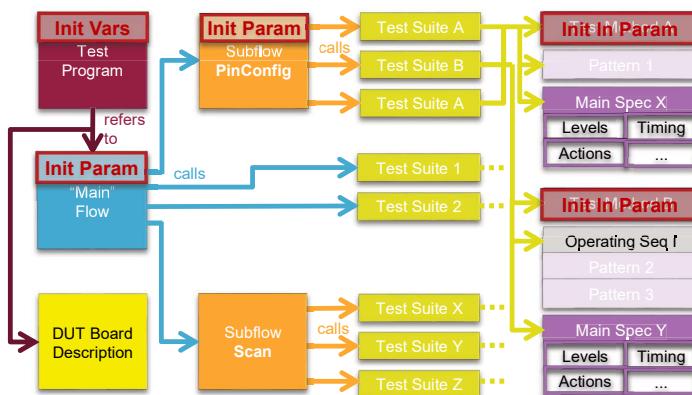
### 0. Load

1. Initialization.
2. Execute "PreBind" flow
3. **"setup()" method**
4. Link.
5. Bind.
6. **"update()" method**
7. **"execute()" method**
  - a) "PreRun" flow
  - b) "PowerUp" flow
  - c) "Main" flow
  - d) "PowerDown" flow
8. Binning.

# Execution: Initialization

Test program variables and testflow parameters are initialized. Input parameters of test methods are set according to the test suite parameters in the "setup" parts of testflow files.

For each defined test suite, the "**initialize()**" method of its test method is executed.

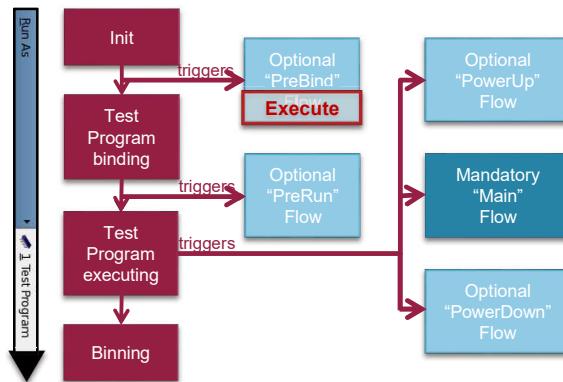


## Execution steps

0. Load
1. **Initialization.**
2. Execute "PreBind" flow
3. **"setup()" method**
4. Link.
5. Bind.
6. **"update()" method**
7. **"execute()" method**
  - a) "PreRun" flow
  - b) "PowerUp" flow
  - c) "Main" flow
  - d) "PowerDown" flow
8. Binning.

# Execution: “PreBind” Flow

This flow is typically used to read *test tables* that are used to set or overwrite parameters of testflows and test suites, to determine test limits, to configure data logging and alarm handling, to define software and hardware bins, to overwrite values of test program variables and more.



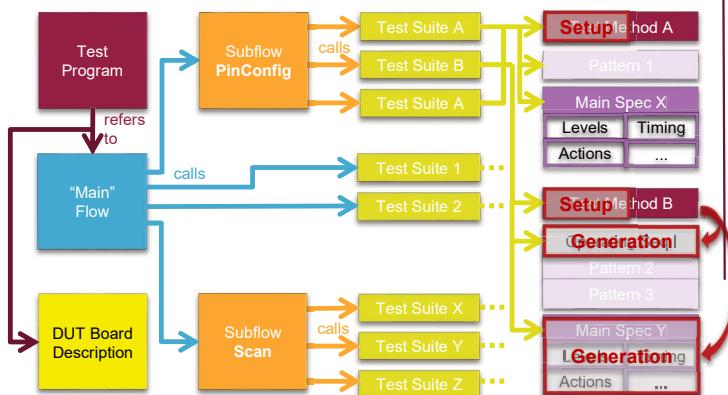
## Execution steps

0. Load
1. Initialization.
2. Execute “PreBind” flow
3. **“setup()” method**
4. Link.
5. Bind.
6. **“update()” method**
7. **“execute()” method**
  - a) “PreRun” flow
  - b) “PowerUp” flow
  - c) “Main” flow
  - d) “PowerDown” flow
8. Binning.

# Execution: “setup()” Method of Test Methods

For each test suite, the “setup()” method of its test method is executed.

In this step additional setup files might be generated from test methods using the *Device Setup API*.

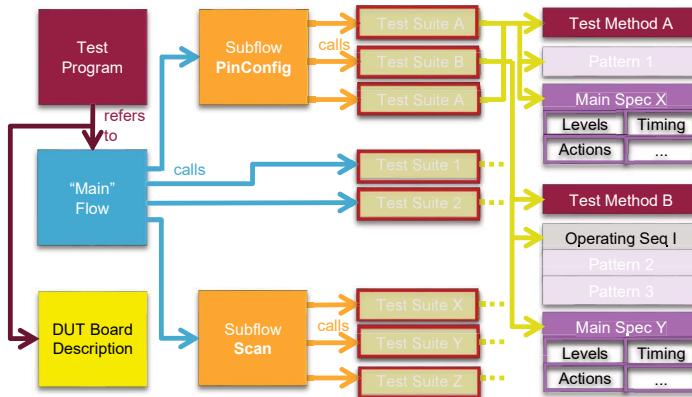


## Execution steps

0. Load
1. Initialization.
2. Execute “PreBind” flow
3. **“setup()” method**
4. Link.
5. Bind.
6. **“update()” method**
7. **“execute()” method**
  - a) “PreRun” flow
  - b) “PowerUp” flow
  - c) “Main” flow
  - d) “PowerDown” flow
8. Binning.

# Execution: Link

Test methods and *measurement* setup objects are linked together. Equations and terms of the setups are evaluated.



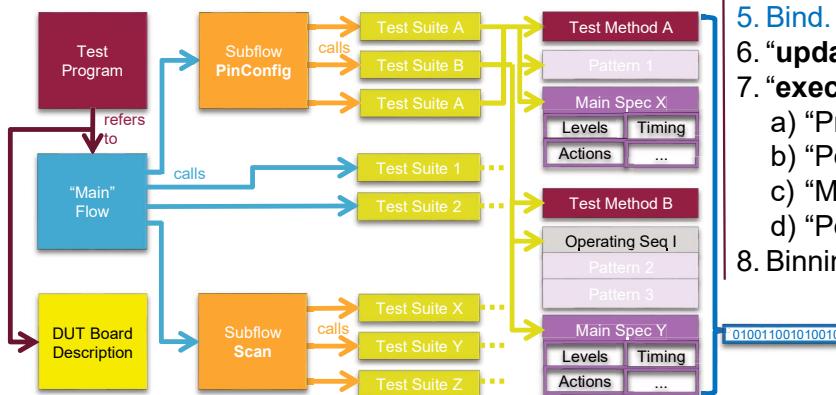
## Execution steps

0. Load
1. Initialization.
2. Execute “PreBind” flow
3. **“setup()” method**
- 4. Link.**
5. Bind.
6. **“update()” method**
7. **“execute()” method**
  - a) “PreRun” flow
  - b) “PowerUp” flow
  - c) “Main” flow
  - d) “PowerDown” flow
8. Binning.

# Execution: Bind and Download

The settings of the *measurements* are mapped to the tester hardware. If it fails, a *bind error* is thrown.

If the setups like patterns, levels, timings, *actions* and *transactions* are new or modified, then these are downloaded to the tester channels.



## Execution steps

0. Load
1. Initialization.
2. Execute “PreBind” flow
3. **“setup()” method**
- 4. Link.**
- 5. Bind.**
6. **“update()” method**
7. **“execute()” method**
  - a) “PreRun” flow
  - b) “PowerUp” flow
  - c) “Main” flow
  - d) “PowerDown” flow
8. Binning.

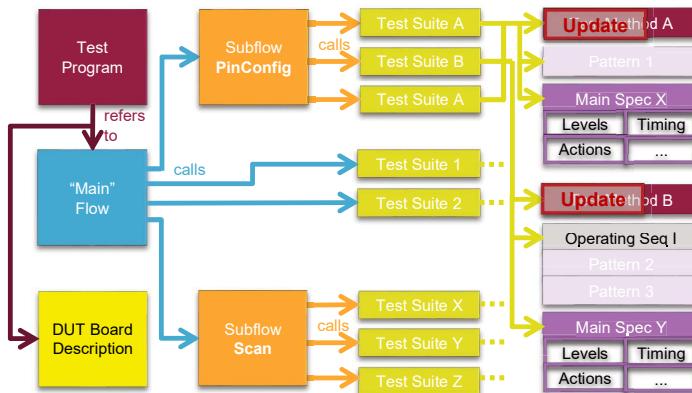


# Execution: “update()” Method of Test Methods

For each defined test suite, the “update()” method of its test method is executed.

In this step typically site specific setup values are set in the *measurements*.

Modified settings trigger additional evaluate and bind steps.



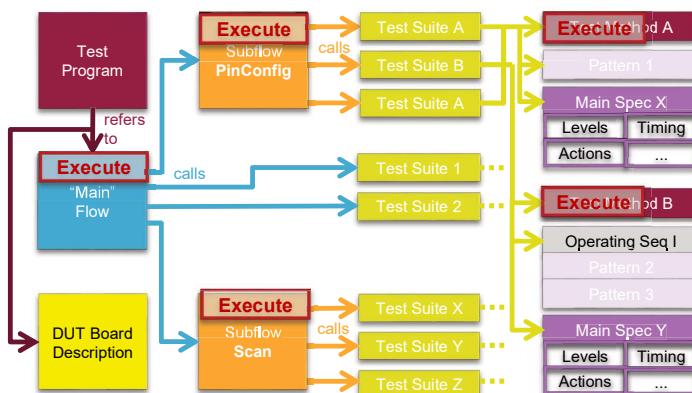
## Execution steps

0. Load
1. Initialization.
2. Execute “PreBind” flow
3. “**setup()**” method
4. Link.
5. Bind.
6. “**update()**” method
7. “**execute()**” method
  - a) “PreRun” flow
  - b) “PowerUp” flow
  - c) “Main” flow
  - d) “PowerDown” flow
8. Binning.

# Execution: “execute()” method of Test Methods

The “execute” part of testflows and the “execute()” method of test methods are executed.

During execution, setup data can be changed which might trigger additional compile and bind steps.



## Execution steps

0. Load
1. Initialization.
2. Execute “PreBind” flow
3. “**setup()**” method
4. Link.
5. Bind.
6. “**update()**” method
7. “**execute()**” method
  - a) “PreRun” flow
  - b) “PowerUp” flow
  - c) “Main” flow
  - d) “PowerDown” flow
8. Binning.

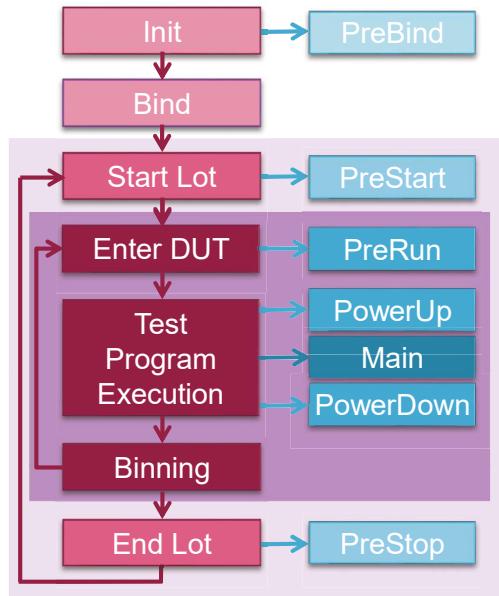
# Execution of Main Flow and Auxiliary Flows

When entering the “execute” step, the test program is bound and prepared to be executed.

Then the complete test program, means *main flow* plus *auxiliary flows*, is executed for all devices in a lot or for all loop iterations.

Exceptions:

- Executed only once per wafer:  
“PreWafer” flow.
- Executed only once per lot:  
“PreStart” and “PreStop” flows.
- Executed only once before this step:  
– “PreBind” flow and  
– in test methods:  
“initialize”, “setup()” and “update()” methods.



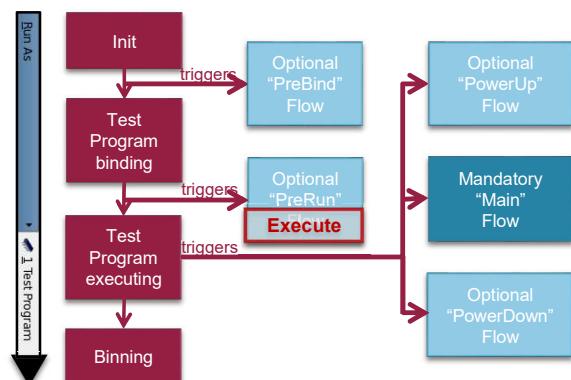
## Execution: “PreRun” Flow

The “execute” part of the optional “PreRun” flow is processed: The “execute()” methods of test methods, that are called by test suites of the “PreRun” flow, are executed.

This flow is typically used to implement adaptive test by statistical sampling of *measurements* or to set up specific data like serial numbers.

### Execution steps

0. Load
1. Initialization.
2. Execute “PreBind” flow
3. **“setup()” method**
4. Link.
5. Bind.
6. **“update()” method**
7. **“execute()” method**
  - a) “PreRun” flow
  - b) “PowerUp” flow
  - c) “Main” flow
  - d) “PowerDown” flow
8. Binning.



# Execution: “PowerUp” Flow

The “execute()” part of the optional “PowerUp” flow is processed:  
It is used

1. to perform optionally continuity and power/short tests;
2. to connect signals, in particular DPS (*dcVI instruments*)  
signals with typical rating through a proper sequence.

## Execution steps

0. Load
1. Initialization.
2. Execute “PreBind” flow
3. **“setup()” method**
4. Link.
5. Bind.
6. **“update()” method**
7. **“execute()” method**
  - a) “PreRun” flow
  - b) **“PowerUp” flow**
  - c) “Main” flow
  - d) “PowerDown” flow
8. Binning



# Execution: “Main” Flow

SmarTest starts with the “execute” part of the testflow that is the *main flow* and then follows the calling hierarchy:

- If a subflow is called then the “execute” part of this testflow will be executed.
- If a test suite is called then the “execute()” method of the corresponding test method will be executed.

## Execution steps

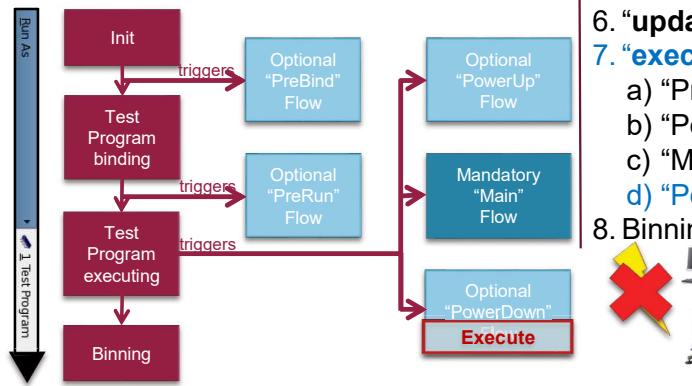
0. Load
1. Initialization.
2. Execute “PreBind” flow
3. **“setup()” method**
4. Link.
5. Bind.
6. **“update()” method**
7. **“execute()” method**
  - a) “PreRun” flow
  - b) **“PowerUp” flow**
  - c) **“Main” flow**
  - d) “PowerDown” flow
8. Binning



# Execution: “PowerDown” Flow

The “**execute()**” method of the “PowerDown” flow and its test methods is executed.

1. Ramp down power supplies and disconnect all *instruments* if special routine is needed.
2. Implement complex binning algorithms.
3. Do other cleanup if necessary.



## Execution steps

0. Load
1. Initialization.
2. Execute “PreBind” flow
3. “**setup()**” method
4. Link.
5. Bind.
6. “**update()**” method
7. “**execute()**” method
  - a) “PreRun” flow
  - b) “PowerUp” flow
  - c) “Main” flow
  - d) “PowerDown” flow
8. Binning

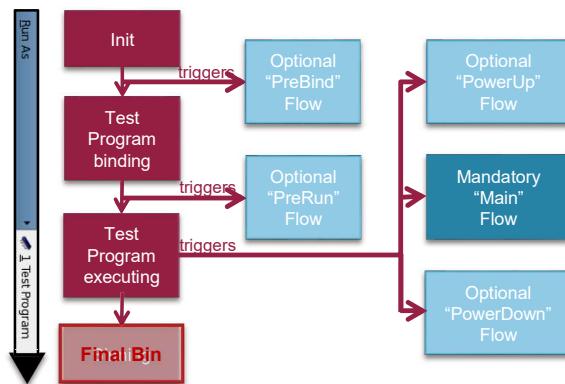


# Execution: Binning

In this step, the final bin has been evaluated:

- A test method in the “PowerDown” flow sets it, or
- it is determined by the built-in default algorithm.

From the *final bin*, the hardware bin is calculated and reported to the material handling hardware.



## Execution steps

0. Load
1. Initialization.
2. Execute “PreBind” flow
3. “**setup()**” method
4. Link.
5. Bind.
6. “**update()**” method
7. “**execute()**” method
  - a) “PreRun” flow
  - b) “PowerUp” flow
  - c) “Main” flow
  - d) “PowerDown” flow
8. Binning.

# Multiple Executions of the “Main” Flow

Steps 7 and 8 might be executed repeatedly.

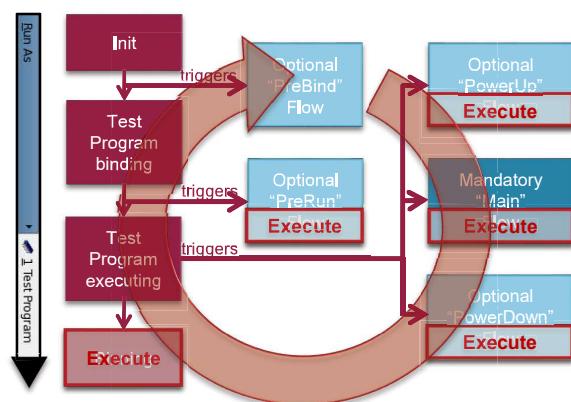
- For engineering typically when the *Run Configurations* are set to repeat the execution multiple times.
- In production for example if devices of a lot are tested through the *Test Cell Control Tool (TCCT)*.

## Execution steps

0. Load
1. Initialization.
2. Execute “PreBind” flow
3. “**setup()**” method
4. Link.
5. Bind.
6. “**update()**” method
7. “**execute()**” method
  - a) “PreRun” flow
  - b) “PowerUp” flow
  - c) “Main” flow
  - d) “PowerDown” flow
8. Binning.



**ADVANTEST**



## Summary - What you should have learned

The sequence of execution steps:

0. Load
1. Initialization.
2. Execute “PreBind” flow
3. “**setup()**” method
4. Link.
5. Bind.
6. “**update()**” method
7. “**execute()**” method
  - a) “PreRun” flow
  - b) “PowerUp” flow
  - c) “Main” flow
  - d) “PowerDown” flow
8. Binning.

In particular, the following is important for this sequence:

- Applying values of input parameters and execution of “initialize()” happens before reading *test tables*. After that, the “**setup()**” methods are executed and then link and bind is performed.
- After executing the “**update()**” methods everything is prepared to execute physical testst.
- “**initialize()**”, “**setup**” and “**update()**” are only executed once.
- After executing the “**update()**” methods once, “PreRun”, “PowerUp”, “Main” and “PowerDown” flows and binning are executed per device or per loop



# Protocol-Aware Software: Introduction

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

Protocol-Aware Software: Introduction - 1

## Agenda

- Protocol Definition
- Protocol Examples
- Solution Motivation



January 2020

All Rights Reserved - ADVANTEST CORPORATION

**ADVANTEST**

Protocol-Aware Software: Introduction - 2

# Basics: Protocol Definition

Communicating devices use well-defined formats for exchanging various messages, i.e. protocols. A protocol “defines the rules syntax, semantics and synchronization of communication and possible error recovery methods” ([en.wikipedia.org, “communications protocols”, February 2019](https://en.wikipedia.org/wiki/Communication_protocol)).

These are typically defined as standards by associations such as IEEE.

Protocols are based on a physical specification:

- Pin definition
- Physical connector/cables
- Timing and synchronization



Protocols define the logical operation:

- Control (master, slave or peer)
- Commands and responses
- Frame, address and data definition



# Basics: Protocol Example

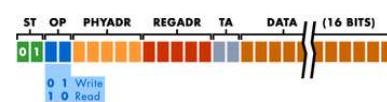
The protocol “Management Data Input/Output” is based on a serial bus. It was originally defined in Clause 22 of the standard IEEE RFC802.3.



## Physical specification:

- Two Signals: clock (MDC) and data input/output (MDIO).
- Frequency of up to 2.5MHz + additional timing specifications.
- Levels: Clause 45 added support low voltage devices down to 1.2V.
- One master and up to 32 slaves.

SLOT	Bits	Description
ST	2	Start of Frame (01 for Clause 22)
OP	2	OP Code (01 Read – 10 Write)
PHYADR	5	PHY Address
REGADR	5	Register Address
TA	2	Turnaround time to change bus ownership from master to slave (if required)
DATA	16	Data <ul style="list-style-type: none"><li>- Driven by the master during Write transaction</li><li>- Driven by the slave during Read transaction</li></ul>



Protocol transactions:

- Write(phy, addr, data);
- Read(phy, addr, data);

# Test Generation: Traditional vs. PA

Generation of stimulus data:

## Traditional:

Setup design simulation to:

- bring device in a proper state;
- monitor the simulated results and
- write the stimulus data

## Protocol-Aware:

High level protocol transactions are

- manually generated or
- copied over from system test or
- derived from high level simulation test bench.

Conversion of stimulus data:

## Traditional:

- Generate cyclized stimulus
- Generate expected response
- Convert timing
- Convert stimulus to vectors

## Protocol-Aware:

- Automatically done by SmarTest before the first run.

For each protocol test this process is needed.

Potentially, repeat again the process when debugging these tests.

## Traditional Approach: Issues

- Time consuming:  
Setup and simulation may require many hours for each setup.
- Error prone:  
Multiple steps within the process and any error/type will cause incorrect results that are challenging to debug.
- Hard to maintain and debug - flat vectors do not show the payload data:
  - Payload data is hardly distinguishable from other protocol data.  
But this is often needed, for example if during debug the protocol communication must be analyzed or modified.
  - If IP cores are reused, it is difficult to segment and leverage:  
Although the new device contains old IP, the generation of the data for protocol based tests must start again from the beginning.

# PA: Goal - Bench Utilization Model

The definition of the protocol interface is provided as a library.

To communicate via the protocol, predefined setups exists which incorporate:

- Timing settings.
- Level settings.
- Protocol definition.

Engineers must only understand the messaging component of the protocol, i.e. write, read, idle.

Therefore, engineers only provide the transaction information to setup the protocol based test, i.e. the sequences of protocol transactions, addresses and data.

## Summary - What you should have learned

- The protocol-aware software does not need any specific hardware.
- It can make the generation of test setups for protocols more efficient and less error prone.
- It allows to work with protocol test setups using the higher abstraction level of protocol transactions:
  - Easier and faster to debug.
  - Facilitates exchanging test setups between tester and bench environment.
  - Easier to understand and maintain.



# Test Methods: Setup Generation

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION



Test Methods: Setup Generation - 1

## Learning Objective

- Understand the benefits of programmatic generation of test setups
- Know how to implement test methods that programmatically
  - generate a *specification file* with *instrument* and *action* setups;
  - generate an *operating sequence file*.
- Know how to debug a *measurement* based on the generated setup files.



January 2020

All Rights Reserved - ADVANTEST CORPORATION



Test Methods: Setup Generation - 2

# Agenda

- Motivation to programmatically generate test setups
- Introduction to the Device Setup API
- Example of an implementation
- Generated setup files of the example
- Debug

## Static Files for Test Setups

```
1 spec SpecFileExample {
2     // group definitions
3     group gDrive    = D00 + D01;
4     group gReceive  = R00 + R01;
5
6     // variables
7     var Time per    = 10e-06 s;
8     var Voltage vcc = 1.4 V;
9
10    // digital instrument setup: timing
11    setup digInOut gDrive+gReceive {
12        wavetable wvt1 { // wavetable
13            xModes = 1 {
14                0: d1:0;
15                1: d1:1;
16                L: d1:Z r1:L;
17                .. d1:7 r1:1;
18            }
19
20        sequence ParallelGroupExample {
21            parallel parGrp1 {
22                sequential seq1_1 {
23                    patternCall PatternA0 PatternLane0;
24                }
25                sequential seq1_2 {
26                    patternCall PatternA1 PatternLane1;
27                }
28            }
29        }
30    }
31
32    Source
```

*Specification files, operating sequence files and other static setup files are based on the SmarTest Setup Format (SSF).*

### Benefits:

- ✓ intuitive settings
- ✓ content assist available
- ✓ concise
- ✓ clear

### Problems of static files:

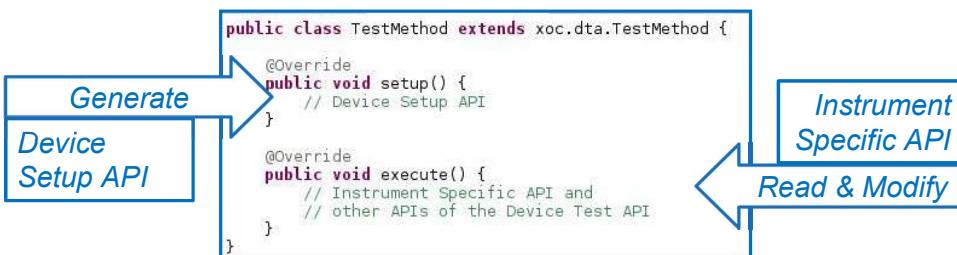
- manual generation
- limited parameterization

### Solution:

Programmatic generation of setup files with the *Device Setup API (DS-API)*.

# Differences between DS-API and other API's

Device Setup API	Instrument Specific API
Used for creating a test setup.	Used to update/modify settings of the <b>current measurement</b> .
Invoked by the SmarTest 8 framework prior to the <i>bind</i> step – when calling “setup()” in the test method.	Invoked by the SmarTest 8 framework after the <i>bind</i> step – when calling “update()” and “execute()” in the test method.
Do not have any “getter” methods to read settings of the <i>instruments</i> .	Provides “getter” methods to read settings of the <i>instruments</i> .



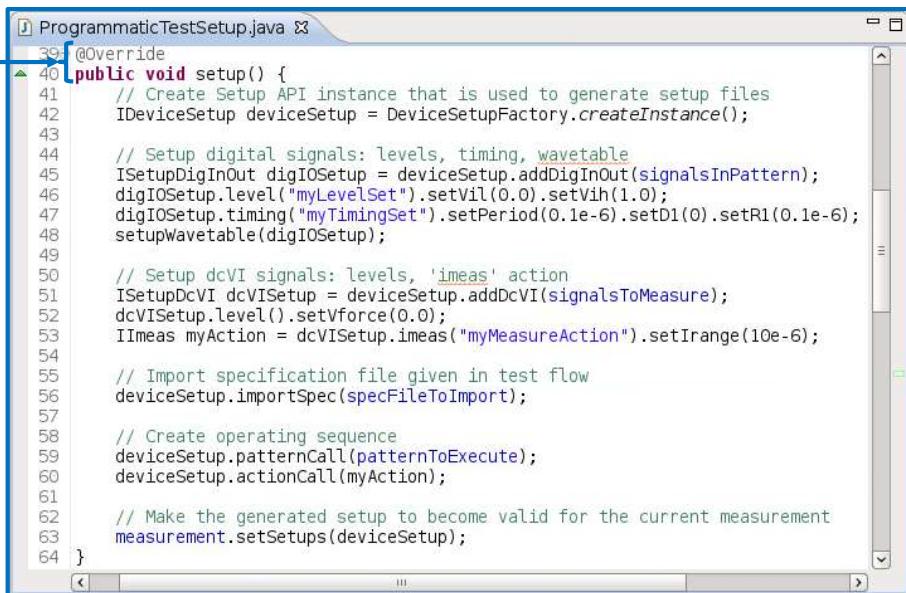
## Device Setup API Elements

The *Device Setup API* allows to create setup elements of *specification files*, of *operating sequence files* and of other file types:

- Setup of any *instrument* type with properties, *actions* and other settings.
- *Operating sequences* with (nested) *parallel* and *sequential groups*.
- Specification variables and equations.
- Signal aliases and signal groups.
- Waveforms for use within instrument specifications.
- Patterns.
- Elements of protocol-aware test setups.

# Programmatic Setup Generation: Typical Steps

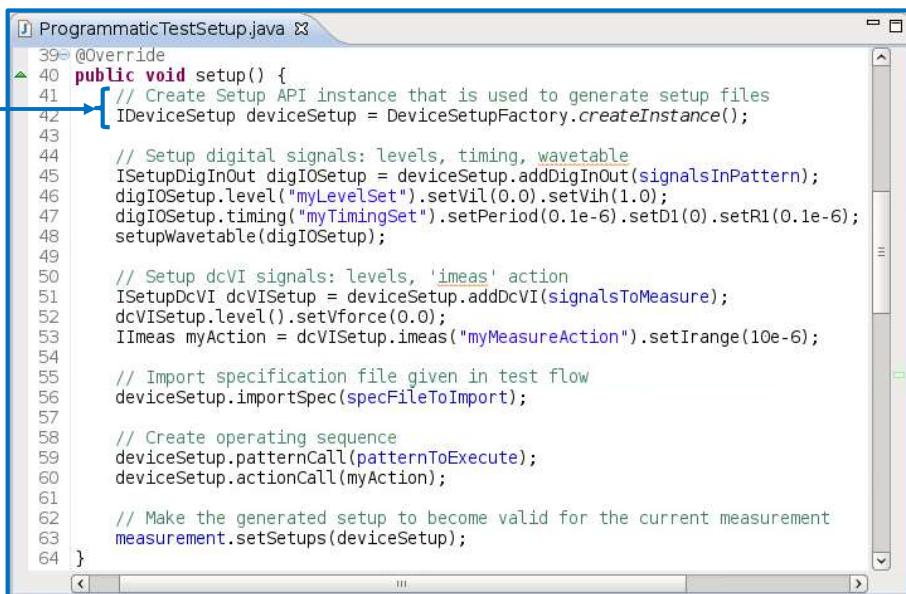
The Device Setup API can only be used in the class method "setup()" of a test method. It must be added if it is not yet implemented in the test method.



```
1 ProgrammaticTestSetup.java
2
3 39- @Override
4 40 public void setup() {
5     // Create Setup API instance that is used to generate setup files
6     IDeviceSetup deviceSetup = DeviceSetupFactory.createInstance();
7
8     // Setup digital signals: levels, timing, wavetable
9     ISetupDigInOut digIOSetup = deviceSetup.addDigInOut(signalsInPattern);
10    digIOSetup.level("myLevelSet").setVil(0.0).setVih(1.0);
11    digIOSetup.timing("myTimingSet").setPeriod(0.1e-6).setD1(0).setR1(0.1e-6);
12    setupWavetable(digIOSetup);
13
14    // Setup dcVI signals: levels, 'imeas' action
15    ISetupDcVI dcVISetup = deviceSetup.addDcVI(signalsToMeasure);
16    dcVISetup.level().setVforce(0.0);
17    II meas myAction = dcVISetup.imeas("myMeasureAction").setIrange(10e-6);
18
19    // Import specification file given in test flow
20    deviceSetup.importSpec(specFileToImport);
21
22    // Create operating sequence
23    deviceSetup.patternCall(patternToExecute);
24    deviceSetup.actionCall(myAction);
25
26    // Make the generated setup to become valid for the current measurement
27    measurement.setSetups(deviceSetup);
28
29 }
```

## Programmatic Setup Generation: Step 1 of 5

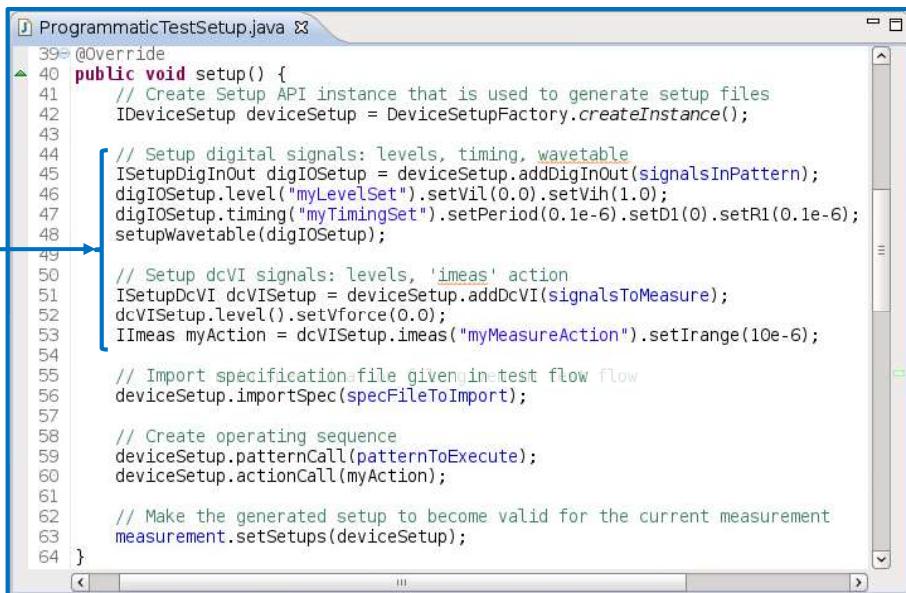
The instantiation of the "IDeviceSetup" object is required to programmatically create setup elements.



```
1 ProgrammaticTestSetup.java
2
3 39- @Override
4 40 public void setup() {
5     // Create Setup API instance that is used to generate setup files
6     IDeviceSetup deviceSetup = DeviceSetupFactory.createInstance();
7
8     // Setup digital signals: levels, timing, wavetable
9     ISetupDigInOut digIOSetup = deviceSetup.addDigInOut(signalsInPattern);
10    digIOSetup.level("myLevelSet").setVil(0.0).setVih(1.0);
11    digIOSetup.timing("myTimingSet").setPeriod(0.1e-6).setD1(0).setR1(0.1e-6);
12    setupWavetable(digIOSetup);
13
14    // Setup dcVI signals: levels, 'imeas' action
15    ISetupDcVI dcVISetup = deviceSetup.addDcVI(signalsToMeasure);
16    dcVISetup.level().setVforce(0.0);
17    II meas myAction = dcVISetup.imeas("myMeasureAction").setIrange(10e-6);
18
19    // Import specification file given in test flow
20    deviceSetup.importSpec(specFileToImport);
21
22    // Create operating sequence
23    deviceSetup.patternCall(patternToExecute);
24    deviceSetup.actionCall(myAction);
25
26    // Make the generated setup to become valid for the current measurement
27    measurement.setSetups(deviceSetup);
28
29 }
```

## Programmatic Setup Generation: Step 2 of 5

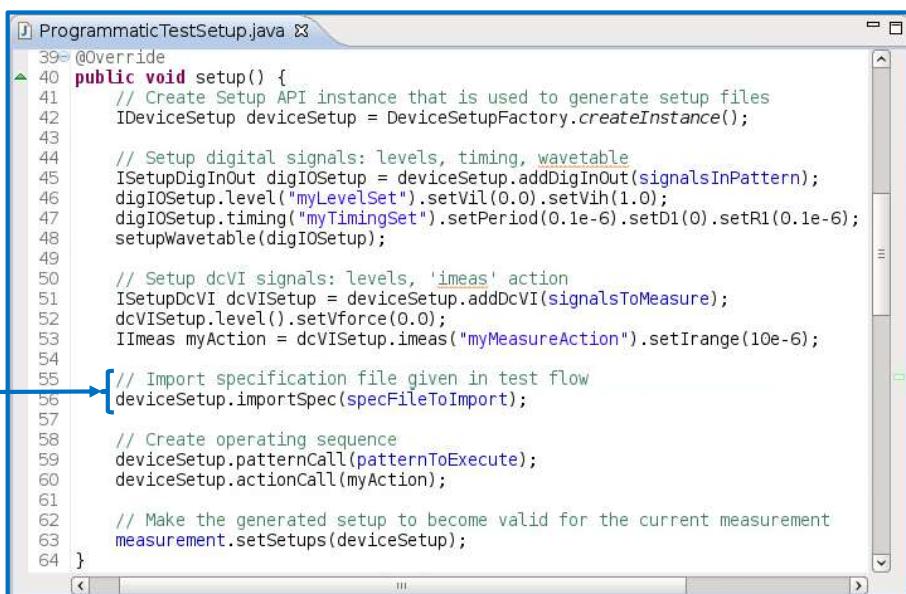
*Creation of a specification file with instrument and action setups and more.*



```
39  @Override
40  public void setup() {
41      // Create Setup API instance that is used to generate setup files
42      IDeviceSetup deviceSetup = DeviceSetupFactory.createInstance();
43
44      // Setup digital signals: levels, timing, wavetable
45      ISetupDigInOut digIOSetup = deviceSetup.addDigInOut(signalsInPattern);
46      digIOSetup.level("myLevelSet").setVil(0.0).setVih(1.0);
47      digIOSetup.timing("myTimingSet").setPeriod(0.1e-6).setD1(0).setR1(0.1e-6);
48      setupWavetable(digIOSetup);
49
50      // Setup dcVI signals: levels, 'imeas' action
51      ISetupDcVI dcVISetup = deviceSetup.addDcVI(signalsToMeasure);
52      dcVISetup.level().setVforce(0.0);
53      IImeas myAction = dcVISetup.imeas("myMeasureAction").setIrange(10e-6);
54
55      // Import specification file given in test flow Flow
56      deviceSetup.importSpec(specFileToImport);
57
58      // Create operating sequence
59      deviceSetup.patternCall(patternToExecute);
60      deviceSetup.actionCall(myAction);
61
62      // Make the generated setup to become valid for the current measurement
63      measurement.setSetups(deviceSetup);
64 }
```

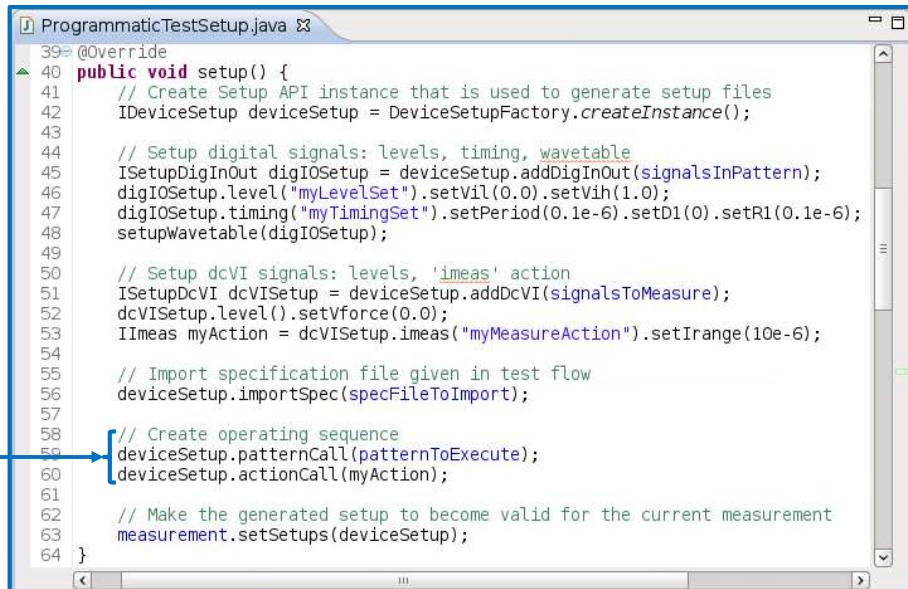
## Programmatic Setup Generation: Step 3 of 5

*Adding an import of an existing specification file.*



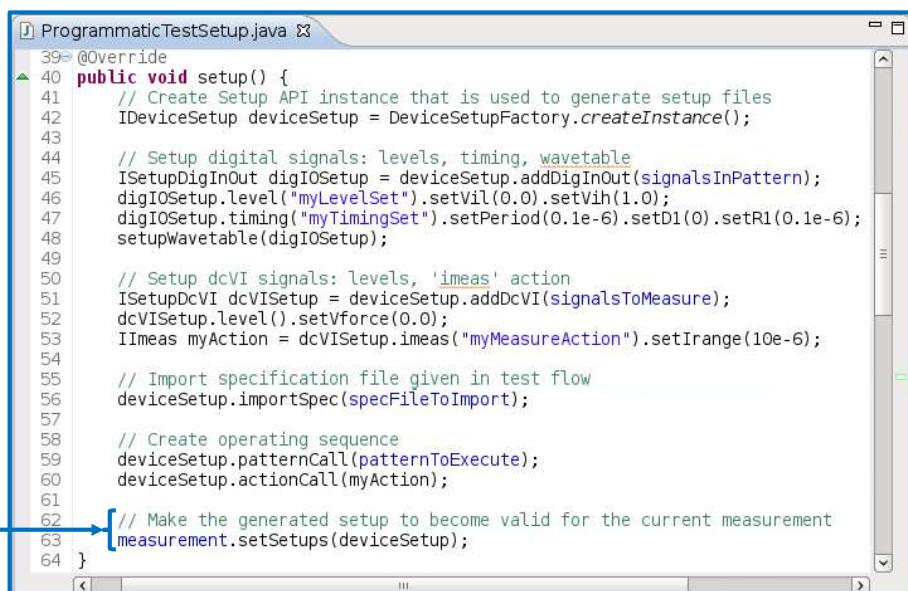
```
39  @Override
40  public void setup() {
41      // Create Setup API instance that is used to generate setup files
42      IDeviceSetup deviceSetup = DeviceSetupFactory.createInstance();
43
44      // Setup digital signals: levels, timing, wavetable
45      ISetupDigInOut digIOSetup = deviceSetup.addDigInOut(signalsInPattern);
46      digIOSetup.level("myLevelSet").setVil(0.0).setVih(1.0);
47      digIOSetup.timing("myTimingSet").setPeriod(0.1e-6).setD1(0).setR1(0.1e-6);
48      setupWavetable(digIOSetup);
49
50      // Setup dcVI signals: levels, 'imeas' action
51      ISetupDcVI dcVISetup = deviceSetup.addDcVI(signalsToMeasure);
52      dcVISetup.level().setVforce(0.0);
53      IImeas myAction = dcVISetup.imeas("myMeasureAction").setIrange(10e-6);
54
55      // Import specification file given in test flow
56      deviceSetup.importSpec(specFileToImport);
57
58      // Create operating sequence
59      deviceSetup.patternCall(patternToExecute);
60      deviceSetup.actionCall(myAction);
61
62      // Make the generated setup to become valid for the current measurement
63      measurement.setSetups(deviceSetup);
64 }
```

## Programmatic Setup Generation: Step 4 of 5



```
39  @Override
40  public void setup() {
41      // Create Setup API instance that is used to generate setup files
42      IDeviceSetup deviceSetup = DeviceSetupFactory.createInstance();
43
44      // Setup digital signals: levels, timing, wavetable
45      ISetupDigInOut digIOSetup = deviceSetup.addDigInOut(signalsInPattern);
46      digIOSetup.level("myLevelSet").setVil(0.0).setVih(1.0);
47      digIOSetup.timing("myTimingSet").setPeriod(0.1e-6).setD1(0).setR1(0.1e-6);
48      setupWavetable(digIOSetup);
49
50      // Setup dcVI signals: levels, 'imeas' action
51      ISetupDcVI dcVISetup = deviceSetup.addDcVI(signalsToMeasure);
52      dcVISetup.level().setVforce(0.0);
53      IImeas myAction = dcVISetup.imeas("myMeasureAction").setIrange(10e-6);
54
55      // Import specification file given in test flow
56      deviceSetup.importSpec(specFileToImport);
57
58      // Create operating sequence
59      deviceSetup.patternCall(patternToExecute);
60      deviceSetup.actionCall(myAction);
61
62      // Make the generated setup to become valid for the current measurement
63      measurement.setSetups(deviceSetup);
64 }
```

## Programmatic Setup Generation: Final Step 5



```
39  @Override
40  public void setup() {
41      // Create Setup API instance that is used to generate setup files
42      IDeviceSetup deviceSetup = DeviceSetupFactory.createInstance();
43
44      // Setup digital signals: levels, timing, wavetable
45      ISetupDigInOut digIOSetup = deviceSetup.addDigInOut(signalsInPattern);
46      digIOSetup.level("myLevelSet").setVil(0.0).setVih(1.0);
47      digIOSetup.timing("myTimingSet").setPeriod(0.1e-6).setD1(0).setR1(0.1e-6);
48      setupWavetable(digIOSetup);
49
50      // Setup dcVI signals: levels, 'imeas' action
51      ISetupDcVI dcVISetup = deviceSetup.addDcVI(signalsToMeasure);
52      dcVISetup.level().setVforce(0.0);
53      IImeas myAction = dcVISetup.imeas("myMeasureAction").setIrange(10e-6);
54
55      // Import specification file given in test flow
56      deviceSetup.importSpec(specFileToImport);
57
58      // Create operating sequence
59      deviceSetup.patternCall(patternToExecute);
60      deviceSetup.actionCall(myAction);
61
62      // Make the generated setup to become valid for the current measurement
63      measurement.setSetups(deviceSetup);
64 }
```

Association of the object, that creates the setup elements, with the measurement object of the test method.

# Example continued: Setup in a Testflow

Settings of the *specification file* and the *operating sequence file* in the setup of the test suite are overwritten, if files are programmatically generated and then associated with the *measurement object*.

Example of a test flow that defines test suites using the previously shown test method:

Values are explicitly assigned to the input parameters of the example test method, which uses these values in the setup generation.

The test method can cover different scenarios if the values of these parameters are adapted accordingly.

If no values of the input parameters are set, then the default values given by the definition of the parameters are valid.

## Details: Wavetable Generation

In line 48 of the example of a programmatic test setup generation a function is called

```
48     setupWavetable(digIOSetup);
```

to generate a wavetable with the dedicated method "setupWavetable()" of the test method class:

More than just one drive or receive action can be defined for a state character.

Example: Two drive actions for a state character of a clock pulse.

```
newWvt.addStateCharDescription('c',
    ISetupWavetable.DriveAction.F10,
    ISetupWavetable.DriveAction.F00,
    ISetupWavetable.ReceiveAction.X);
```

# Generated Files: Operating Sequence File

The programmatically generated *operating sequence file* is typically named “OpSeq\_1”. All generated files are stored in a subfolder “*dsa\_gen*” of the source folder.

```
25 sequence OpSeq_1 uses dsa_gen.Main.DcFlow.dsApiTestSetValues.Spec_1
26 {
27     sequential _seq_1
28     {
29         patternCall crossconnect.operatingSequence.patterns.Functional00_03x1;
30         actionCall myMeasureAction;
31     }
32 }
```

The actual content of the generated file starts at line 25, while the previous lines are comments to inform users about:

- The test method that generated the file.
- The testflow and test suite that calls this test method.

Reference to the specification file that was generated as well.

Note, that the generated files are stored in subfolders of “*dsa\_gen*” according to the calling hierarchy of the corresponding test suite.

# Generated Files: Specification File

The path of the *specification file* is the same as for the *operating sequence file*.

```
import crossconnect.common.SignalGroups;
spec Spec_1
{
    set myLevelSet;
    set myTimingSet;
    action myMeasureAction;

    setup digInOut gD00_D03 + gR00_R03
    {
        set level myLevelSet
        {
            vil = 0 V;
            vih = 1 V;
        }

        set timing myTimingSet
        {
            period = 100.000000 ns;
            d1 = 0 s;
            r1 = 100.000000 ns;
        }
    }
}

OpSeq_1.seq
wavetable myWavetable
{
    xModes = 1
    {
        0: d1:F00 r1:X;
        1: d1:F10 r1:X;
        Z: d1:FNZ r1:X;
        L: d1:FNZ r1:L;
        H: d1:FNZ r1:H;
    }
}
setup dcVI D04
{
    level.vforce = 0 V;
    action im meas myMeasureAction
    {
        irange = 10.00000 uA;
    }
}
```

# Reducing the Number of Generated Setup Files

Practical experiences using the *Device Setup API* have shown that test suites often generate many setup files that have the same content for the test setup.

For large test programs this resulted in large delays when binding so many files.

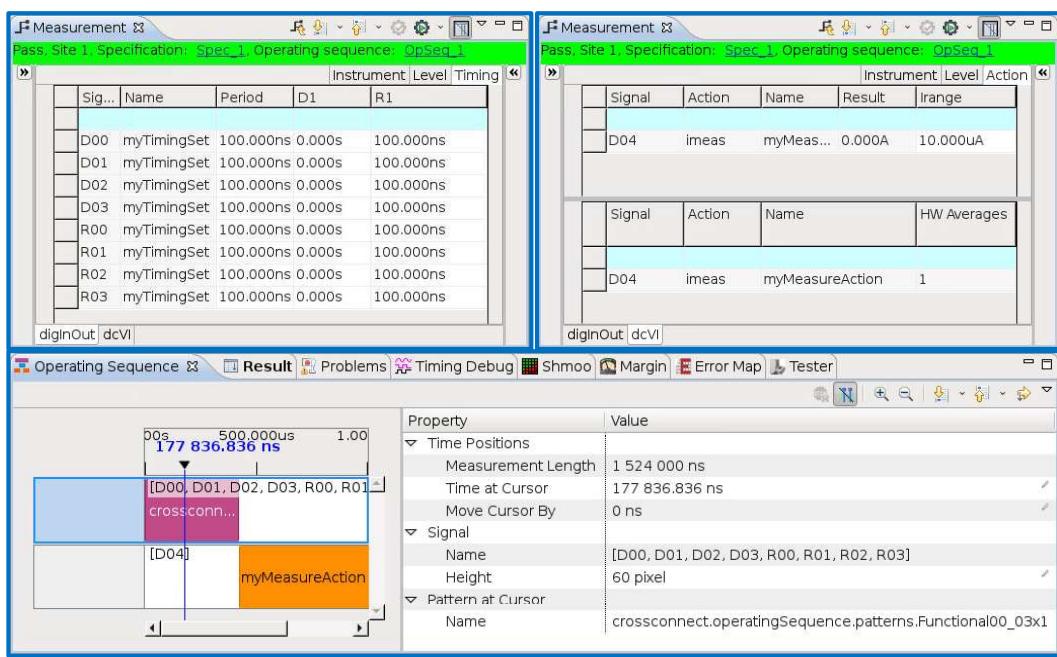
Therefore, an optimization for generated files is switched on per default, so that all generated setup files are unique with respect to their test setup content.

However, then the paths of the generated files no longer follow the calling hierarchy of the corresponding test suite, which is not convenient for debugging.

To switch off the optimization, put a file with the name “.jvm\_options” in the project folder and insert the following line:

`-Ddsa.mode=NO_OPTIMIZATION`

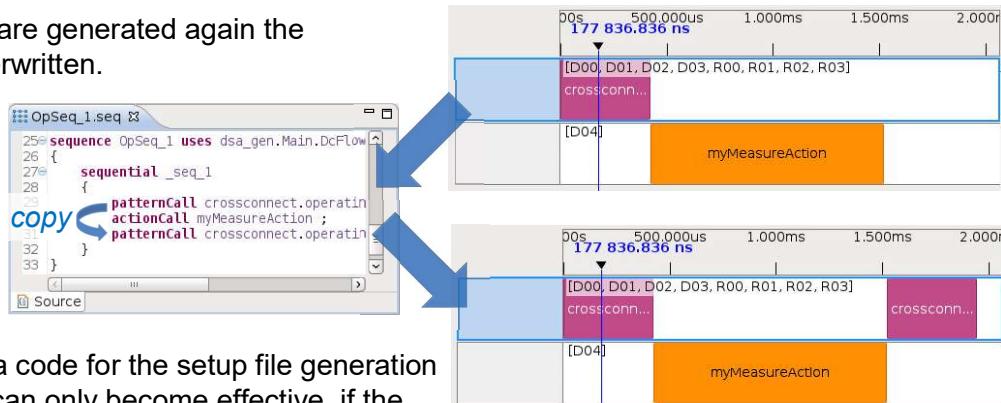
## Debug: Check Setup of Generated Files



# Debug of Programmatically Generated Setups

For debug, generated setup files can be modified and then the corresponding *measurement* can be executed again.

However, once the files are generated again the modifications will be overwritten.



Modifications of the Java code for the setup file generation in the “setup()” method can only become effective, if the

“setup()” part is executed again:

Use “Rebind and Execute” - or - terminate and restart the debug session.

## Summary - What you should have learned

- To generate programmatically test setups, use the *Device Setup API* in the class method “setup()” of test methods.
- The programmatical test setup generation produces setup files that are stored in the subfolder “dsa\_gen” of the source folder.
- The combination of input parameters with programmatical setup generation in the “setup()” method allows to implement generic and flexible test methods.
- Generated specification files can be mixed with manually created specification files by importing the latter into the generated setup.
- Generated setups can be analyzed in different ways while in debug mode:
  - use the *measurement view* to modify instruments settings;
  - change the created *specification* or *operating sequence file*;
  - adapt the code using the *Device Setup API* in the “setup()” method and make sure that it is executed again.



# Input and Output Parameters of Test Methods

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest Test Methods: Parameters - 1

## Learning Objective

- Know how to implement more generic test methods using input parameters.
- Know how output parameters of a test method can be used to access results from outside.



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest Test Methods: Parameters - 2

# Agenda

- Introduction
- Simple test method with input and output parameter
- Test method with an array as an input
- Test method with a collection of *parameter groups* as input

## Example of a Parameter: “measurement”

```
package slideExamples.testMethods;
import xoc.dta.TestMethod;

public class SimpleTestMethod extends TestMethod {
    public IMeasurement measurement;
    @Override
    public void execute() {
        measurement.execute();
    }
}
setup {
    suite SimpleSuite calls SimpleTestMethod {
        measurement.specification = setupRef(simpleSpec);
        measurement.operatingSequence = setupRef(simpleOpSeq);
    }
}
execute { SimpleSuite.execute(); }
```

Test method

Testflow

The “measurement” variable is defined as a **public** member of the test method class.

Because the variable is **public**, it can be accessed from the testflow to set specification and operating sequence of the measurement.

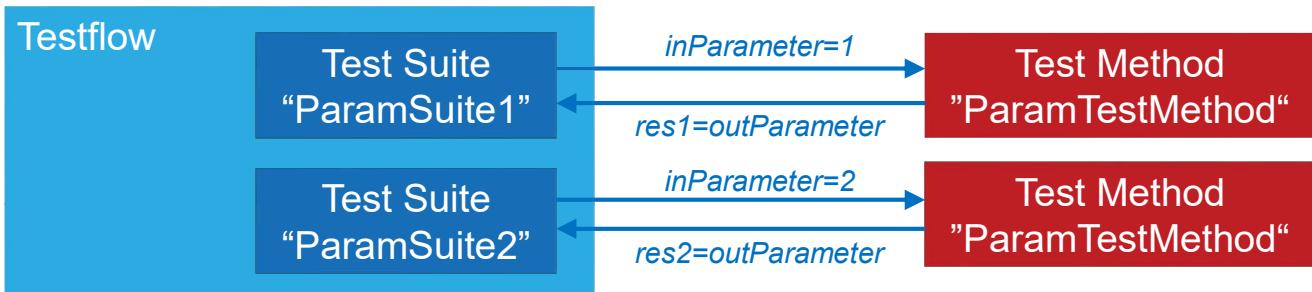
# Parametrization

The parameters of a test suite are basically the input and output parameters of the test method called by the test suite.

Input parameters allow to write parametrized test methods that are more generic.

Setting appropriate values for the input parameters allows to adapt a generic test method to a specific test scenario.

Output parameters provide access to test results of test methods at the testflow level.



*The testflow may use the output of the first suite as input for the second test suite:  
inParameter=res1*

## Three Examples of Test Methods to Burst Patterns

To show the usage of parameters, three different implementations are shown for a test method that burst patterns in the following way:

- Inputs to the test method are paths of patterns and the test method puts these patterns sequentially in an *operating sequence* like a burst.
- After the execution of the test method, which will run the *operation sequence*, the test method returns as output the pass/fail result per site.

# Bursting Three Patterns

The first example of a test method uses three input parameters (public member variables) to allow users to determine the three patterns to be executed in a burst.

The public member variable “results” contains the output: the pass/fail results.

```
public class Burst3Patterns extends TestMethod {           Test method
    public IMeasurement measurement;
    public IFunctionalTestDescriptor mainFtd;
    →@In public String specToExecute = null;
    →@In public String pattern1 = null;
    →@In public String pattern2 = null;
    →@In public String pattern3 = null;
    →@Out public MultiSiteBoolean results = null;

    @Override
    public void setup() {
    ...
}

Test method:
"@In" and "@Out" are recommended annotations.           Testflow: Definition of a test suite that calls the test method.
```

```
suite Burst calls Burst3Patterns {
    specToExecute =
        setupRef(example.specs.Functional);
    pattern1 = "example.pats.patA";
    pattern2 = "example.pats.patB";
    pattern3 = "example.pats.patC";
}
```

Testflow

## Accessing Results with an Output Parameter

The class method “setup()” of the test method generates the *operating sequence* via *Device Setup API* and assigns it to the *measurement* object.

The class method “execute()” runs the *measurement* and the pass/fail results of the *measurement* are assigned to the variable “results”.

```
public class Burst3Patterns extends TestMethod {           Test method
    ...
    @Out public MultiSiteBoolean results = null;

    @Override
    public void setup() {
    ...
}

execute {
    Burst.execute();
    message(5, "Passed: " +
        Burst results.toString()
        + ".");
}
Variable name of the output parameter
Testflow
```

The following “message” is produced for 4 passing sites in the console:

```
[message:5:Burst] Passed: [1: true, 2: true, 3: true, 4: true].
```

## Second Example: Flexible Number of Input Parameters

The previous example of a test method is flexible in that way, that the user can select the test patterns that are sequentially executed in the *operating sequence*.

However, the number of input parameters is fixed to three patterns:

If more than three patterns should be executed in a sequence, then the test method cannot be used.

To allow a flexible number of input parameters of the same type, arrays or collections of *parameter groups* can be used for input parameters.

The next, second example explains, how an array of strings is used as input parameter for the patterns.

## Bursting an Array of Patterns

This example shows how to use an array as input parameter.

The array of strings allows to define patterns that should be run sequentially in a burst.

Any number of patterns can be selected in the test suite definition.

The diagram illustrates the usage of an array of strings as an input parameter. On the left, a code snippet for a **Test method** named `BurstList` is shown. It defines a field `patterns` as a list of strings. On the right, a **Testflow** snippet shows the `patterns` field being assigned a value: an array containing two strings, "example.pats.patA" and "example.pats.patB". A blue arrow points from the `patterns` field in the test method to its corresponding assignment in the testflow. Another blue arrow points from the `patterns` field in the testflow back to the `patterns` field in the test method, indicating they refer to the same variable.

```
public class BurstList extends TestMethod {  
    public IMeasurement measurement;  
    public IFunctionalTestDescriptor mainFtd;  
    @In public String specToExecute = null;  
    @In public List<String> patterns =  
        new ArrayList<String>();  
    @In public Double waitTime = 0;  
    @Out public MultiSiteBoolean results =  
        new MultiSiteBoolean();  
  
    @Override  
    public void setup() {  
        ...  
    }  
}  
  
suite Burst calls BurstList {  
    measurement.specification =  
        setupRef(example.specs.Functional);  
    patterns = # [ "example.pats.patA",  
                  "example.pats.patB" ];  
    waitTime = 100e-6;  
}
```

Declaration and setting of the input array.

Test method

Testflow

# Generation of the Operating Sequence

The example introduces another input parameter “waitTime” of type “Double” which allows users to insert some waiting time between the sequential execution of two patterns.

In the class method “setup()” of the test method the *operating sequence* will be generated using the *Device Setup API* as follows:

```
deviceSetup.sequentialBegin();
for (String singlePattern : patterns) {
    deviceSetup.patternCall(singlePattern);
    if (waitFor > 0) { // insert waiting time if needed
        deviceSetup.waitCall(waitTime);
    }
}
deviceSetup.sequentialEnd();
```

*Test method->setup()*

For each pattern path that is provided in the input array “patterns”, a call of this pattern will be added to the *operating sequence*.

## Waiting Time and Datalogging per Pattern

Now consider the scenario, that the test method should allow these inputs:

- An arbitrary number of patterns to be bursted.
- For each pattern: A specific waiting time after executing the pattern.
- For each pattern: Datalogging of pass/fail cycles

Then defining inside the test method a *nested class*, that extends the provided class “ParameterGroup”, allows to group together pattern path, waiting time and *test descriptor* (needed to enable data logging):

```
public class BurstList extends TestMethod {
```

*Test method*

*nested class*

```
    static public class PatternSetting extends ParameterGroup {
        public String path = "";
        public IFunctionalTestDescriptor patternFtd;
        public Double waitTime = 0;
    }
    @In public ParameterGroupCollection<PatternSetting>
        patternGroup = new ParameterGroupCollection<>();
```

# Definition of an Input Parameter Group

```
static public class PatternSettings extends ParameterGroup {  
    public String path = "";  
    public IFunctionalTestDescriptor patternFtd;  
    public Double waitTime = 0;  
}  
@In public ParameterGroupCollection<PatternSettings>  
patternGroups = new ParameterGroupCollection<>();
```

*Test method*

The new public class “PatternSettings” extends the class “ParameterGroup” and contains member variables for all the data to be grouped together.

The variables of a *parameter group* like “PatternSettings” must be public.

The input parameter “patternGroups” is a collection of the *parameter group* “PatternSettings”.

The collection allows users to define multiple *parameter groups* (consisting of the path to the pattern, the waiting time and the test descriptor) as input to the test method.

# Usage of an Input Parameter Group

```
static public class PatternSettings extends ParameterGroup {  
    public String path = "";  
    public IFunctionalTestDescriptor patternFtd;  
    public double waitTime = 0.0;  
}  
@In public ParameterGroupCollection<PatternSettings>  
patternGroups = new ParameterGroupCollection<>();
```

*Test method*

```
suite Burst calls BurstGroup {  
    measurement.specification =  
        setupRef(example.specs.Functional);  
    patternGroups [ExamplePat1] = {  
        path = "example.pats.patA";  
        waitTime = 10e-6;  
    };  
    patternGroups [ExamplePat2] = {  
        path = "example.pats.patB";  
        waitTime = 5e-6;  
    };  
}
```

*Testflow*

This example of a test suite calling the test method sets up the execution of two patterns in a burst.

So two input *parameter groups* with values are set up for the collection “patternGroups”.

“ExamplePat1” and “ExamplePat2” are names to identify the parameter groups of type “PatternSettings”.

# Parameter Group Example: Creation of the Operating Sequence

For the example using *parameter groups* as inputs, the *operating sequence* is generated via the *Device Setup API* as follows:

```
deviceSetup.sequentialBegin();
for (PatternSettings singleGroup : patternGroups.values()) {
    deviceSetup.patternCall(singleGroup.path);
    if (singleGroup.waitTime > 0) {
        deviceSetup.waitCall(singleGroup.waitTime);
    }
}
deviceSetup.sequentialEnd();
```

*Test method->setup()*

As a result, the generated *operating sequence* will call a pattern as many times as elements of the collection “patternGroups” are defined in the testflow.

# Parameter Group Example: Evaluation and Datalog

The evaluation of tests using the test descriptors can be done as follows:

```
int patternCounter = 0;
for (PatternSettings singleGroup : patternGroups.values()) {
    singleGroup.patternFtd.setTestText("ResultOfGroup" + singleGroup.getId());
    singleGroup.patternFtd.evaluate(
        patternPassFailResults[patternCounter].getSignalPassFail());
    patternCounter++;
}
```

*Test method->execute()*

The source code shows how identifier names of parameter groups can be accessed using the function “getId()”, for example:

The text of tests is “ResultOfGroup**ExamplePat1**” and “ResultOfGroup**ExamplePat2**”.

The test descriptors are identified in the “Tests” sheet of the *test table* by:

Burst.patternGroups[ExamplePat1].patternFtd

Burst.patternGroups[ExamplePat2].patternFtd

# Summary - What you should have learned

- Input and output parameters of a test method must be defined as public member variables of the test method class.
- The output parameter of a test suite can be accessed in the “execute()” part, use <test\_suite\_name>.<variable.name>.
- For multiple values of the same data type as input, arrays can be used to allow users to define any number of input values
- Multiple input setting that belong together can be assembled in a *parameter group*.
- A *parameter group* is set up as a public class in the test method class, enhancing the “ParameterGroup” class.
- Users can provide values for multiple *parameter groups* as input of a test method, if it defines as input variable a collection of *parameter group* objects.  
Then the input is provided by specifying elements of the collection.
- *Parameter groups* can be used to create *test descriptors*.



# Debug Tools

SmarTest 8.2.5 Training

January 2020

# Learning Objectives

- Understand the available debug tools
- Understand which debug tool to use depending on the debug scenario

# Agenda

- First step: Launching debug
- Introduction of debug tools:
  - *result view*
  - *site result view*
  - *measurement view*
  - *operating sequence view*
  - *timing debug view*
  - *pattern editor*
  - *error map*
  - *tester view*
  - *instrument view*
  - *Flow Chart view*
- Use case: setup data vs. hardware state

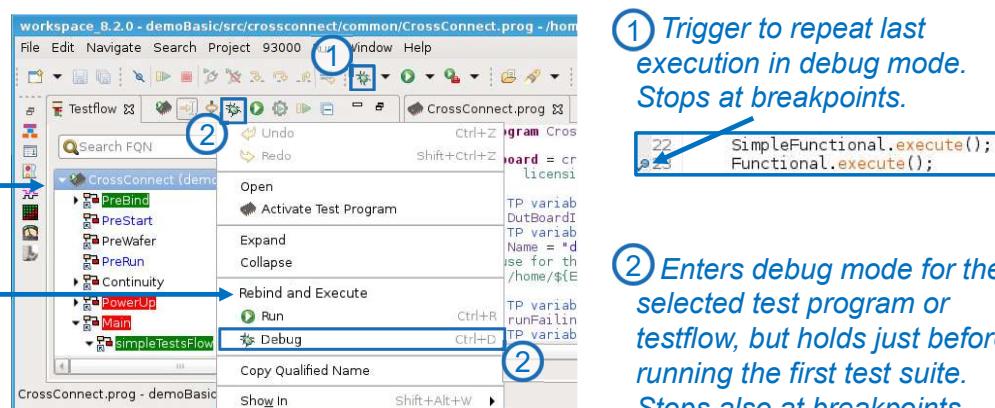
# First Step: Start Debug of Test Programs or Testflows

Debug tools show the setup and results of the last executed *measurement* including the resulting state of the tester hardware.

The debug mode allows to control, which *measurement* is to be the last executed one by setting breakpoints or by executing single test suites.

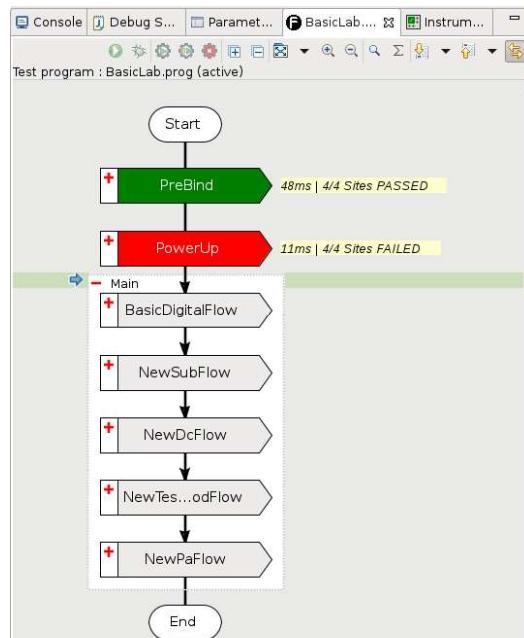
Select testflow or test program for debug and right-click

Resets setups to the content of file



## Start Debug from Flow Chart view

The Flow Chart like the testflow view allows you to debug or execute a test program or testflows without requiring manual setup of breakpoints beforehand.



# Result View

The screenshot shows the SmarTEST Result View window. At the top, there's a toolbar with various icons. Below it is a table titled "Results of the last 500 execution(s) run". The table has columns: Fully Qualified Name, Site, SW Bin, P/F, DUT Time, Time, and Testflow File. A blue circle highlights the "P/F" column header. A red row in the table is labeled "Fail". The bottom of the window has tabs: Test Program, Testflow, Test Suite, Test, and Signal.

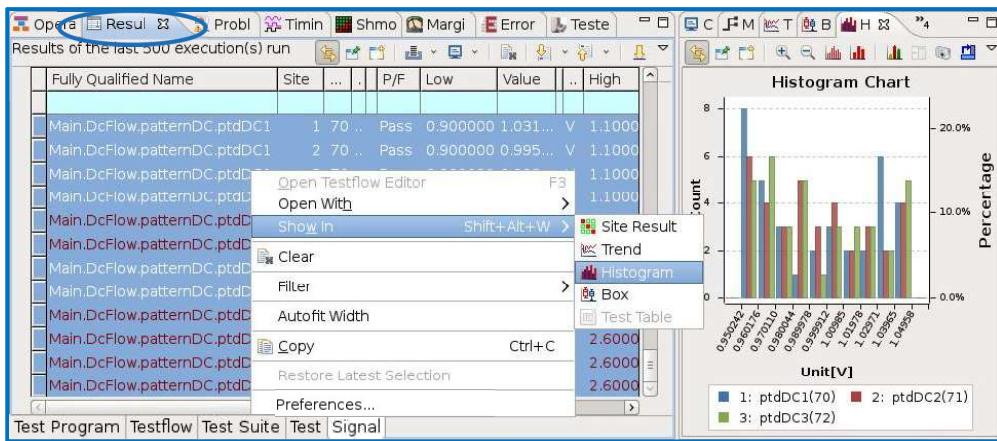
The **result view** is a GUI for real time updated test results, pass/fail and parametric values.

- Four tabs show test results of different granularity of details.
- Display filters, annotations, and navigation allow to locate failures quickly.
- Navigation to testflow and statistic windows in *analysis perspective*.
- Ad-hoc generation of graphs (histogram, trend, shmoo) from selected results.

## Result View - Tabs

The image contains two screenshots of the SmarTEST Result View. The top screenshot shows the "Test Suite" tab. It has a table with columns: Fully Qualified Name, Site, P/F, Test Suite Time, and Foreground Time. A blue circle highlights the "Foreground Time" column. The bottom screenshot shows the "Test" tab. It has a table with columns: Site, Test#, Type, Pattern, P/F, Low, Value, Unit, High, and Test Text. Blue arrows point from text labels to specific parts of the interface: one arrow points to the "Foreground Time" column with the text "Shows time differences because of releaseTester()", and another arrow points to the "P/F" column with the text "Fail markers".

# Navigation between Debug Tools

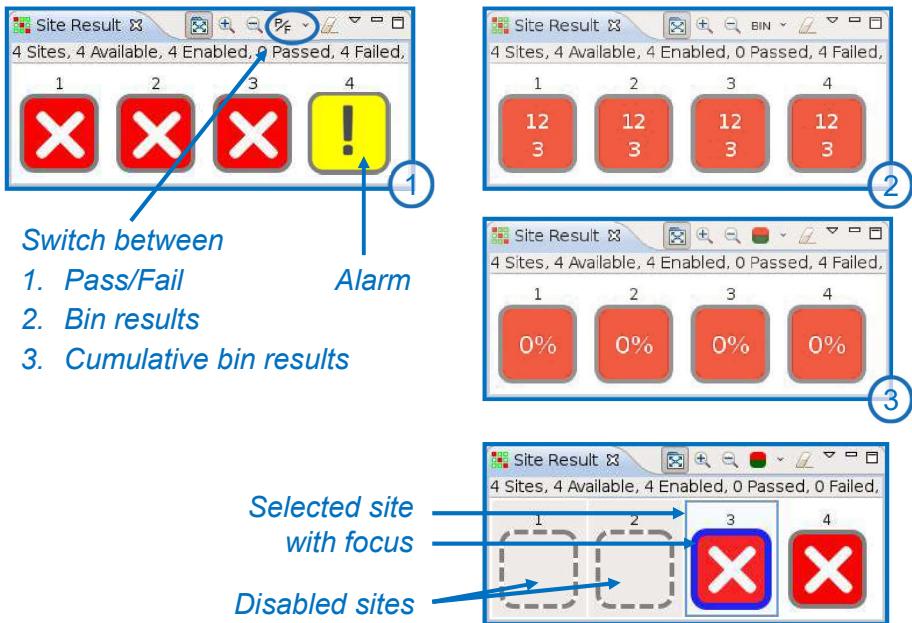


Debug tools allow to navigate to other tools by mouse right-click and then select “Open With” or “Show in”.

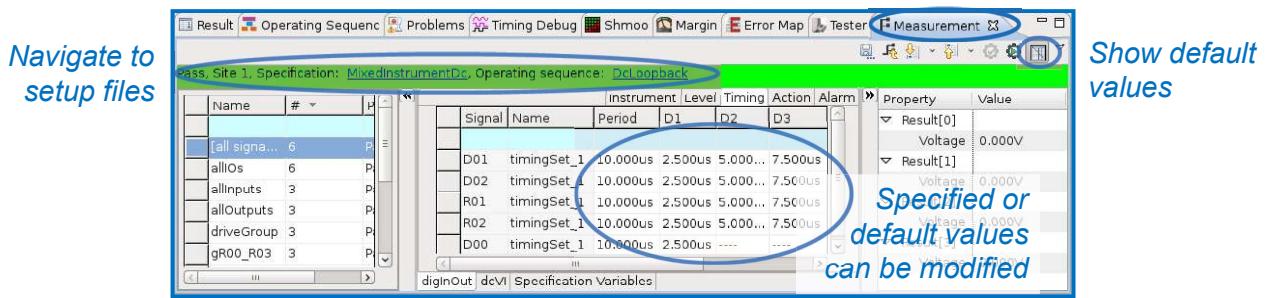
## Site Result View

The **site result view** is a GUI for displaying per site pass/fail and binning results graphically.

- Support for high site count.
- Allows to globally control the focus on the sites for that results are shown in other debug tools.
- Allows to select one or multiple sites (pass/fail/alarm)
- Allows to enable or disable selected sites



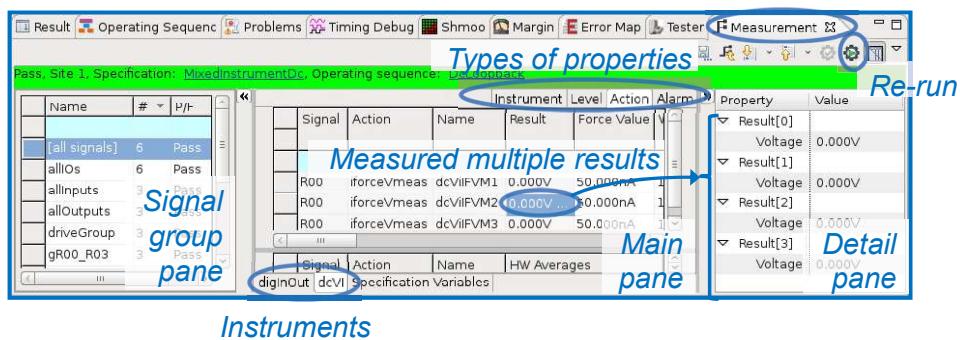
# Measurement View - Overview



The **measurement view** is a GUI for inspecting the specification data downloaded to the tester channels according to the setup of the last executed **measurement**.

- Specified values (also evaluated from equations) and default values (optionally) are displayed when the execution is paused.
- Property values can be modified on the spot to overwrite defined or default specification values.
- Provides navigation to *timing debug view*, *pattern editor*, setup files, etc.

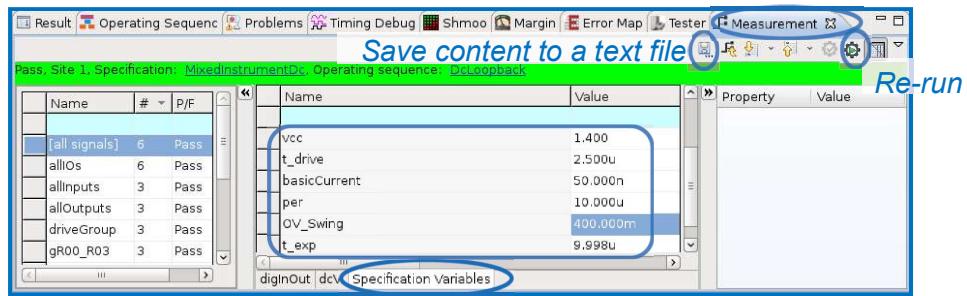
## Measurement View – Panes



The **measurement view** consists of the **main pane**, the **signal group pane** and the **detail pane**.

- **Signal group pane**: Select a signal group, then only for this group data is shown in the **main pane**.
- **Main pane**: Signal settings are grouped by *instruments* and their specific properties + alarms.
- **Detail pane**: Results of the entire *measurement*, per signal pass/fail, and values of *actions* are displayed.

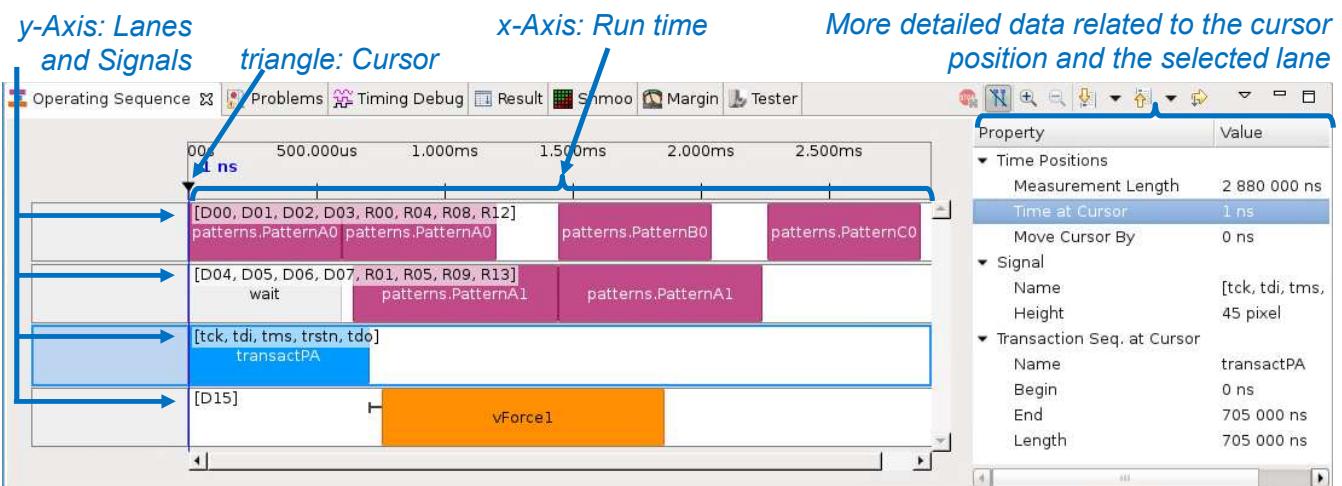
# Measurement View – Re-run the Measurement



## The measurement view – modify, re-run and save settings:

- Modified settings can be applied to the tester again by re-running the last executed *measurement*.
- In particular, values of specification variables are displayed and can be modified and then applied to the tester again by re-running the *measurement*.
- The *measurement view* allows to save to a text file the content like
  - specified settings,
  - retrieved results and
  - reported alarms.

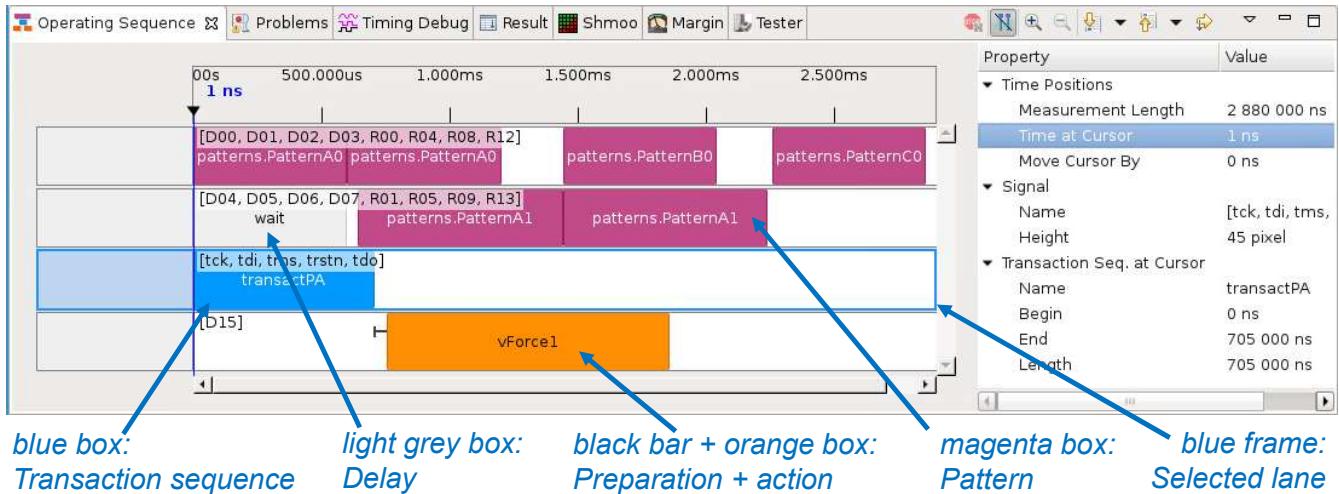
# Operating Sequence View



## The operating sequence view is a GUI to view the timing profile of an operating sequence.

- Visualization of the execution sequence of patterns, *actions* and protocol *transaction sequences*.
- Measure time duration with a cursor and markers.

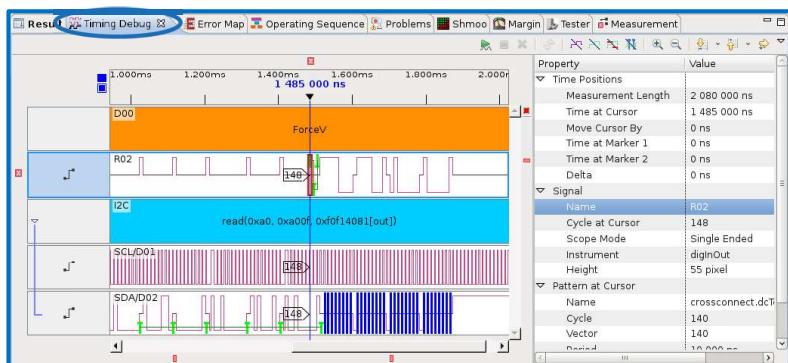
# Operating Sequence View – Building Blocks



**The operating sequence view: Visualization of the different types of building blocks.**

- Fails are displayed as overlay annotations on patterns, *actions* and *transaction sequences*.
- Navigation to referenced setup entities and other debug tools.

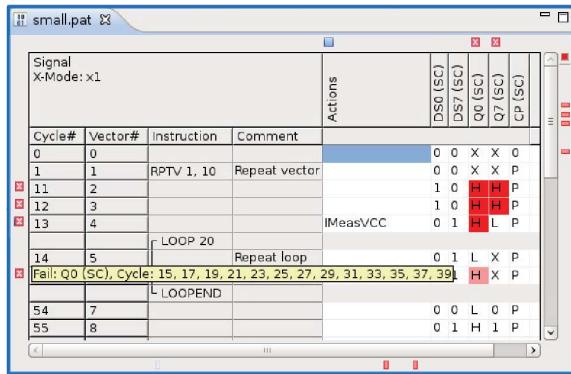
# Timing Debug View



**The timing debug view is a GUI for debugging timing and level results.**

- Multi-domain drive/received data (digital and transaction sequences) and RF, DC and mixed signal actions are shown on a common time axis allowing zooming and resizing in both dimensions.
- Automatically gathers fail results → fail annotations and navigation.
- Soft scope functionality to capture true waveform from device.
- Seamless navigation to pattern editor, error map, setup files, etc.

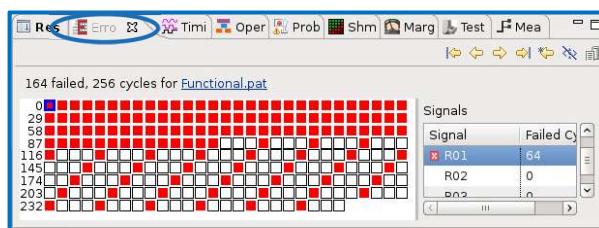
# Pattern Editor



**The pattern editor** is a GUI for editing & displaying patterns and its fails.

- Fail results are displayed as overlay.
- Overview ruler and annotation icon to locate fails quickly.
- Detailed information of triggered actions in properties view.
- Seamless navigation to desired signals/cycles in other debug GUIs by “Show In”.

# Error Map View



**The error map view** is a GUI for displaying locations of failed cycles graphically on a flattened pattern.

- Easy and fast navigation to previous/next or first/last failed cycle.
- Masking and unmasking of unwanted signals.
- Dumping failed cycles in plain ASCII format.
- Seamless navigation to desired signal/cycle in other debug GUIs by right-click and “Show In”.

# Tester View

Signal	DUT Signal	Pogo Pin	Instrument	Product Name	Option	Line	State	Last Used In
D15	D15	10215	dcVI	PS1600	....	0.000V	PowerUp.DpsOn.measurement	
R00	R00	10102	diginOut	PS1600	....	hold	Main.dcfLow.patternDC.measurement	
R01	R01	10104	dcVI	PS1600	....	....	Main.dcfLow.patternDC.measurement	
R02	R02	10106	dcVI	PS1600	....	-10.000uA	PowerUp.ContinuityFlow.ContinuityTest...	

Overview

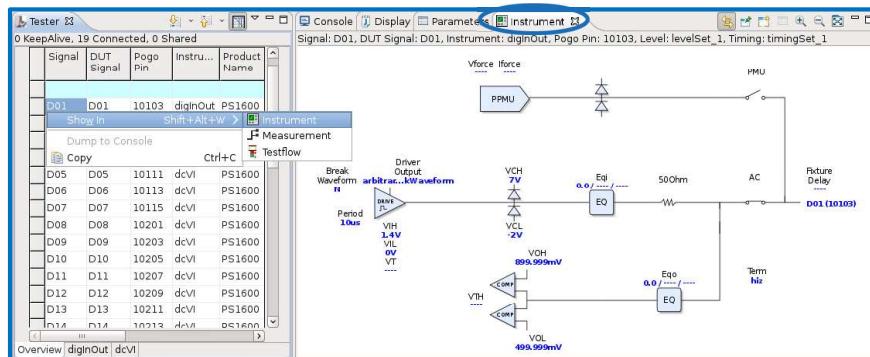
Si...	DUT Sig...	Co...	Keep Alive	Connect State	Timing	Period	Sequence...	Level	Term	VIL	VIH	VT	VOL
D07	D07	true	false	singleE...	timingSet_1	88.000ns	176.000ns	levelSet_1	hiz	0.000V	1.400V	....	499.999n
D08	D08	true	false	singleE...	timingSet_1	66.000ns	198.000ns	levelSet_1	hiz	0.000V	1.400V	....	499.999n
D09	D09	true	false	singleE...	timingSet_1	66.000ns	198.000ns	levelSet_1	hiz	0.000V	1.400V	....	499.999n

Details of signals last configured as digInOut

The **tester view** is a GUI for inspecting the current state of the tester hardware.

- Current states of active *instruments* are displayed at a glance.
- Includes *instruments* activated outside of the current context (e.g. for keep alive).
- Displayed states of signals are organized by the *instrument* type which corresponds to the “last configured” type.
- Displayed states are read-only. Tester states can be saved to a text file.
- Provides navigation to most recently used *instrument* state.

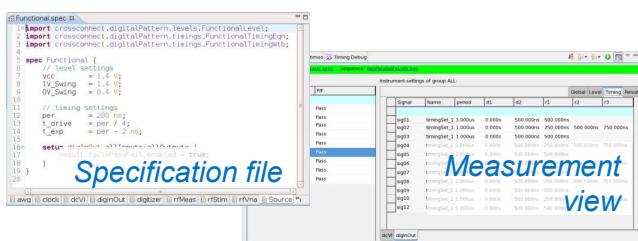
# Instrument View



The **instrument view** is a GUI to display for the selected *instrument* the current state in schematic diagrams during debugging.

- More detailed hardware states can be displayed.
- Tight linkage to *tester view*, synchronized with *measurement view* as needed.

# Use Model – Setup Data vs. Current State

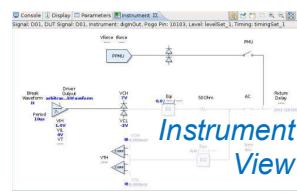
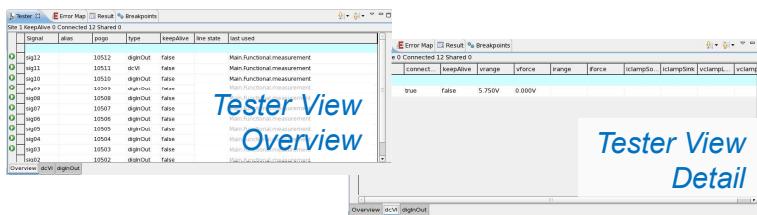


## Setup as it is downloaded to tester channels

- Make sure that the specified setup is as expected and equations are computed correctly.
- Modify setup properties on the spot and re-run it.
- The most recently activated/executed measurement can be examined.

## State of tester channels

- Make sure that the desired setup is applied to the hardware.
- Make sure hardware's current state matches with the intended setup.



ADVANTEST

SmarTEST 8

January 2020

All Rights Reserved - ADVANTEST CORPORATION

SmarTest 8 Debug Tools - 20

## Summary - What you should have learned

- Debug tools can be only used when in executing a test program or testflow in debug mode.
- Debug tools show the setup, test results and resulting hardware state of the last executed *measurement*.
- The *result view* shows what test were run and displays the test results written to datalog.
- The *site result view* displays binning results and allows to control enabled and disabled sites
- The *measurement view* displays the instrument setups of the last *measurement* in tables and allows to modify the setups and rerun the *measurement*.
- The *operating sequence view* and the *timing debug view* visualizes based on a time axis how a test and its basic test steps like *patterns*, *actions* and *transaction sequences* are executed.
- The *pattern editor* and *error map* are helpful for debugging digital patterns.
- The *tester view* and the *instrument view* show the state of the hardware in the test head.

SmarTEST 8

ADVANTEST

January 2020

All Rights Reserved - ADVANTEST CORPORATION

SmarTest 8 Debug Tools - 21



# Characterization Tools

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest 8: Characterization Tools - 1

## Learning Objective

- Know the characterization tools in SmarTest 8
- Know how to use the *shmoo tool* and the *margin tool*



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest 8: Characterization Tools - 2

# Agenda

- Basics
- Shmoo over a *measurement* execution
- Shmoo over a test suite or over a testflow
- *Margin tool*

## Concept of Characterization tools

Characterization tools are used to gain insight into the performance of the devices at different process corners.

The obtained data is valuable for Design/Product Engineering to improve yield and to feed back learnings into the design of upcoming devices.

For characterization, the *shmoo* or *margin tool* are used, which allow to run multiple times a test while modifying settings for some setup resources.

The user defines, which settings and how settings are varied for the multiple executions.

The results are graphically visualized to facilitate interpreting the results.

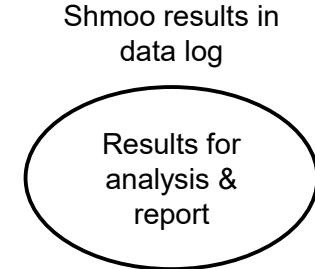
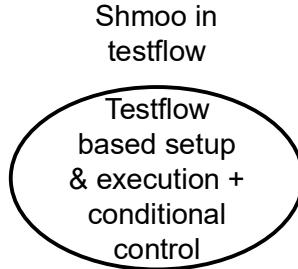
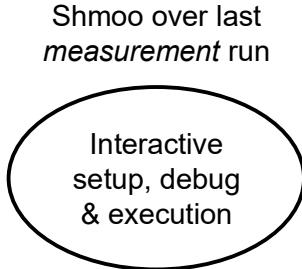
Example of a characterization test setup:

The voltage levels of power supplies are varied in order to find out, for which voltage range good DUTs are working in a reliable way.



# Use Models

The different use models and their purpose, given for example for shmoo.

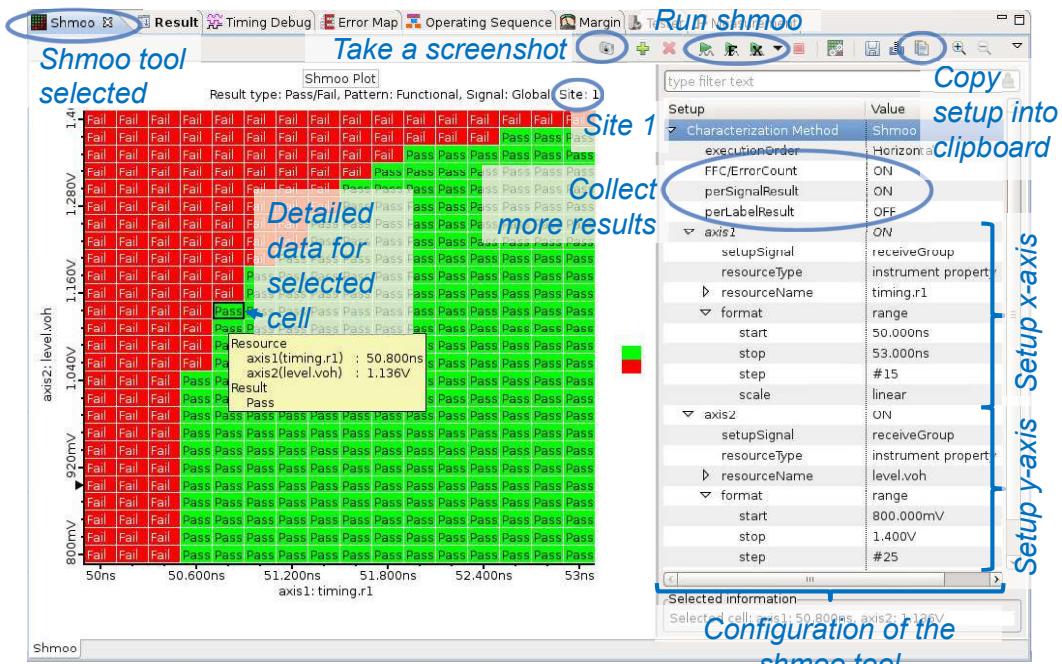


## Shmoo Tool: Properties

### The *shmoo tool*

- Can be interactively configured and run in debug mode over the last executed *measurement*.
- Can be set up in testflow files to run over test suites or even testflows.
- Supports configuration of tracking parameters that change values during the shmoo run aligned with the changes of other parameter values.
- Works with all instruments supported by SmarTest of the domains DC, digital, analog and RF instruments.
- Works with tests using protocols.
- Visualizes graphically the results.
- Stores results in datalog.

# Shmoo over Measurement in Debug Mode



SmarTEST 8

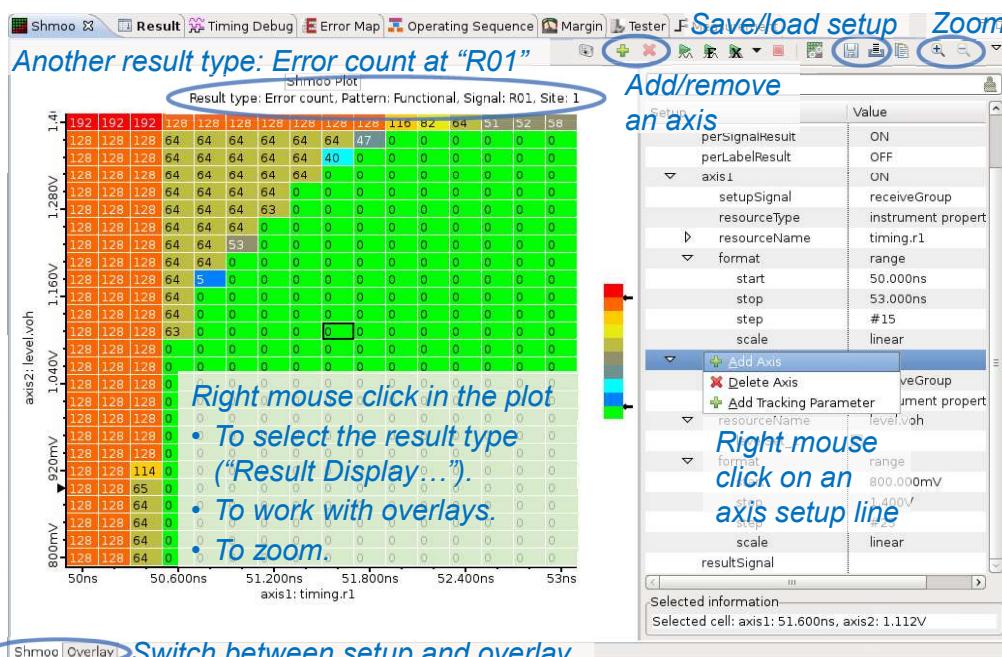
January 2020

All Rights Reserved - ADVANTEST CORPORATION

ADVANTEST

SmarTest 8: Characterization Tools - 7

## Shmoo Plot: Error Count



SmarTEST 8

January 2020

All Rights Reserved - ADVANTEST CORPORATION

SmarTest 8: Characterization Tools - 8

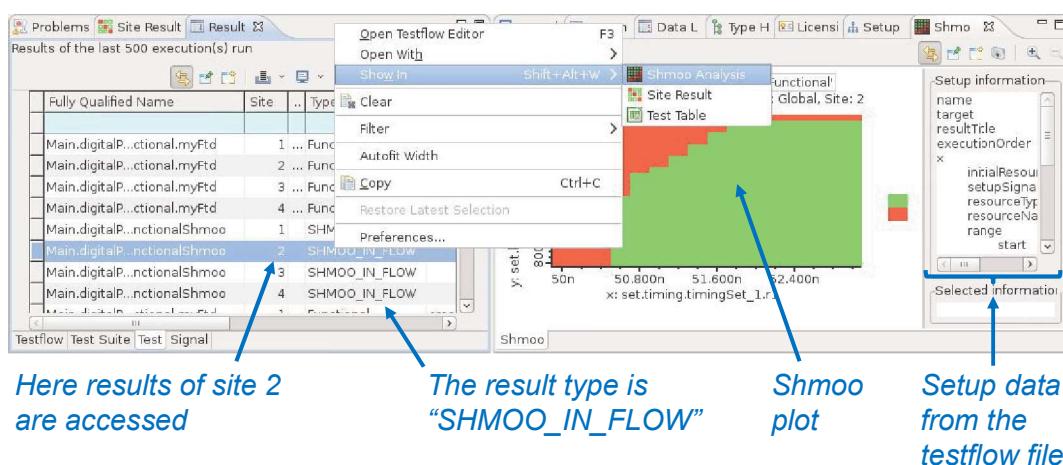
## Shmoo Setup in Testflow Files: Setup

The *shmoo tool* is configured in a *testflow file* in a similar way as the interactive shmoo with respect to the properties and keywords.

```
60: shmoo {SimpleFunctionalShmoo {
61:     target = SimpleFunctional;
62:     resultTitle = "Shmoo over test suite 'SimpleFunctional'";
63:     executionOrder = horizontal;
64:     axis[x] = {
65:         setupSignal = "receiveGroup";
66:         resourceType = instrumentProperty;
67:         resourceName = "set.timing.timingSet_1.r1";
68:         range.start = 50e-9;
69:         range.stop = 53e-9;
70:         range.steps = 15;
71:     };
72:     axis[y] = {
73:         setupSignal = "receiveGroup";
74:         resourceType = instrumentProperty;
75:         resourceName = "set.level.levelSet_1.voh"; Use
76:         range.start = 0.8; "Copy setup into clipboard"
77:         range.stop = 1.4;
78:         range.steps = 25;
79:     };
80: }
```

## Shmoo Setup in Testflow Files: Results

The results of a *shmoo* over a *testflow file* can be easily accessed from the *result view* with a mouse right click.



# Shmoo Setup: Resource Types

SmarTest allows to run the *shmoo tool* using three different resource types:

- Properties of *instruments* – example given above.
- Variables defined in *specification files* – example:

*Shmoo over a specification variable*

```
80 axis[x] = {  
81     setupSignal = "receiveGroup";  
82     resourceType = specVariable;  
83     resourceName = "crossconnect.digitalPattern.mainSpecs.FunctionalComplete.t_exp";  
84     range.start = 50e-9;  
85     range.stop = 53e-9;  
86     range.steps = 15;};
```

*"resourceName" is the fully qualified name of the specification file + the name of the variable*

*Number of steps*

- Input parameters of test suites (only in *testflow files*) – example:

*Shmoo over the input parameter of a test suite*

*Note: No "setupSignal" setting*

```
72 axis[x] = {  
73     resourceType = suiteParameter;  
74     resourceName = "specValue";  
75     range.start = 50e-9;  
76     range.stop = 53e-9;  
77     range.resolution = 0.2e-9;};
```

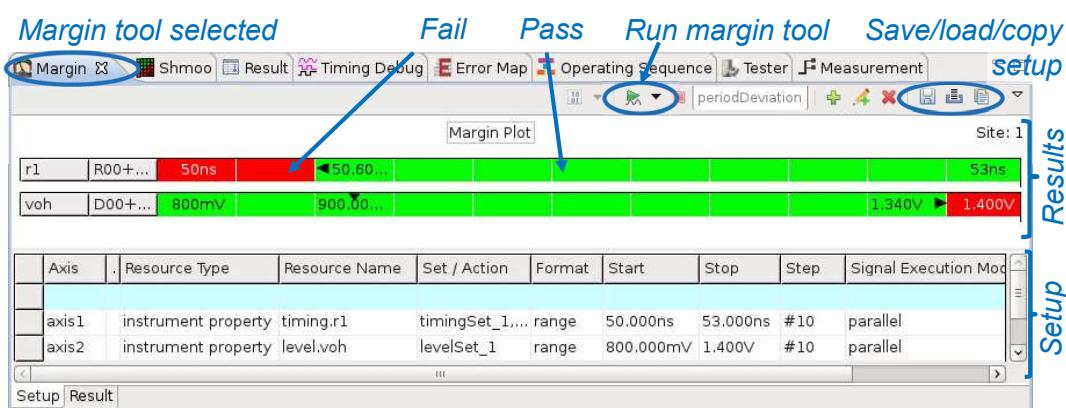
*Step width*

# Margin Tool: Introduction

The *margin tool* is quite similar to the *shmoo tool*:

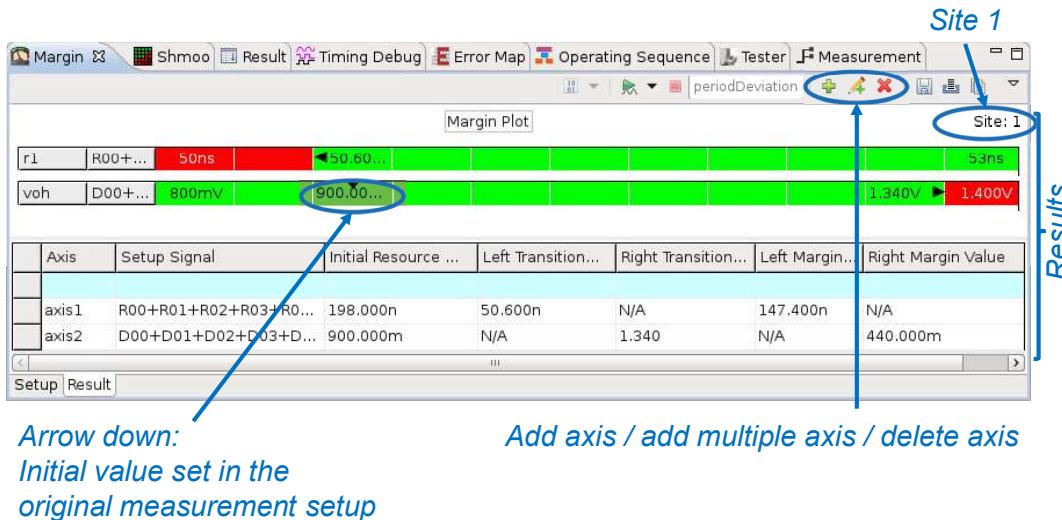
It can be interactively configured in debug mode (as shown below).

Or it can be setup in a *testflow file*.

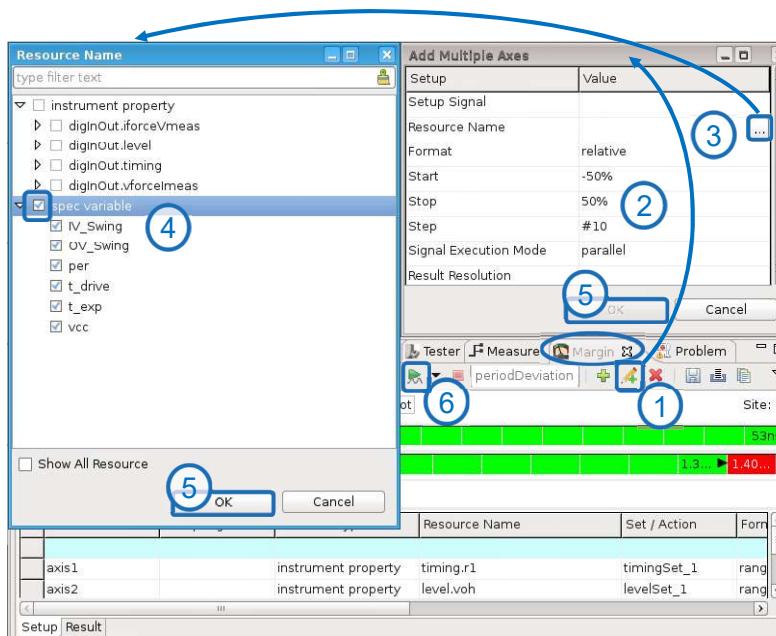


# Margin Tool: Result Tab

Here is the “Result” tab of the interactive *margin tool* shown.



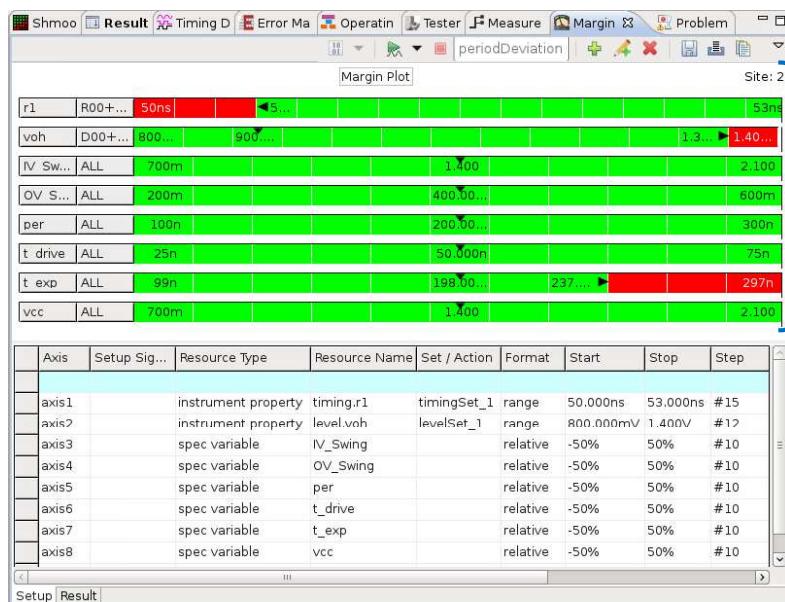
# Margin Tool: Quickly Adding Multiple Axis



**Example:**  
Run margin tool over all spec variables:

1. Select “Add multiple axis”.
2. Adapt tool setup.
3. Open window to select resources.
4. Select “spec variable”.
5. Press “OK”.
6. Run margin tool.

# Results After Adding Multiple Axis



# Margin Tool: Configuration in a Testflow File

The figure shows a code editor window displaying a testflow file named "MarginToolExample.flow". The file contains a flow definition with setup, margin, and execute sections. The margin section configures a SimpleFunctionalMargin object with various parameters like target, resultTitle, and multiple axis definitions. Each axis is defined with its setupSignal, resourceType, resourceName, range, and resolution.

```

1 flow MarginToolExample {
2     setup {
3         suite SimpleFunctional calls acLib.ExecuteTest {
4             measurement.specification = setupRef(crossconnect.digitalTests.mainSpecs.Functional);
5             measurement.pattern = setupRef(crossconnect.digitalTests.patterns.Functional);
6         }
7         margin SimpleFunctionalMargin {
8             target = SimpleFunctional;
9             resultTitle = "Margin over test suite 'SimpleFunctional'";
10            axis [ t_exp ] = {
11                setupSignal = "receiveGroup";
12                resourceType = specVariable;
13                resourceName = "crossconnect.digitalTests.mainSpecs.Functional.t_exp";
14                range.start = 50e-9;
15                range.stop = 53e-9;
16                range.steps = 15;
17            };
18            axis [ voh ] = {
19                setupSignal = "receiveGroup";
20                resourceType = instrumentProperty;
21                resourceName = "set.level.levelSet_1.voh";
22                range.relativePercentage.start = -100;
23                range.relativePercentage.stop = 100;
24                range.resolution = 0.18;
25            };
26        }
27    }
28
29    execute {
30        SimpleFunctionalMargin.execute();
31    }
32 }

```

## Summary - What you should have learned

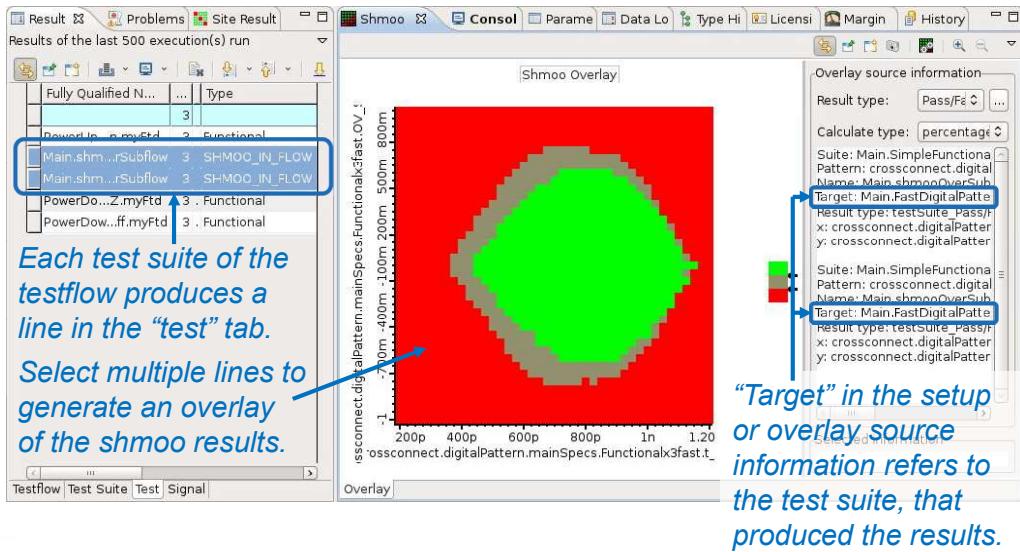
SmarTest provides state of the are characterization tools that allow to run tests while modifying settings for some setup resources. The mains tools used for characterization are:

- Shmoo over measurement
- Shmoo over testflow
- Margin tool

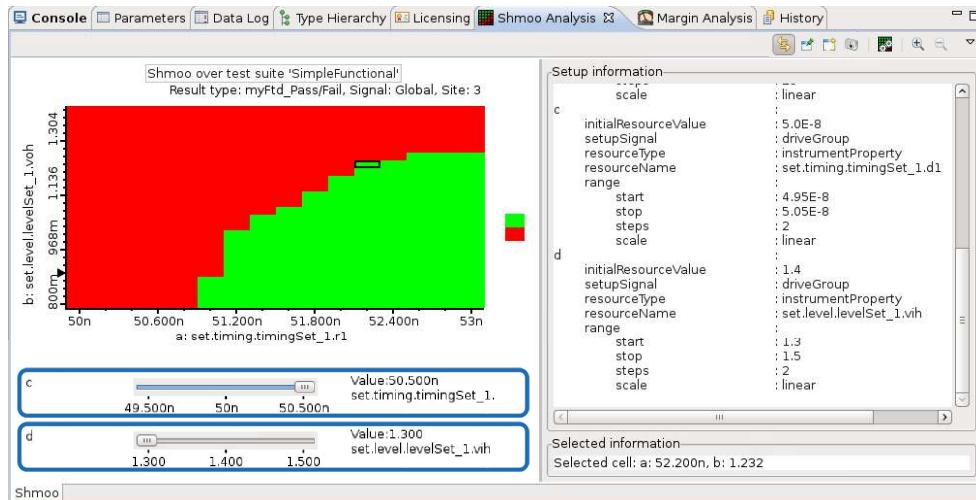
## Backup

# Shmoo over Testflow: Results

Results of running the *shmoo tool* over a testflow containing two test suites.



## Results of a Shmoo Setup with Four Axis



If more than two axis are set up, then sliders are used for these additional axis.



# Recommendations for the Structure of Setup Files

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved – ADVANTEST CORPORATION

ADVANTEST

SmarTest Setup File Structure - 1

## Learning Objective

Understand the recommendations how to structure test setups for SmarTest in various files and folders, in order to:

- Facilitate debugging and maintenance of SmarTest test programs.
- Support reuse of test settings, of test program parts or of full test programs.
- Meet requirements for collaborative development and revision control.



January 2020

All Rights Reserved – ADVANTEST CORPORATION

ADVANTEST

SmarTest Setup File Structure - 2

# Agenda

- Introduction: The flexibility of SmarTest setups
- Splitting up setup data into various files
- Recommendations
  - Signal group definition
  - The one uniquely named subfolder for all setup data
  - Dedicated folders for test categories
  - Specific “common” folder
  - Dedicated projects for test methods
  - Names of folders and files
- Summary

## Test Program Setups in SmarTest

SmarTest gives significant flexibility to users to handle setup data:

- For levels, timings, *actions*, groups, etc. of a test, the user decides, in how many different *specification files* these test settings are stored.
- The user is free to spread the setup files over various folders.
- The user has control on how files and folders are named.

These recommendations are guidelines on how to utilize the flexibility of SmarTest 8.

Here is the first recommendation:

**Source folders should contain only test program setup data.**

Big chunks of data, that is not test program setup data but stored in source folders, will decrease the performance of SmarTest.

# Splitting Up Setup Data and Reuse

General recommendation for splitting up setup data:

**If a part of setup data is reused in a different way than other parts of setup data, then the setup data should be partitioned according to reuse and the different parts should be stored in separate files.**

Example:

A digital test setup contains a level setup, that is the same for all other tests, and a timing setup, that is only used for this specific test.

Then the test setup should be split up in two files:

- The first file contains the level setup that is re-used by all tests:  
The setups of all tests import this file.
- The second file contains the timing setup that is not reused at all.

# Splitting Up Data in Specification Files

**If specification data is reused differently by various test suites, then**

- **timings,**
  - **levels,**
  - **declarations of common variables, common definitions of signal groups and signal aliases,**
  - **and other, specific settings**
- should be split up into separate *specification files*.**

A *specification file* might assign values to variables that affect timing, levels and other settings – in this case splitting makes typically no sense.

**If specification data (combination of timing, levels and other settings) is always used the same way, then this should be put into a single file, that is reused.**

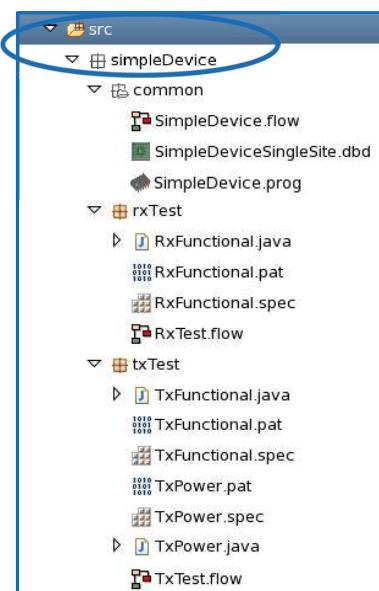
# Signals and Signal Groups

Signal aliasing and signal group definitions in *specification files*:

- Global, i.e. used in many places in the whole test program:  
Should be stored in a dedicated *specification file* in the folder “common”.
- Local, e.g. only used to make a single timing setup simpler:  
Should be stored in the *specification file* where it is used.

Specification variables should be handled in a similar manner.

## Uniquely Named Subfolder for All Data

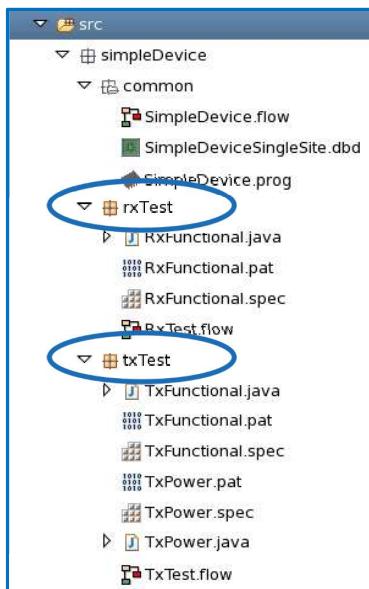


Recommendation to prepare for test program reuse:  
**A source folder should contain a single subfolder, that contains all setup files.**

The unique name of this folder should describe its content.

Example:  
Source folder: “src”.  
Subfolder: “simpleDevice”.

# Next Level of Folders: Test Categories



Recommendations:

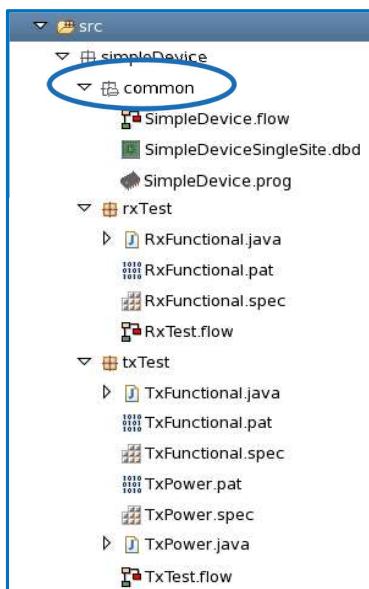
**The uniquely named subfolder should only contain folders of test categories.**

**All tests should be grouped in test categories.**

**All setup files used only for a certain test category should be pulled together in the corresponding folder.**

Examples: “rxTest”, “txTest”.

## The “common” Folder



Recommendation:

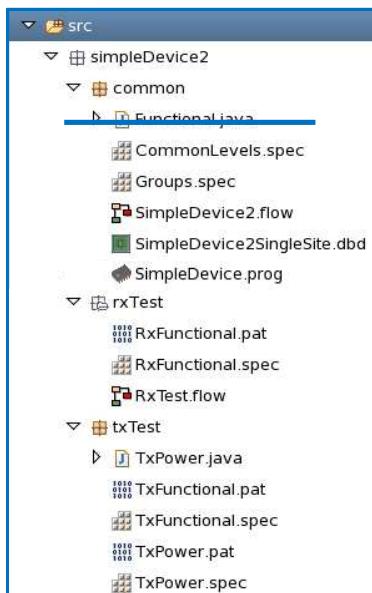
**A folder named “common” should be added to the folders of test categories.**

**Purpose:** It should contain setup data that is common for tests of (almost) all categories.

*Test program files, DUT board description files and the main flow files should always be stored in “common”.*

The folder might also contain test methods or files to set up common signal groups, level settings, specification variables, signal aliases, protocol signal assignments etc.

# Dedicated Projects for Test Methods



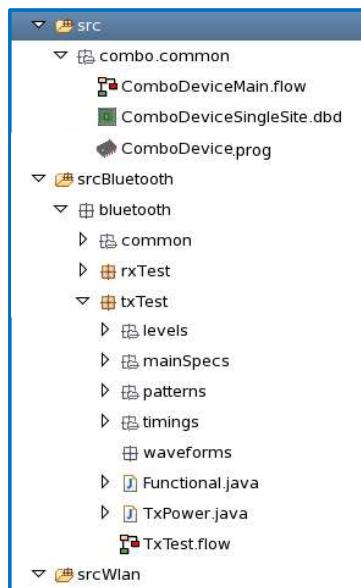
Recommendation:

**User specific test methods shared between multiple devices should be stored in a test method library that is developed in a dedicated SmarTest project.**

SmarTest provides a library of basic test methods.

A folder of a test category should contain only test methods, that will not be reused by other devices.

## Names of Subfolders



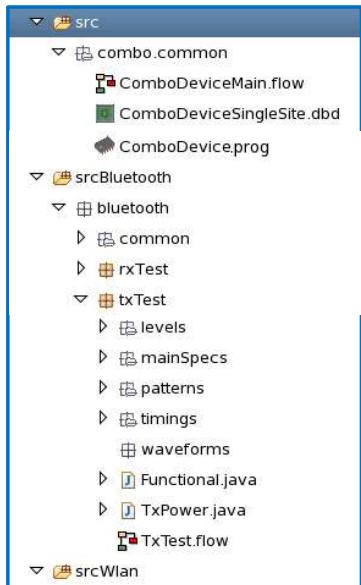
Recommendation:

**If a folder of a test category contains too many files, then group these into subfolders according to the type of settings they contain.**

**Consistently use the same names for the subfolders:**

- “**mainSpecs**”
- “**levels**”
- “**timings**”
- “**patterns**”
- “**waveforms**”
- “**opSeqs**”
- “**trSeqs**”

# Naming of Files and Folders

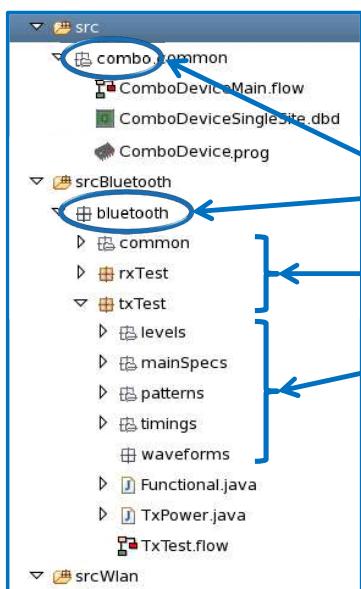


Recommendation for the case of first characters of names:

**Files:** Start with an upper case letter.

**Folders:** Start with a lower case letter.

## Summary - What you should have learned



- Split up setups in multiple files only if needed for reuse or collaborative development
- Folder names should start with a lower case letter; file names should start with an upper case letter.
- Insert in *source folders* uniquely named subfolders to prepare for reuse.
- Use test category folders like “rxTest”, “txTest”. Additionally: “common”.
- To structure the setup files in test category folders, consolidate setup files with similar type of content in subfolders with fixed names.



# Utility Lines

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest 8: Utility Lines - 1

## Learning Objective

- Understand the purpose of utility lines
- Know how to define and use utility lines in test applications



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest 8: Utility Lines - 2

# Agenda

- Introduction
- Utility lines hardware
- Control utility lines via routing in the *DUT board description file*
- Control utility lines as *utility instruments*
- Debugging utility lines

## Introduction: Utility Lines

Utility lines are used to access and control electronic circuits on the DUT board, for example:

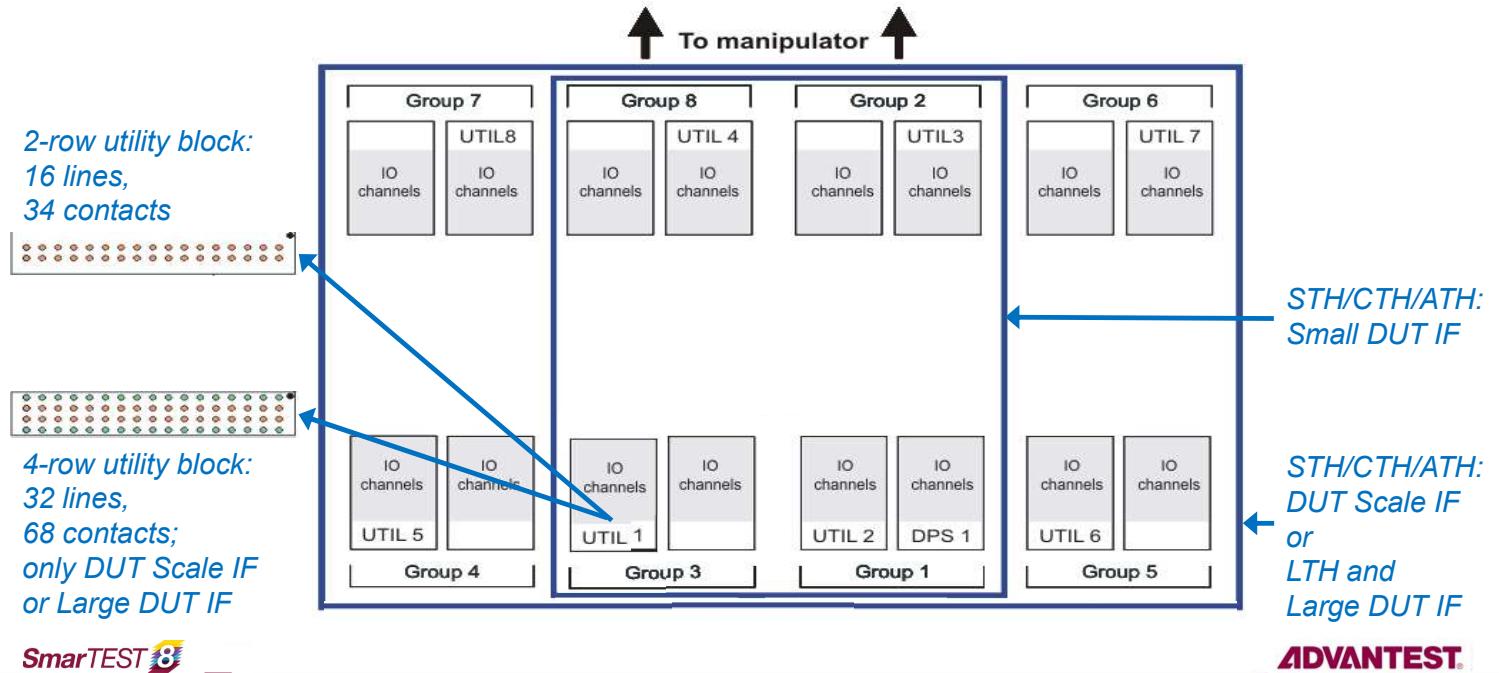
- switch relays on the DUT board,
- read DUT board EPROMs,
- connecting external instruments.

The test heads of the V93000 provide utility lines that are available through separate blocks of pogo pins.

Utility lines can be implicitly controlled if they are set up in the *DUT board description file* as selectors for routing *signals* from pogo pins to DUT pins.

Utility lines can be explicitly controlled in Smartest by using them as *utility instruments*.

# Basics: Test Head Location of Utility Lines



# Basics: Utility Lines Hardware

This is the pogo pin assignment of the 2-row utility line pogo blocks.

The table shows the pogo pin assignments for the 2-row utility line pogo blocks, divided into two blocks:

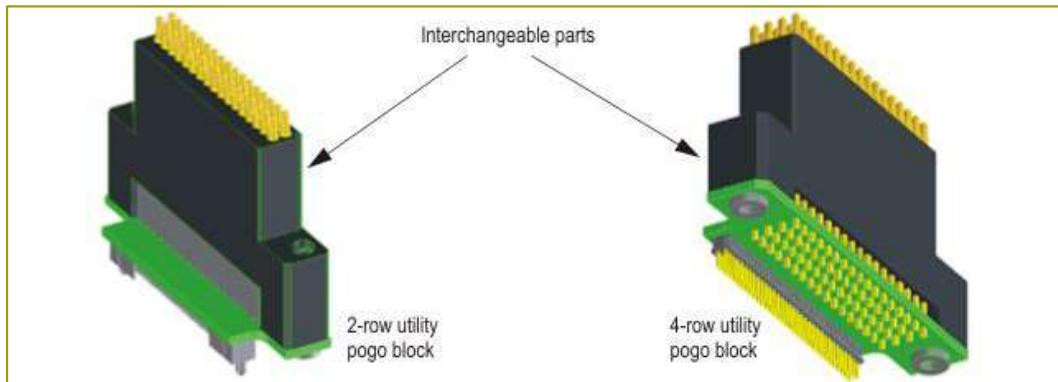
PIN NO	UTILITY		Block 1, 3, 5, 7		Block 2, 4, 6, 8			
	a	b	a	b	a	b		
01	⊕ ⊕		1	URW00	UGND	1	URW00	UGND
02	⊕ ⊕		2	UGND	URW01	2	RES.	URW01
03	⊕ ⊕		3	URW02	URW03	3	URW02	URW03
04	⊕ ⊕		4	URW04	UP5V	4	URW04	UP5V
05	⊕ ⊕		5	UP5V	URW05	5	UP5V	URW05
06	⊕ ⊕		6	URW06	URW07	6	URW06	URW07
07	⊕ ⊕		7	URW08	UGND	7	URW08	UGND
08	⊕ ⊕		8	DSC	URW09	8	DSC	URW09
09	⊕ ⊕		9	URW10	UGND	9	URW10	RES.
10	⊕ ⊕		10	UGND	URW11	10	UGND	URW11
11	⊕ ⊕		11	URW12	UGND	11	URW12	RES.
12	⊕ ⊕		12	SINP	URW13	12	SINP	URW13
13	⊕ ⊕		13	URW14	SOUT	13	URW14	SOUT
14	⊕ ⊕		14	UGND	URW15	14	UGND	URW15
15	⊕ ⊕		15	PDCS	PDCF	15	PDCS	PDCF
16	⊕ ⊕		16	PGDS	PGRD	16	PGDS	PGRD
17	⊕ ⊕		17	FRAMn	SCLK	17	FRAMn	SCLK

Orientation marker: 127, 254, 254

# Basics: Utility Pogos

Each utility pogo block on the DUT Scale interface is connected via the utility cable to the corresponding connector on the wiring board.

Depending on the type of pogo block selected, 2-row or 4-row, 16 or 32 utility lines are available on each block



## Controlling Utility Lines

Utility lines configured as instrument in a *specification file*

```
spec utilityLineDemo {  
    // An action must first be declared  
    // before it can be defined  
    action selectDUT_Signal_1;  
    action selectDUT_Signal_2;  
    setup utility uti0 {  
        value= 1;  
        // Default state 1 is equivalent to  
        // setting the utility line to "on"  
        action setState selectDUT_Signal_1 {value=1;}  
        action setState selectDUT_Signal_2 {value=0;}  
    }  
}
```

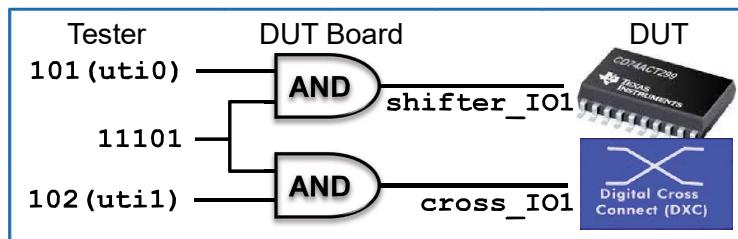
Utility lines as part of the routing in a *DUT board description file*

```
signal uti0 {  
    site 1 {pogo=101;}  
}  
signal util1 {  
    site 1 {pogo=102;}  
}  
signal DUT_Signal_1 {  
    site 1 {pogo=11101;}  
}  
signal DUT_Signal_2 {  
    site 1 {pogo=11101;}  
}
```

# Routing in the DUT Board Description File

Example:

A DUT board with routing and its description in the *DUT board description file*.



```
DutBoardwRouting.dbd
1 dutboard DutBoardwRouting {
2   sites = 1;
3
4   utilityLine uti0 { pogo=101; settlingTime=2ms; }
5   utilityLine util { pogo=102; settlingTime=2ms; }
6
7   signal shifter_IO1 {
8     site 1 { pogo=11101; }
9     routing { site 1 { uti0=1; util=0; } }
10  }
11  signal cross_IO1 {
12    site 1 { pogo=11101; }
13    routing { site 1 { uti0=0; util=1; } }
14  }
15 }
```

Keyword "utilityLine":

Definition of names, pogo pins and optionally settling times of utility lines

Keyword "routing": Settings of the utility lines needed to connect the corresponding signal.

# Routing and Specification Files

Example continued:

When running a *measurement* with a *specification file* that uses signal "shifter\_IO1", SmarTest will configure the utility lines according to the routing in the *DUT board description file*.

```
DutBoardwRouting.dbd
1 dutboard DutBoardwRouting {
2   sites = 1;
3
4   utilityLine uti0 { pogo=101; settlingTime=2ms; }
5   utilityLine util { pogo=102; settlingTime=2ms; }
6
7   signal shifter_IO1 {
8     site 1 { pogo=11101; }
9     routing { site 1 { uti0=1; util=0; } }
10  }
11  signal cross_IO1 {
12    site 1 { pogo=11101; }
13    routing { site 1 { uti0=0; util=1; } }
14  }
15 }
```

```
MainSpec.spec
1 spec MainSpec {
2   set timingSet_1; // declaration
3   set levelSet_1; // declaration
4   setup digInOut shifter_IO1 {
5     wavetable wvt1; // wavetable
6     set timing timingSet_1; // timing
7     set level levelSet_1; // level
8   }
9 }
```

Benefits:

- Users need not care about settings of utility lines.
- Changes to the routing settings only affect the *DUT board description file*.

*SmarTest will set "uti0" to 1 and "uti1" to 0.*

# Tables of the DUT Board Description File

The figure shows three windows related to the DUT Board Description File (DutBoardwRouting.dbd):

- Code Editor:** Shows the board description code with annotations for utility lines (utio, util) and signals (shifter\_IO1, cross\_IO1).
- Utility Configuration:** A table showing utility settings for Site 1. It includes columns for Utility, disabled, and Pogo.
- Routing Configuration:** A table showing routing settings for Site 1. It includes columns for Signal, Utility, and UtilityState.

The tabs “Utility” and “Routing” show the setup for utility lines and routing in two tables.

## Setup as Utility Instrument: Statically Controlled

The SmarTest *utility instrument* allows users to explicitly control utility lines.

The figure shows two windows illustrating the setup of utility lines as signals:

- DUT Board Description File:** Shows the board description code defining utility lines (utio, util) and signals (shifter\_IO1, cross\_IO1). Annotations indicate "The utility lines are defined as signals in the DUT board description file." and "No routing defined."
- Specification File:** Shows the UtilSpec.spec file containing setup commands for utility lines utio and util, along with declarations for timingSet\_1, levelSet\_1, and digInout shifter\_IO1. Annotations indicate "In the specification file, values are explicitly given. ‘0’ means high impedance state, ‘1’ means ‘on’."

# Setup of Dynamically Sequencer-Controlled Utility Lines

Sequencer-controlled utility lines require in hardware, that on the DUT board three PS1600 channels are connected to the utility block (pins 12a, 17a, and 17b):

These are used as digital SPI protocol channels (“sinp”, “fram”, “sclk”).

```
DutBoardwDynamicUtil >1
1 dutboard DutBoardwDynamicUtility {
2   sites = 1;
3
4   utilityControl {
5     SPI UT1, UT2 {
6       sinp = 10414;
7       fram = 10413;
8       sclk = 10406;
9     }
10
11   signal utio {
12     site 1 { pogo=101; }
13   }
14   signal util {
15     site 1 { pogo=102; }
16   }
17
18   signal shifter_I01 {
```

Additional “utilityControl” block:  
“UT1, UT2” denotes utility lines 101-132 and 201-232.

For setting up sequencer-controlled utility lines in SmarTest, add in the *DUT board description file* a “utilityControl” block to determine these PS1600 channels.

Switching dynamically controlled utility lines can be setup as actions and then switched during test execution.

For an example, see the next slide.

## Digital Tester Channels as Utility Instruments

A digital tester channel can be setup as a *utility instrument* as well.

```
DutBoardExplUtil.db >2 DutBoardDigChanUtil >4
1 dutboard DutBoardDigChanUtil {
2   sites = 1;
3
4   signal utio {
5     site 1 { pogo= 10301; settlingTime = 2ms; }
6   }
7   signal util {
8     site 1 { pogo= 10302; settlingTime = 2ms; }
9   }
```

Pogo of a digital channel

Settling times can be set for signals in the DUT board description file as well.

```
DigChanUtilSpec.spec >1
1 spec DigChanUtilSpec {
2   action selectShifter;
3   action selectCross;
4   setup utility utio {
5     value = 1; // force fixed voltage of 5V
6     action setState selectShifter { value = 1; }
7     action setState selectCross { value = 0; }
8   }
9   setup utility util f
10  value = 0; // high impedance
11  action setState selectShifter { value = 0; }
12  action setState selectCross { value = 1; }
13 }
14
15 set timingSet_1; // declaration
16 set levelSet_1; // declaration
17 setup digInOut shifter_I01 {}
```

Note:

For a digital channel

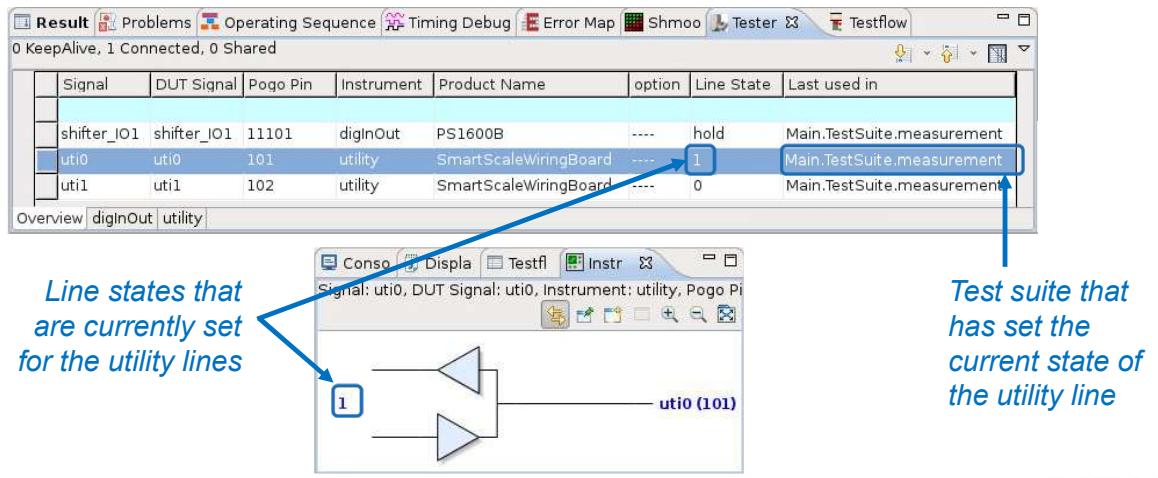
“0” means high impedance and

“1” means forcing a fixed voltage of 5V.

# Debugging Utility Lines

Tester view and instrument view show in debug mode

- the setting of a “utilityLine” defined in the *DUT board description file* and
- the setting of a signal used as a *utility instrument*.



## Summary - What you should have learned

- Utility lines allow to control or read the states of external devices such as relays, EEPROMS and indicators on your DUT board.
- Utility lines can be configured as routing in the DUT board description file or as instrument in the specification file.
- The physical location of utility lines depends on the test head size and the DUT scale interface in use.
- Sequencer-controlled utility lines and digital channels that are used as *utility instrument* allow to switch states during test execution at an exact certain point of time while sequencers are running.



# Licensing

## SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest Licensing - 1

## Learning Objective

- Understand the different operating modes of licensing
- Learn how to determine required licenses
- Learn how to generate a licensing file



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest Licensing - 2

# Agenda

- Overview of licensing in SmarTest 8
- Operating modes of licensing:
  - *fixed licensing mode*
  - *fixed feature licensing mode*
  - *dynamic licensing mode*
  - *mixed licensing mode*
- Checking required licenses: *licensing view*
- Setting up a file to control licensing

## Overview: SmarTest 8 Licenses

Running SmarTest 8 and the usage of V93000 test head card features are controlled by licenses. This capability is enabled by an industry standard licensing software, which is typically used to make with a license server licenses available on a network.

For more details on the licensing software, search for the key word “Licensing” in the TDC.

SmarTest 8 will always check out only the licenses needed – not more.  
So if the needed licenses are available, SmarTest 8 can be run without caring about licensing.

# Test Program Licensing File

The **test program licensing file** (with suffix .tplic) allows to control the license requirements of your test program.

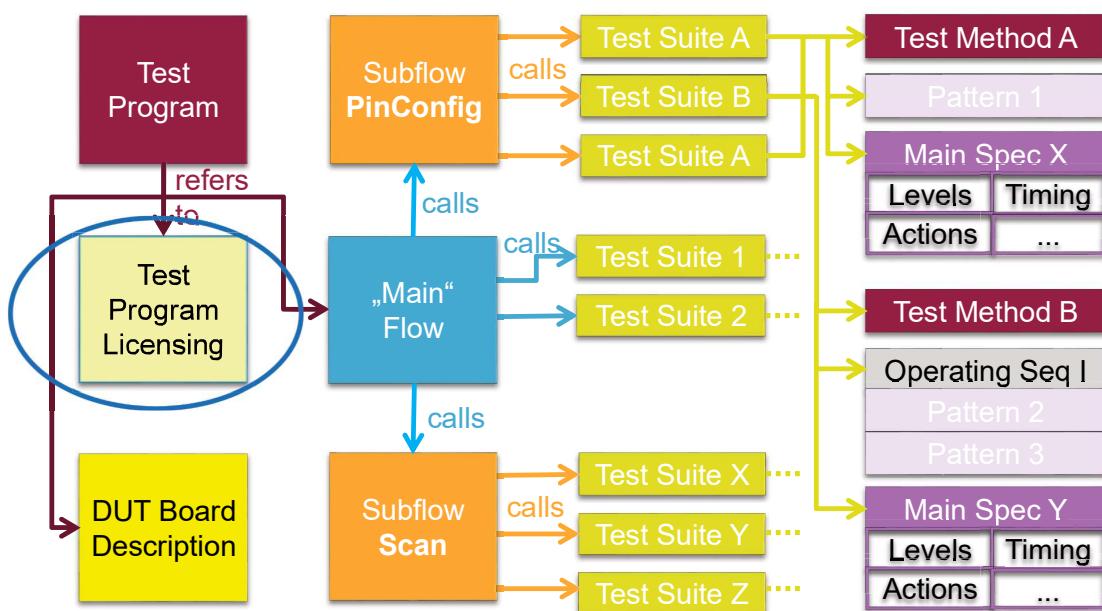
The **test program licensing file** is optional and it is referenced from the **test program file**.

- The **test program licensing file** specifies the licensed resources and features which are required for a test program - independent from the tester hardware.
- SmarTest requests licenses for these resources and features.  
If available, these licenses are checked out.

The **test program licensing file** is the only file type in a SmarTest 8 test project that contains data with respect to licensing.

With a **test program licensing file**, the required licenses are checked out in the background while the test program is activated, which minimizes the impact on the run time.

## Test Program with Licensing File



# Test Program Activation and Licenses

- Working with a test program in offline mode requires only a “SmarTest Offline” license.
- In online mode, if a *test program licensing file* is provided, then activating a test program will result in requesting the licenses for resources and features specified in the file.
- SmarTest might request additional licenses while execute tests online.
- Activating another test program releases any previously requested licenses.
- Shutting down SmarTest releases all licenses.

## Licensing Modes

- Dynamic
- Fixed
- Fixed feature
- Mixed

# Licensing Mode: Dynamic

The **dynamic licensing mode** is the default mode:

No *test program licensing file* is referenced from the *test program file*.

In this mode, license check-out typically happens “just-in-time” during test program execution:

- During test program execution, SmarTest detects the demand for licensed features and resources.
- Required licenses are checked out on demand from the license server.
- If SmarTest cannot obtain a license for the needed resources and features, then an error is thrown.

```
6@ testprogram CrossConnect {  
7    dutboard = crossconnect.common.CrossConnect4Site;  
8 //    feature licensing = crossconnect.common.mainFlow;
```

# Licensing Mode: Fixed

The **fixed licensing mode** requires a valid *test program licensing file*, that is referenced from the *test program file*.

In this mode, if a user activates a test program, then

- the referenced *test program licensing file* will be read and
- according to the resources and features listed in the file, SmarTest will request the relevant licenses that cover these resources and features.

The test program execution can only use resources and features enabled by the requested licenses.

Note:

- SmarTest does not request any additional licenses.
- If a test program execution requires more resources and features than checked out, SmarTest throws an error.

## Licensing Mode: Fixed – Setup Example

```
6 testprogram CrossConnect {  
7 dutboard = crossconnect.common.CrossConnect4Site;  
8 fixed licensing = crossconnect.common.mainFlow; ←  
9  
10 // optional auxiliary testflow to read and write test tables  
11 testflow PreBind {  
12   flow = crossconnect.common.PreBind;  
13 }  
14 // mandatory main testflow  
15 }
```

Reference to the test program licensing file.  
Only one file can be referenced per test program.

Keyword for fixed licensing mode

```
1 licensing mainFlow {  
2   features D00 {  
3     memory = 3813 B; // 3.724 kB  
4     speed = 1 Gbps;  
5   }  
6   features D01 {  
7     memory = 4437 B; // 4.333 kB  
8     speed = 1 Gbps;  
9   }  
10  features D02 {  
11    memory = 3501 R; // 3.419 kB
```

Test program licensing file referenced from the test program file above

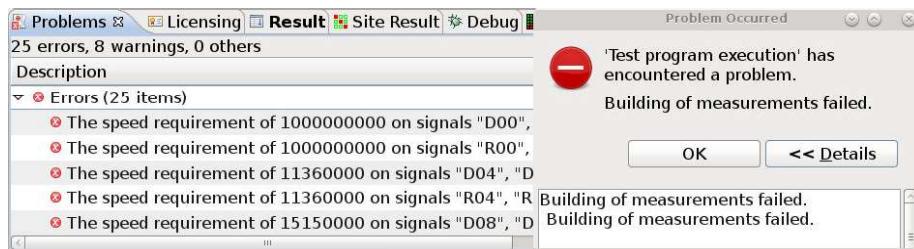
Requested resources are specified irrespective of any license steps

## Licensing Mode: Fixed – Error Example

Example of an error that is thrown because of licensing issues:

The test program execution requires more resources and features than enabled by the licenses checked out according to the *test program licensing file*.

These errors are thrown during the bind step when all parameters of the test program are known and measurements get bound to hardware.



# Licensing Mode: Fixed Feature

The **fixed feature licensing mode** is similar to the **fixed licensing mode**.

The difference is in the check of the needed resources and features:

- For **fixed feature licensing mode**, Smartest prevents the test program from using more resources and features than specified in the licensing file.
- For **fixed licensing mode**, Smartest prevents the test program from using more resources and features than covered by the minimum licenses checked out to address the needs of the licensing file (a super set).

Example: The licensing file specifies a memory size of 20MB for a certain signal, so that the memory license of 32MB is checked out.

If the test program needs 30MB for the signal, then in **fixed feature licensing mode** an error will be thrown (>20MB), but not in **fixed licensing mode** (<32MB).

```
6 testprogram CrossConnect {  
7   dutboard = crossconnect.common.CrossConnect4Site;  
8   fixed feature licensing = crossconnect.common.mainFlow;
```

# Licensing Mode: Mixed

The **mixed licensing mode** is a combination of

- the **fixed licensing mode** and
- the **dynamic licensing mode**.

It requires a valid **test program licensing file**.

In this mode, if a user activates a test program and afterwards starts executions, then:

- the referenced **test program licensing file** will be read;
- according to the resources and features listed in the file, SmarTest will request the relevant licenses that cover these resources and features;
- if during a test program execution the demand for additional licenses is detected, then SmarTest requests these from the license server.

```
6 testprogram CrossConnect {  
7   dutboard = crossconnect.common.CrossConnect4Site;  
8   licensing = crossconnect.common.mainFlow;
```

# Required Licenses - Licensing View

The **licensing view** uses multiple tabs to show

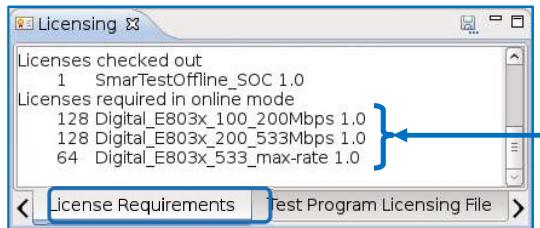
- the required features and resources and
- the licenses checked out.

The results are accumulated over the executions of the activated test program and of its testflows.

To release the licenses and to reset the licensing results, activate the test program again.

Example:

Use the tab “License Requirements” to check required licenses for a test program which ran in offline mode.

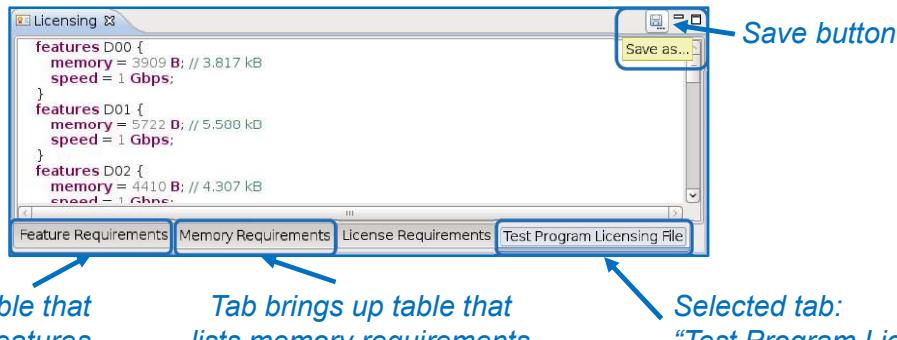


In offline mode, the tool shows the licenses that would be needed for the online mode as well.

## Generation of a Test Program Licensing File

The “Test Program Licensing File” tab of the *licensing view* displays the required features and resources in a format that is similar to a *test program licensing file*.

The “save” button in the *licensing view* allows to write the content shown in the “Test Program Licensing File” tab to a file. The user will be prompted to provide a name.  
The file can be used as a *test program licensing file*. It can be edited if needed.



Tab brings up table that lists required features

Tab brings up table that lists memory requirements

Selected tab:  
“Test Program Licensing File”

# Changes to Licenses

In general, licenses purchased for a tester for an earlier version of SmarTest are also valid for SmarTest 8.

The most important changes affect memory licenses:

Memory licenses include now only the memory required for vectors, not for the result memory:

- Existing memory license depth is the vector memory depth.
- Remaining physical channel memory is freely available for result memory.

Furthermore, a special license for protocol aware software is not required in SmarTest 8.

## Summary - What you should have learned

- A *test program file* can reference a *test program licensing file*, which determines the operation mode for licensing.
- Four operation modes are provided for licensing:
  - *Dynamic licensing mode*: No *test program licensing file* is read. Licenses are requested during executions. Default mode if no *test program licensing file* is set.
  - *Fixed licensing mode*: A *test program licensing file* is read and licenses are requested during test program activation. Additional licenses required during an execution will result in an error.
  - *Fixed feature licensing mode*: A *test program licensing file* is read and licenses are requested during test program activation. More features or resources required during an execution will result in an error.
  - *Mixed licensing mode*: A *test program licensing file* is read and licenses are requested during test program activation. Additional licenses might be requested during test program executions.
- The *licensing view* shows the required features and resources and the licenses checked out. It can generate *test program licensing files*.



# Test Cell Control Tool

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest Patterns - 1

## Learning Objectives

- Understand the intended use of TCCT for lot processing
- Understand the recipe capabilities and use model



January 2020

All Rights Reserved - ADVANTEST CORPORATION



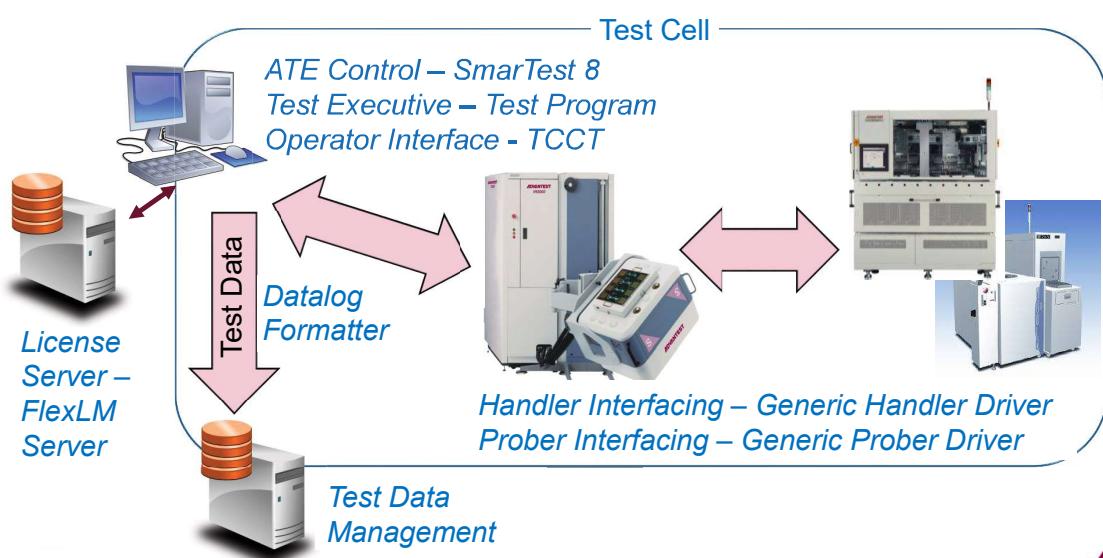
SmarTest: Test Cell Control Tool - 2

# Agenda

- Installation and start of the Test Cell Control Tool (TCCT)
- Processing engineering lots with the Test Cell Control Tool
- Datalog control and running recipes

## SmarTest 8 Test Cell Software Components

The term “Test Cell” refers not only to the ATE system itself, but also all of the “components” that need to be combined to the system in order to create a complete production system.



# Test Cell Access Package

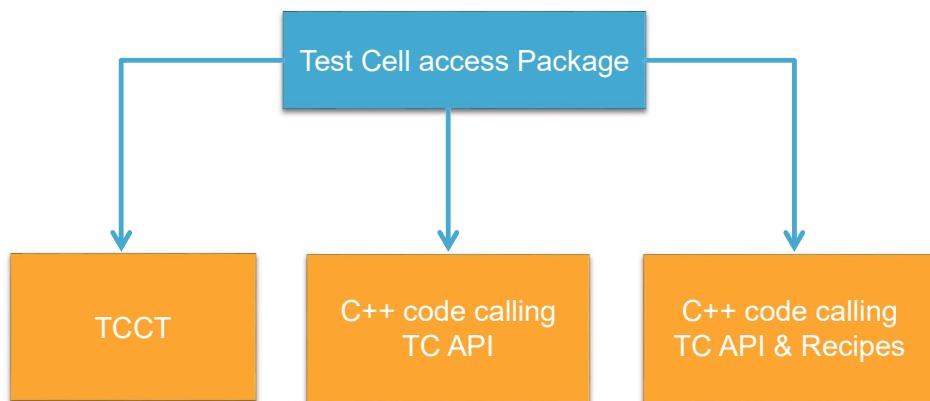
The Test Cell Access and Control package is part of the SmarTest software suite and shares the same revision number as the SmarTest core package.

Essentially it contains 3 components:

- Test Cell Control API (TC-API),
  - Prober/Handler Control (PH-Control),
  - Test Cell Control Tool (TCCT).
- **TC-API:** Set of easy-to-use programming interfaces to control the tester and the prober/(handler equipment.
  - **PH-Control:** Application to load and execute the V93000 Generic Prober/Handler drivers and thus to provide access to the Prober/Handler equipment.
  - **TCCT:** Graphical user interface built on the TC-API that allows to run the device tests for both, engineering lots and production lots.

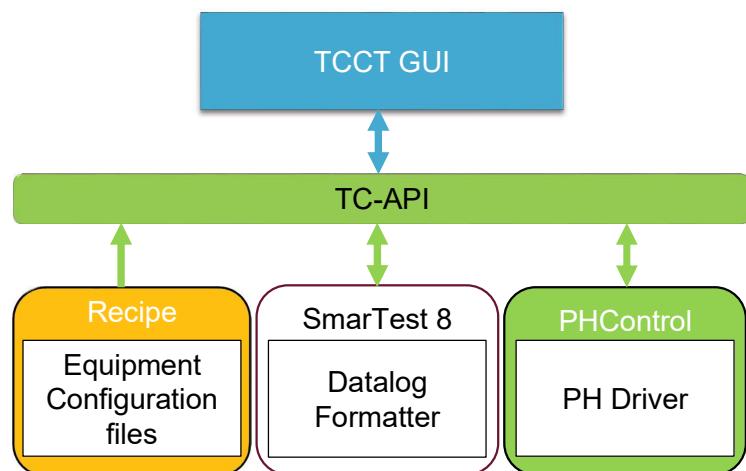
## Control of Production Test Cell

The Test Cell Access Package enables three different ways to control the production test cell environment:



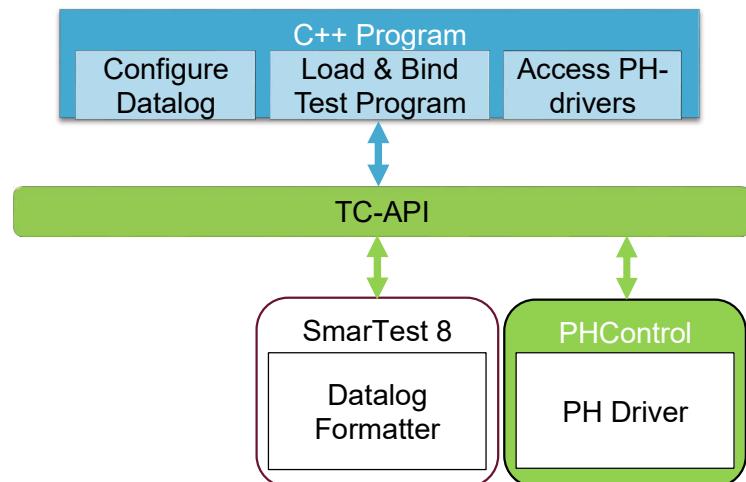
## Control via TCCT

- Select the test program..
- Access recipe file and equipment configurations.
- Configure datalog settings.
- Load & bind test programs.
- Access and configure prober and handler equipment.
- Execute test programs.



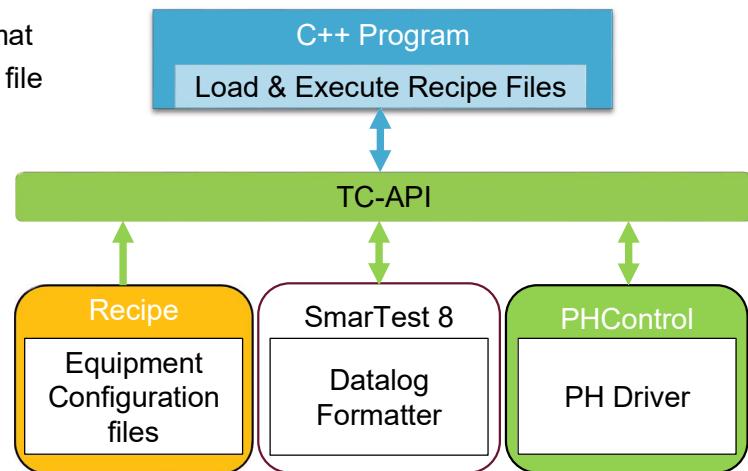
## Control via TC-API Program

- Attach to an active SmarTest session.
- Activate, load, bind, run and reload test programs and access test program variables.
- Control the execution of testflows.
- Set up and control the generation of datalog files.
- Monitor testflow execution.
- Set up and access prober and handler equipment via the “PHControl” class.



## Control via TC-API Program & Recipe

- C++ program calls a recipe file in XML format
- Similar to the Test Cell Control Tool, recipe file can:
  - Access and configure prober/handler equipment.
  - Configure datalog settings.
  - Load and bind test programs.
  - Execute test programs.



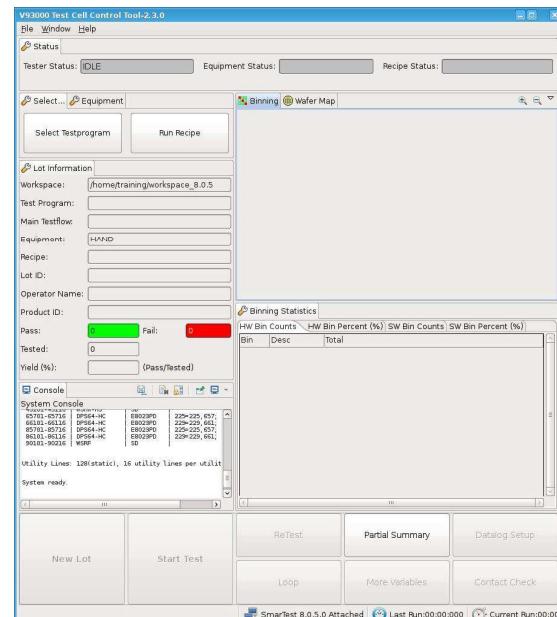
## Production Operator Interface: TCCT

Test cell operator interface

- **Interactive execution:** Test program centric operation (typically “hand” test)
- **Production mode:** Recipe centric operation
- Interactive datalog control
- Result displays
- Customizable layout
- Seamless integration with recipes

Comparison to SmarTest 7:

Replaces the application model and work-order manager



# Production Test Data: Formatters & TCCT

The main datalog source of SmarTest 8 is the event data log, version 2, named “EDL2”. It contains all raw test-event data.

SmarTest 8 provides various data log formatters converting EDL2 test data:

- STDF: Standard data log formatter that converts data to STDF.
- ASCII: Converts EDL2 test data to text data log, similar to the output in the data log console view.
- SUMMARY: A summary of the EDL2 test data in text form.

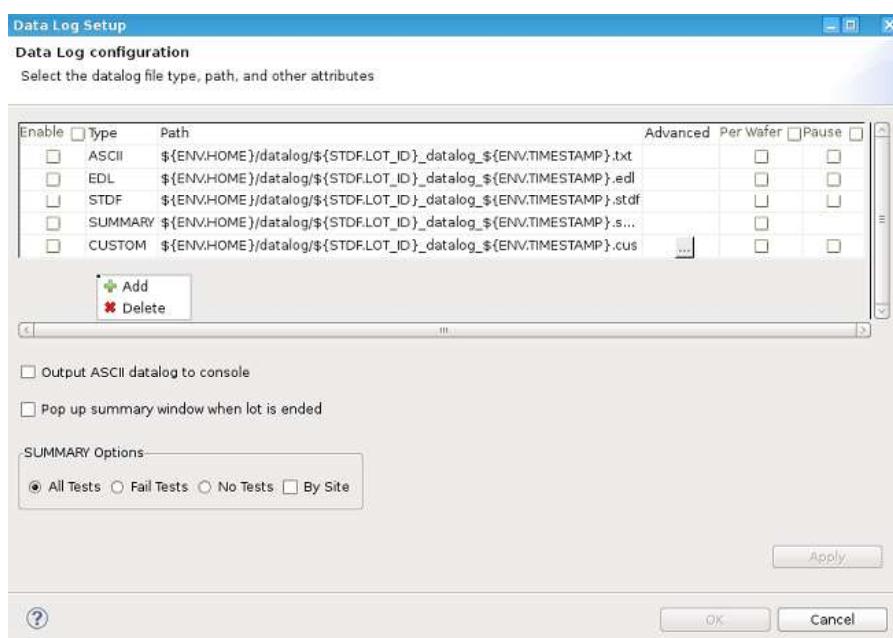
Additionally, the TCCT supports custom data formats:

- CUSTOM: Data log formatter provided by the user.

The TCCT allows to enable/disable/configure these data log formatters

- in an interactive way or
- based on recipes.

# Production Test Data: Formatters & TCCT



# Starting the Test Cell Control Tool (TCCT)

The following command starts the TCCT in engineering mode:

```
/opt/hp93000/testcell/bin/tcct
```

The following command starts the TCCT in production mode:

```
/opt/hp93000/testcell/bin/tcct.prod
```

- If no SmarTest session is running, TCCT will attach to the next SmarTest session that is started.
- If a single SmarTest session is running, TCCT will attach to that.
- If more than one SmarTest session is running, a dialog will pop up asking to select the session to work with.

# Configuring the Layout of the TCCT

The layout of the Test Cell Control Tool window is controlled by an XML configuration file.

When TCCT starts, it reads for layout configuration the file

```
/etc/opt/hp93000/testcell/tcct_gui_configuration.xml
```

If the file is not available, the default TCCT layout is used.

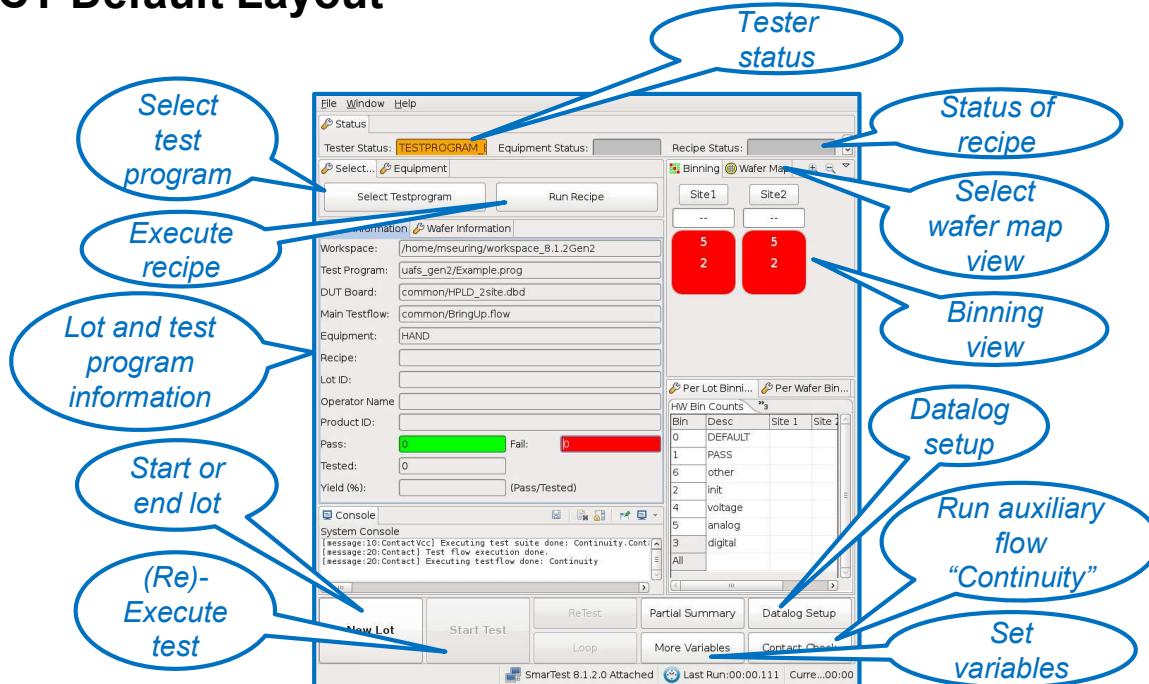
When starting TCCT, the configuration file can be selected with the “-u” option.

For production mode, a customized path to the layout configuration file can be set in the setup file for the production environment:

Root user authority is needed to specify the configuration file in the setup file

```
/etc/opt/hp93000/testcell/production_environment_config.xml
```

# TCCT Default Layout



## TCCT and Recipes

To load a recipe when starting TCCT use the “-r” option:

```
/opt/hp93000/testcell/bin/tcct -r my_recipe.xml
/opt/hp93000/testcell/bin/tcct.prod -r my_recipe.xml
```

If it is specified in the recipe file, the TCCT will start SmarTest.

The environment variable “TCCT\_RECIPE\_FILE\_LOCATION” specifies a folder containing recipe files. Set it on the command line as follows:

```
export TCCT_RECIPE_FILE_LOCATION=/home/demo/my_recipes
```

If you start TCCT in the production mode, then a pop-up dialog appears that allows to select a recipe file from that folder.

# Configuration of the TCCT in Production Mode

Settings for the in production mode can be customized in this file

`/etc/opt/hp93000/testcell/production_environment_config.xml`

This file configures:

- Layout configuration file of the Test Cell Control Tool
- Folder containing recipes
- SmarTest startup mode
- Model file
- Settings of environment variables

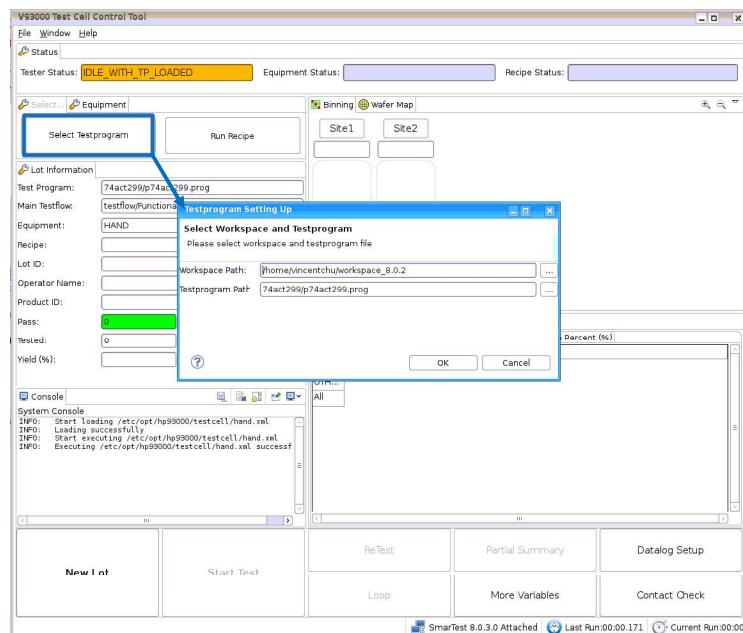
```
*production_environment_config.xml <?
1<?xml version="1.0" encoding="UTF-8"?>
2<production>
3  <smartest>
4    <startup_mode>offline</startup_mode>
5    <show_SWC>false</show_SWC>
6    <model_file>/etc/opt/hp93000/soc_common/model</model_file>
7    <env>
8      <!-- here can specify the environment variable for smartest using, for example-->
9      <!-- <LD_LIBRARY_PATH>/path/or(paths/to/3rd_party_libs</LD_LIBRARY_PATH> -->
10     </env>
11   </smartest>
12   <tcct>
13     <ui_config_file>/etc/opt/hp93000/testcell/tcct_gui_configuration.xml</ui_config_file>
14     <recipe_file_location>/tmp</recipe_file_location>
15   </tcct>
16</production>
17|
```

## Select Test Program

Before selecting a test program, users have to select the workspace which contains the test program.

After selecting the *test program file* and confirming with “OK”, the setup data of the test program is activated and bound.

Then the path of the *test program file* and the name of the main testflow are shown.



# Running an Engineering Lot

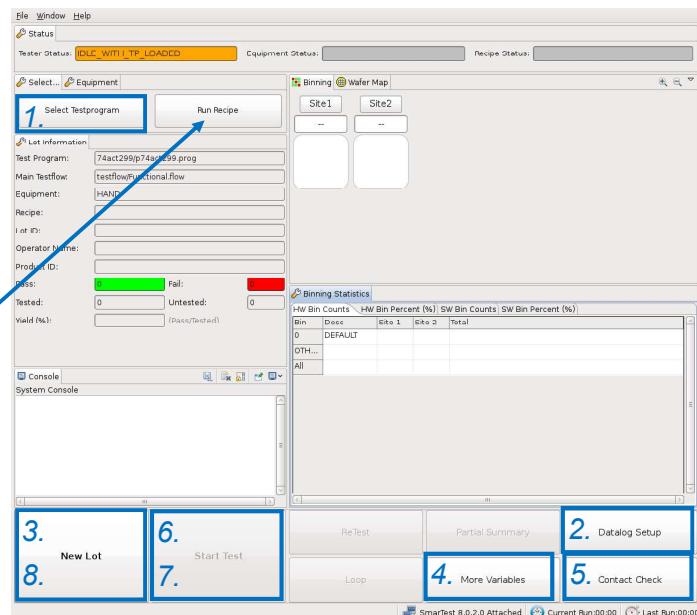
Steps to execute a test program:

1. Select a test program as shown on the previous slide.
2. Use the "Datalog Setup" button to customize data logging.
3. The "New Lot" button brings up the "Lot Information" dialog. Enter the settings of the new lot. Afterwards, "Start Test" is no longer greyed out.
4. "More Variables" opens the "Datalog Variables Setting" window.
  - Two tabs: "Production Variables" and "Testprogram Variables"
  - Change value of a variable via double-click.
  - Right-click allows to add or delete variables.
5. Select "Contact Check" to execute the auxiliary flow "Continuity".
6. The "Start Test" button executes the test program.  
Data log files will be generated as specified.  
Yield, binning information and statistics are shown after execution.
7. Use "Start Test" again to test the next device.
8. When all the devices of a lot are tested, then press the "End Lot" button.

## Summary: Running an Engineering Lot

1. Select test program.
2. Customize data logging.
3. Start new lot.
4. If needed set variables.
5. Optional: Contact check.
6. Start test execution.
7. Repeat (5.) as needed for other devices.
8. When done, end lot.

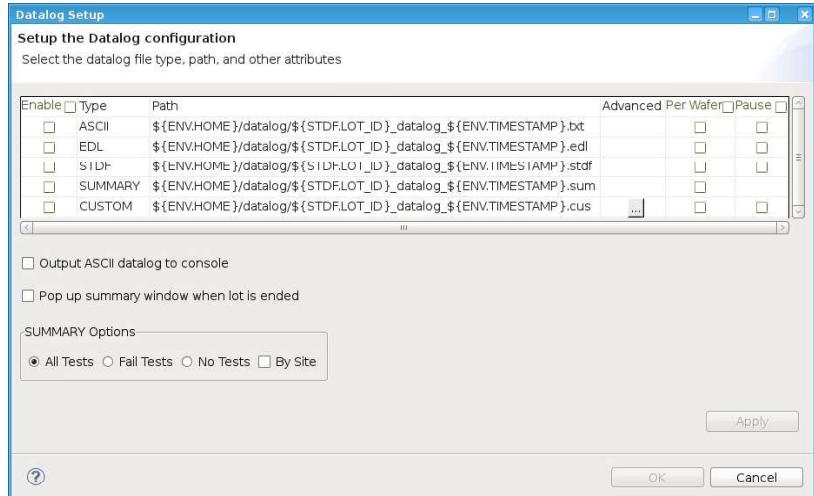
To execute a recipe, press the "Run Recipe" button and select the recipe file.



# Datalog Setup Window: Details

The “Datalog Setup” button invokes the “Datalog Setup” dialog window

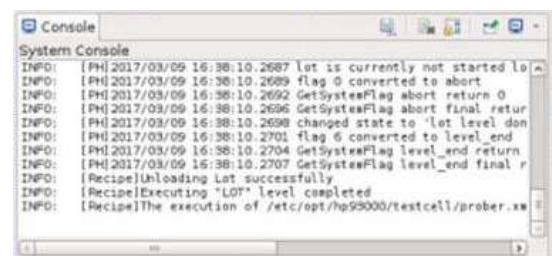
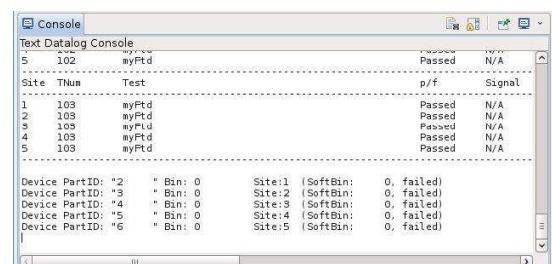
- Datalogs can be added and removed by right-click.
- The “Enable” checkbox enables a specific datalog.
- The “Pause” checkbox is used to pause or resume.
- The “Advanced” button allows to configure a custom formatter.



# TCCT GUI (after execution)

The *console* view can show two consoles:

- **Text Datalog Console** Shows the content of the text data log file in the console. The Output ASCII datalog to console option in the Datalog Setup dialog is checked by default to enable the console output. The source file for this console is located in `/opt/hp93000/testcell/log/`. DefaultAsciiDatalogFile.
- **System Console** Shows messages from SmarTest UI reports, prober or handler driver outputs, recipe execution outputs, and other test cell clients' outputs.



# Summary - What you should have learned

- Typically the test cell software consists of SmarTest, a Test Cell Control Tool, a test program, generic prober or handler driver, data log formatter, and license server.
- The Test Cell Access and Control package is part of the SmarTest software suite and contains three components:
  - Test Cell Control API (TC-API),
  - Prober/Handler Control (PH-Control),
  - Test Cell Control Tool (TCCT).
- The Test Cell Control Tool is an example of a production operator interface and allows to execute test programs in a graphical user interface for both, engineering lots and production lots.
- The “Datalog Setup” dialog of the Test Cell Control Tool lets you flexible customize the datalog output.



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest: Test Cell Control Tool - 23



# Test Program Migration Framework

SmarTest 8.2.5 Training

January 2020



January 2020

All Rights Reserved - ADVANTEST CORPORATION



SmarTest 8 Test Program Migration Framework - 1

# Learning Objective

- Learn how to use the *test program migration framework* to convert setup data of a SmarTest 7 test flow to SmarTest 8

# Agenda

- Introduction: Scope of the *test program migration framework*
- Invocation and setup of the *test program migration framework*
- Conversion modes:
  - Conversion of all setup data of a complete testflow
  - Modular conversion of all setup data of all test suite groups
  - Modular conversion of all setup data of selected test suite groups
- Running the conversion and results

# Test Program Migration Framework: Scope

The scope of the *test program migration framework* is to help users to convert an existing SmarTest 7 test program to SmarTest 8.

As it is not possible to map all features of SmarTest 7 to SmarTest 8, there inherently exist some limitations for a conversion based on a tool:

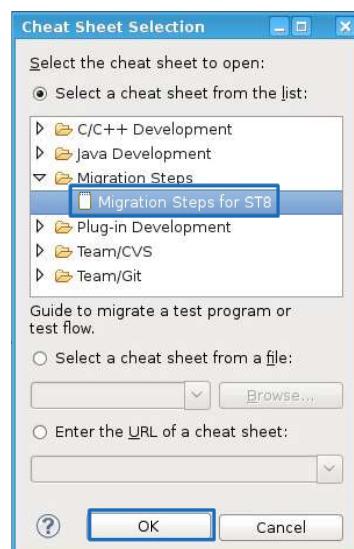
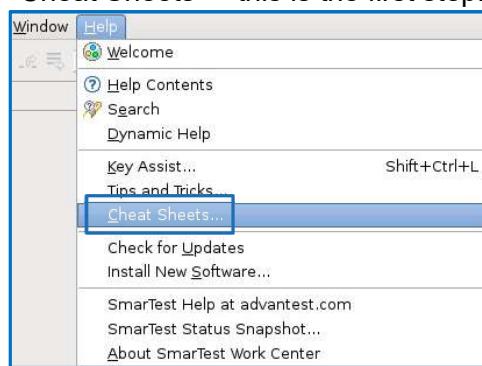
- Test methods are not parsed and translated.
- Not all SmarTest 7 features are supported.

The conversion of a SmarTest 7 test program consists of three steps:

1. Check and understand the SmarTest 7 test suites.
2. Use the *test program migration framework* to convert setups.
3. Utilize the converted setup files as starting point to implement the building blocks of the SmarTest 8 test suites.

## Invocation with Cheat Sheets

The *test program migration framework* can be started via the tool bar entry “Help” and “Cheat Sheets” – this is the first step:



Second step: Start the *test program migration framework* from the “Cheat Sheet Selection” window.

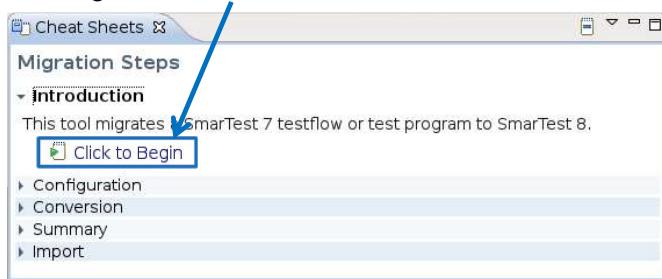
## Usage: Begin

Alternatively, the *test program migration framework* can also be started via the "93000" menu item "Device → Migrate Device".



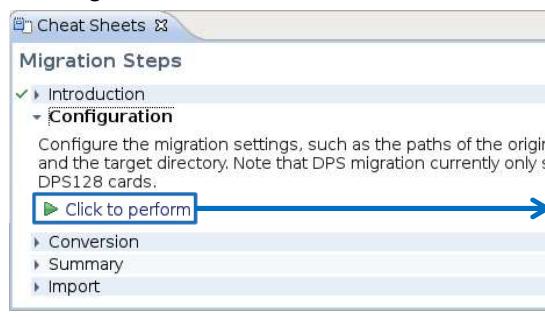
Once the *test program migration framework* has been started, a new window showing the migration steps will come up.

To begin click here.



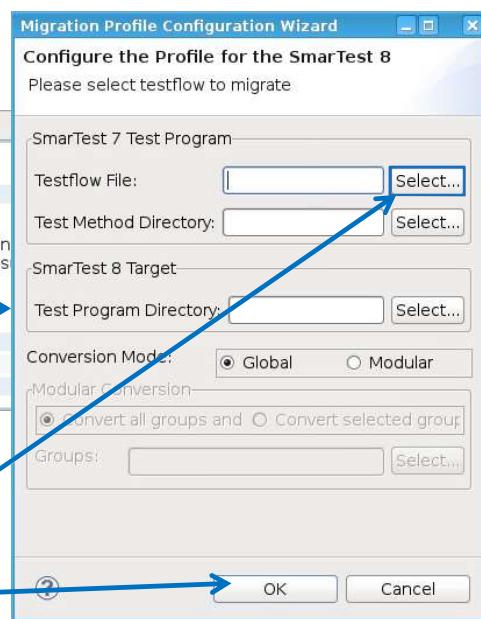
## Usage: Configuration

The next step is to configure the settings. Click "Click to perform" to invoke the "Configuration Wizard".



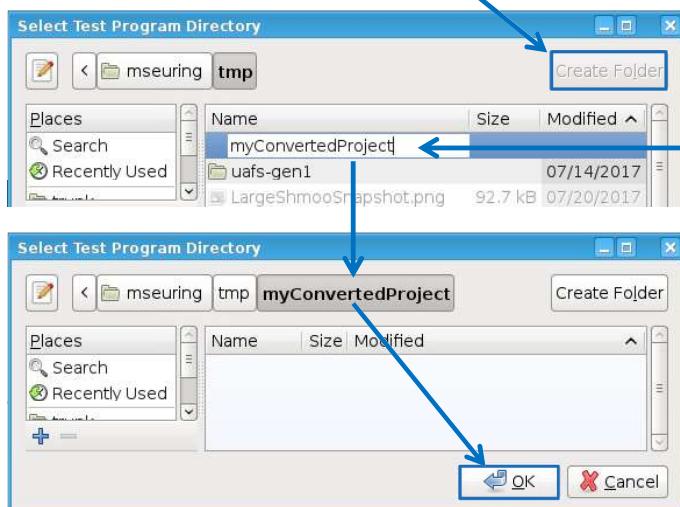
The configuration wizard facilitates the setup of the framework.

First you should select a test flow file of the SmarTest 7 test program to be converted.



# Usage: Selecting the SmarTest 8 Project Folder

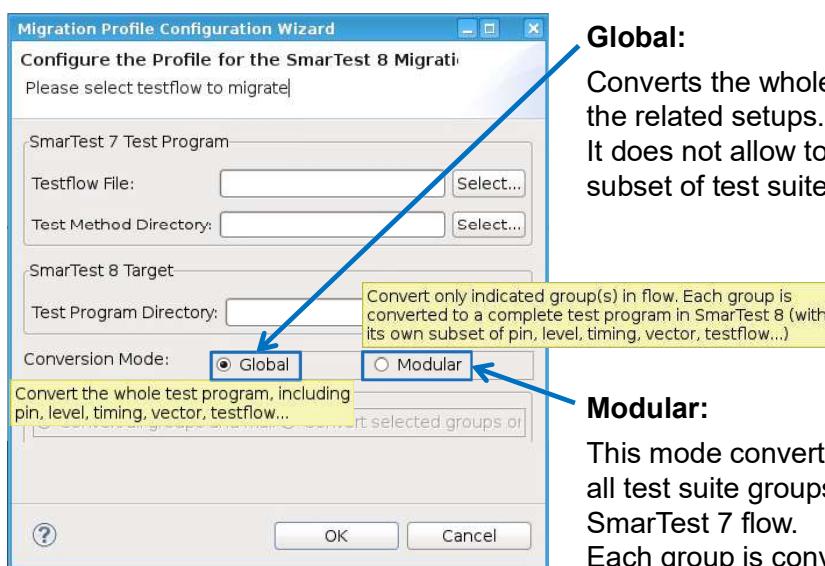
The next step is to select a folder for the SmarTest 8 test program. It is recommended to create a new folder for that.



Once you have selected a name for the new folder, do not miss to hit "return".

After selecting a name for the new folder for the SmarTest 8 test program, make sure it is shown before confirming with "OK".

## Conversion Modes: Global and Modular



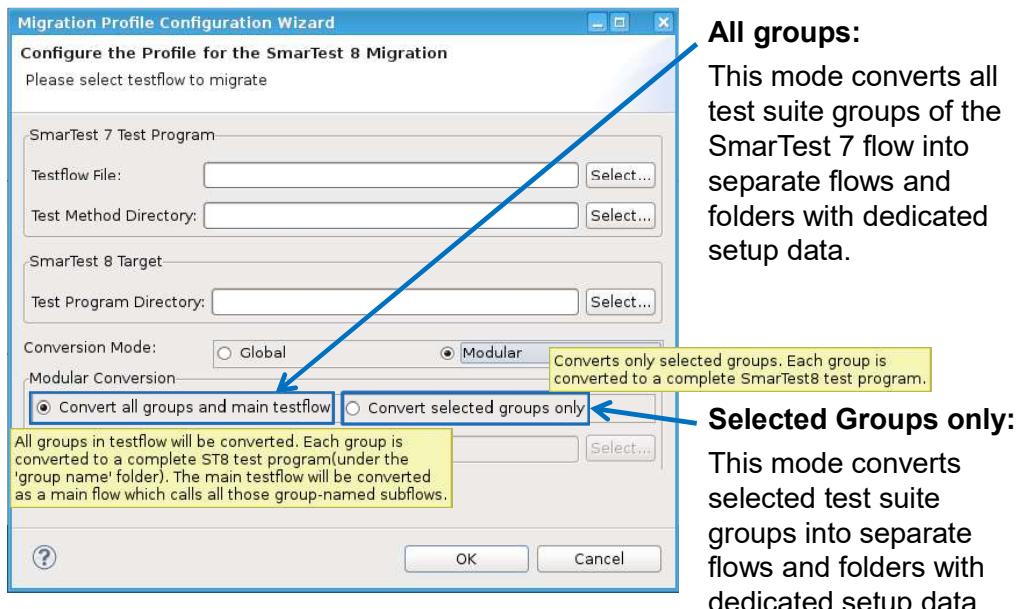
### Global:

Converts the whole test flow and the related setups.  
It does not allow to select only a subset of test suites.

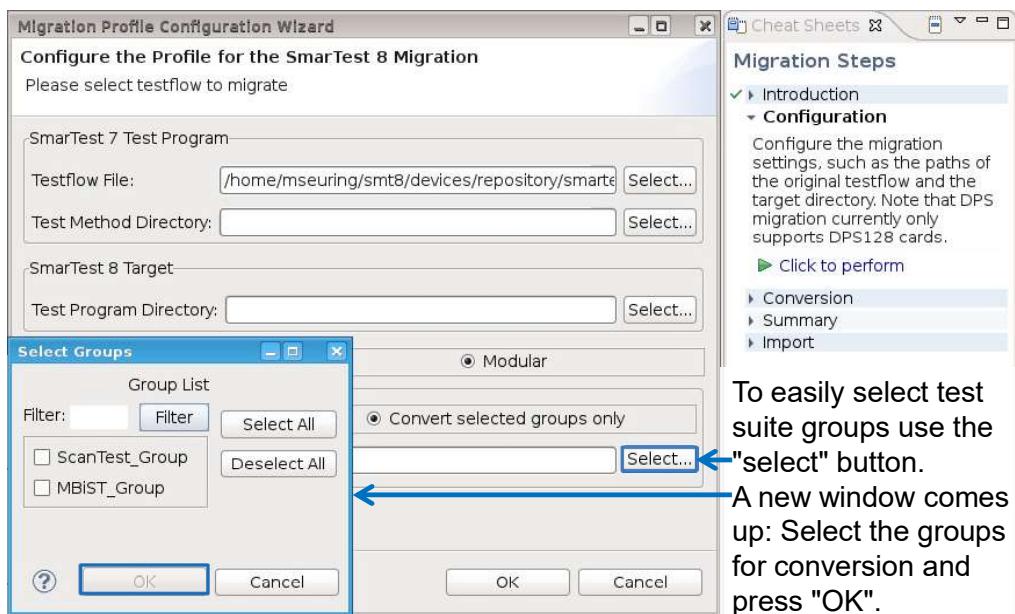
### Modular:

This mode converts selected or all test suite groups of the SmarTest 7 flow.  
Each group is converted in a separate flow with dedicated setup data.

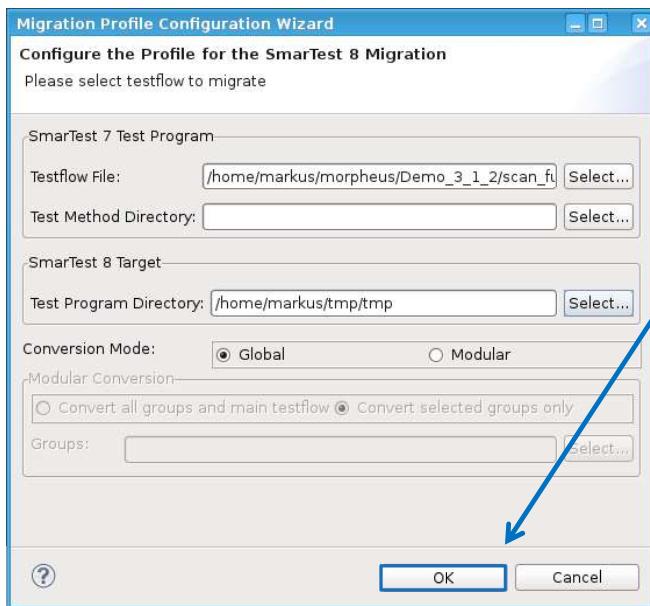
# Details of Modular Conversion



## Conversion Mode: Modular – Select Groups



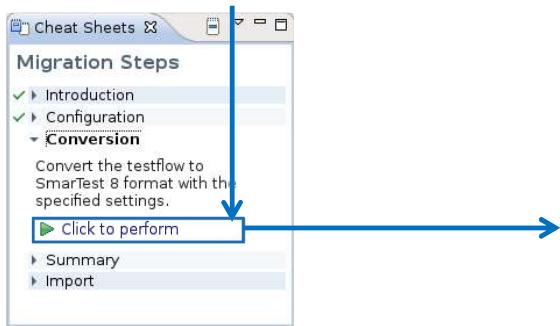
# Usage: Configuration Settings Done



Once all entries in the configuration wizard are correctly set, press "OK".

# Usage: Execution of Conversion

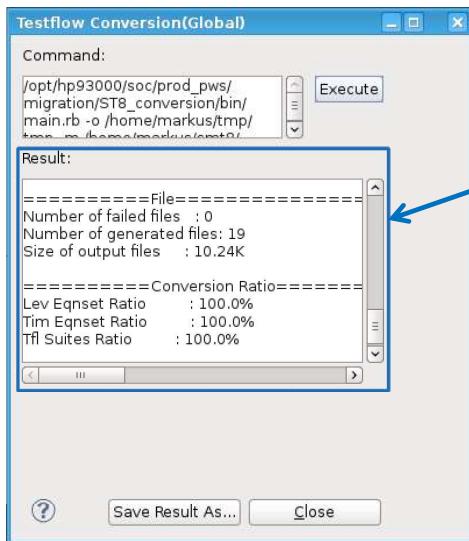
After configuration is done, the conversion can be started here.



In a new window the conversion is started as shown on the right side.

"Command:" shows how to run the same conversion from the command line. To trigger conversion, press "Execute".

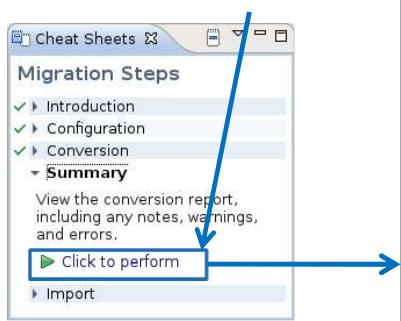
## Usage: Execution Results



After the conversion was triggered and has finished, results are shown.

## Usage: Execution Results in the Browser

The summarized results of the conversion can be reviewed in an HTML file. This is triggered here.



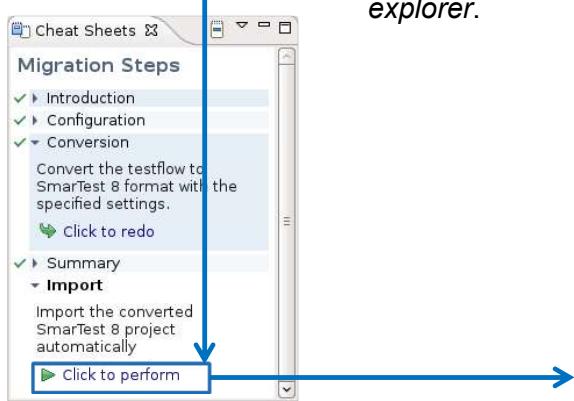
The screenshot shows a browser window titled 'TaCoST8 Conversion Summary Report'. The URL in the address bar is 'file:///home/markus/tmp/tmp/ConvertStatusSum/'. The page content is titled 'Conversion Result Summary' and includes the following information:

Conversion Info	
Conversion command	<pre>/opt/hp93000/soc/prod_pws/migration/ST8_conversion/bin/main.rb -o /home/markus/tmp/tmp -m /home/markus/osram /osram_led_testprograms/trunk/uafs-gen2/offline.model /home/markus/morpheus/Demo_3_1_2 /scan_full_spec_dev/testflow /shifter_scan_full_spec.tf</pre>
Source	<ul style="list-style-type: none"><li>/home/markus/morpheus/Demo_3_1_2 /scan_full_spec_dev/testflow /shifter_scan_full_spec.tf</li></ul>
Target	<ul style="list-style-type: none"><li>/home/markus/tmp/tmp</li></ul>
Target test program	<ul style="list-style-type: none"><li>/home/markus/tmp/tmp</li></ul>

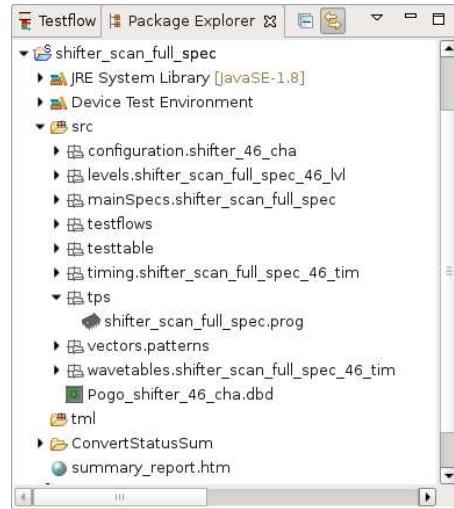
Here the HTML result file is shown in the browser of the SmarTest Work Center.

# Usage: Import Converted Data

The new SmarTest 8 project generated from the converted data can be easily imported by clicking here.



Finally, the new SmarTest 8 project with the converted data is imported as shown here in the *package explorer*.



## Summary - What you should have learned

- The *test program migration framework* allows to convert setups of a SmarTest 7 test program to SmarTest 8.
- You can work with the tool interactively in the *SmarTest Work Center* or use the commandline.
- To work interactively, select “93000” > “Device” > “Migrate Device” in the *SmarTest Work Center*.
- You can select between the following Conversion modes:
  - Conversion of all setup data of a complete testflow.
  - Modular conversion of all setup data of all test suite groups.
  - Modular conversion of all setup data of selected test suite groups.
- After conversion, you can import the converted files to the SmarTest Work Center.



# STIL Reader: Basic conversion of STIL Files

SmarTest 8.2.5 Training

January 2020



January 2020

All rights reserved – ADVANTEST CORPORATION

**ADVANTEST**

STIL Reader: Basic conversion of STIL Files- 1

## Learning Objective

- Understand how to convert STIL files to SmarTest 8 setups



January 2020

All rights reserved – ADVANTEST CORPORATION

**ADVANTEST**

STIL Reader: Basic conversion of STIL Files- 2

# Agenda

- Introduction and scope
- Minimum requirements for a STIL file
- Concept of converting STIL format to 93000 SmarTest setups
- Conversion procedure
- Verify the conversion output using SmarTest

## Introduction and Scope

STIL (Standard Test Interface Language):

- Widespread format that is used for cyclized data like scan test data.
- Allows to describe signals/signal groups, ports, channels, levels, wavetables, waveforms, patterns, etc.

This presentation describes the conversion of STIL files using “`stilreader`” or “`stilreader-gui`” of “SmarTest Data Link” (STDL).

The software package STDL is a solution for converting test data from formats used in semiconductor design to the SmarTest 8 format.

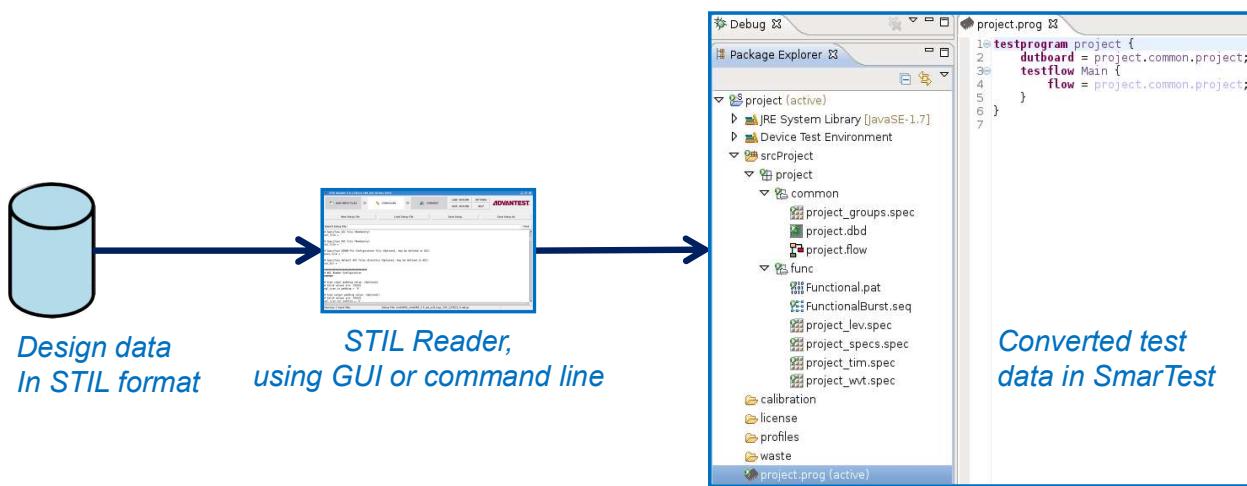
Therefore, it can also convert test data files in TDL and WGL format.

# STIL Files: Minimal Requirements

For conversion of a digital pattern, a STIL file needs to contain:

- Signal list with names and directions.
- Timing information for each signal or group of signals.
- Pattern information for each signal or group of signals.

## From STIL Format to SmarTest Setups



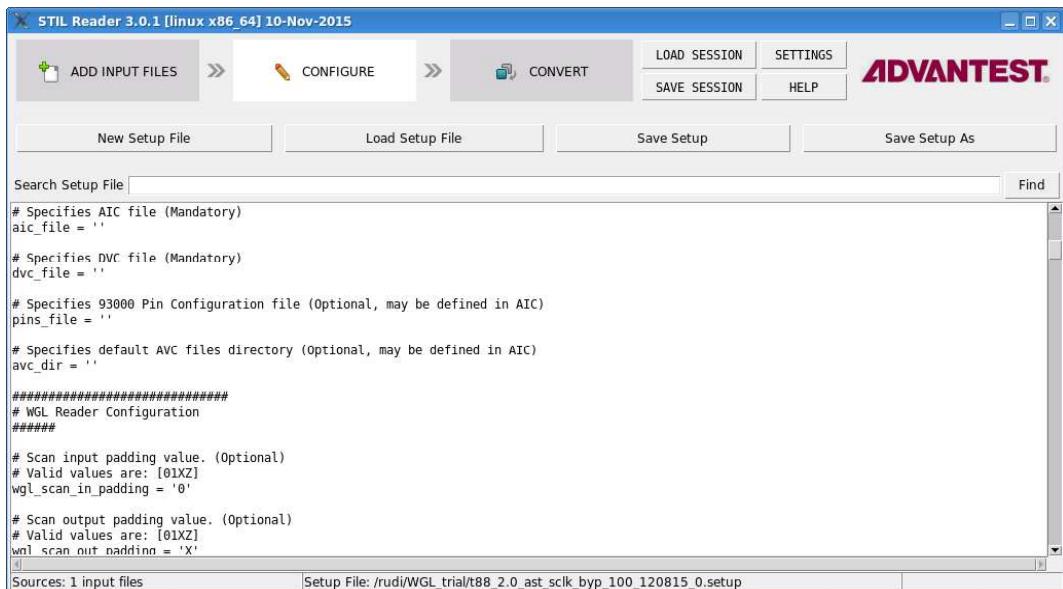
# Conversion Procedure

1. Start the graphical user interface:  
`/opt/93000/stdl/bin/stilreader_gui &`
2. Select “ADD INPUT FILES” to select STIL file to be converted.
3. Select “CONFIGURE” to set options:  
“New Setup File” or “Load Setup File”.
4. Modify and “Save Setup File”.
5. Select “Convert”.
6. Click “Run”.

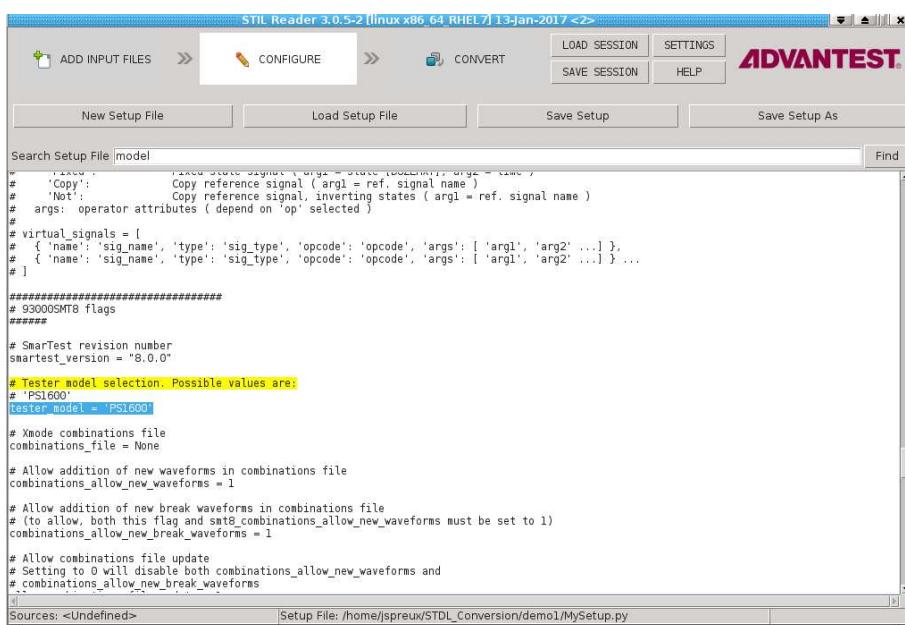
## “stilreader” Installation and Execution

- “stilreader” is delivered and installed as part of the “STDL” RPM
- “stilreader” requires a license that must be entered in your system license file.
- Installation via command line:  
`rpm -ivh <package_name>`
- Installation folder:  
`/opt/93000/stdl/bin`
- Start the command line tool:  
`/opt/93000/stdl/bin/stilreader`
- Start the graphical tool:  
`/opt/93000/stdl/bin/stilreader-gui`

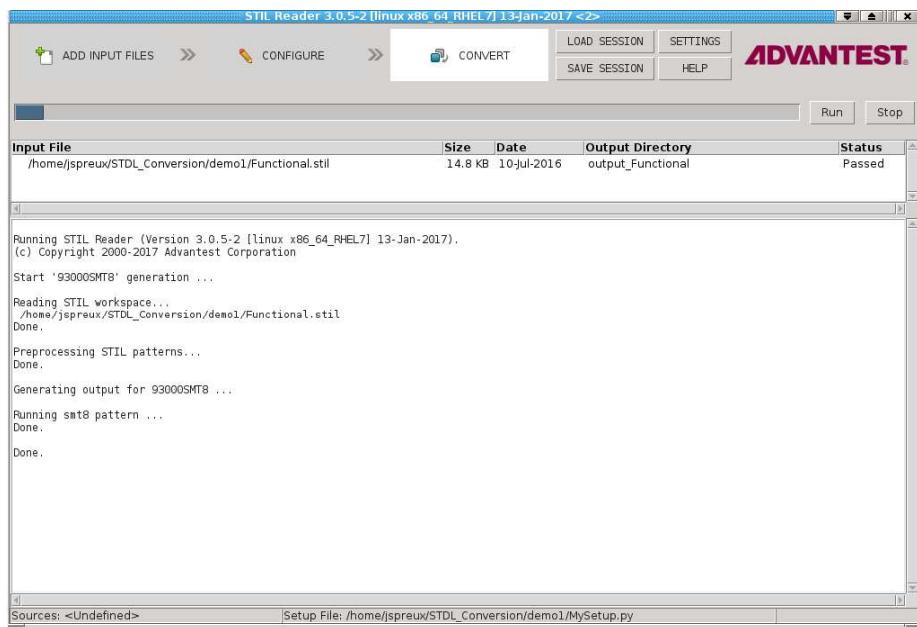
# The “stilreader” Graphical User Interface (GUI)



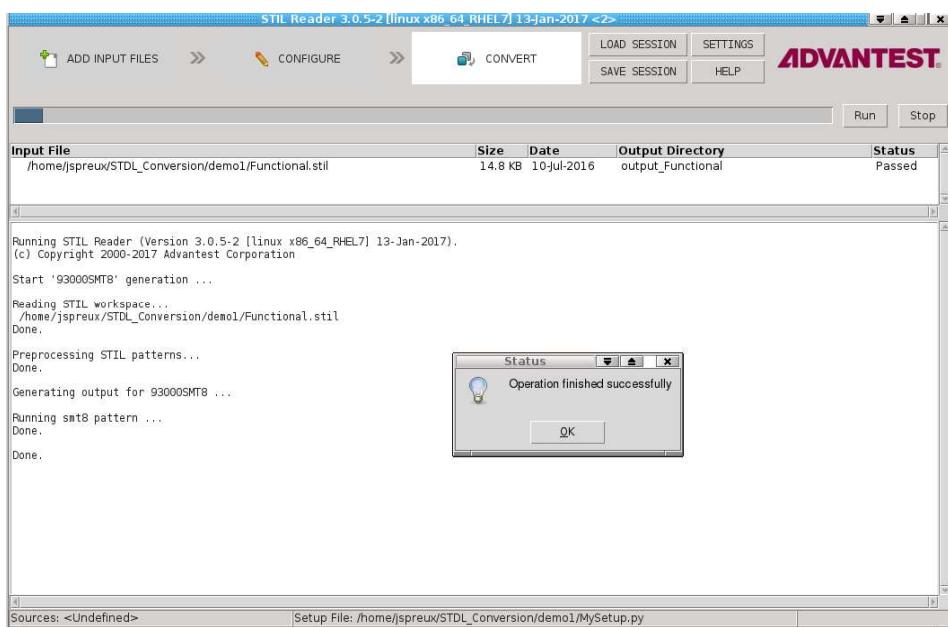
## Edit Configuration File



# Trigger Conversion

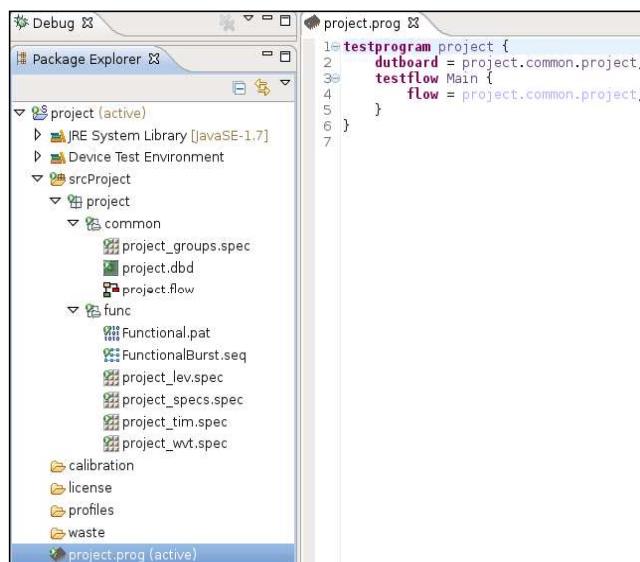


# Verify Conversion Status



# Viewing Conversion Result

To review the results of the conversion, import project into the *SmarTest WorkCenter*.



## Summary - What you should have learned

- To convert STIL files use the tools “**stilreader**” or “**stilreader-gui**” of the “SmarTest Data Link” (STDL) package.
- You can also convert test data files in TDL and WGL format with these tools.
- The STDL package comes in a dedicated RPM package that must be installed before usage
- The conversion from STIL, WGL or TDL to the *SmarTest Setup Format* is configured in a setup file.
- In the conversion process the setup data like signals and signal groups, ports, channels, levels, wavetables, waveforms and patterns is translated into settings in *specification files*, *operating sequence files* and pattern files.
- The generated files are part of a completed SmarTest test program project that can be imported into the *SmarTest WorkCenter* to review the converted test data.

# Appendix

## Steps to Prepare Your First STIL Conversion

- Prepare your STIL file with the relevant data for a conversion, minimum signals, timing and pattern
- Create a folder in which you will store the STIL file
- In that folder start either the “stilreader” GUI
  - \$ **stilreader\_gui**
  - ...or start “stilreader” on the command line:  
\$ **stilreader** -target 93000SMT8 <stil\_filename>.stil  
... or ...  
\$ **stilreader** -setup <xxx>.setup <stil\_filename>.stil

STIL Reader tools are located in:  
**/opt/93000/stdl/bin/**

# Run STIL Reader and Verify the Conversion Result Using SmarTest

- Run “stilreader” on the command line in a terminal:  
  \$ **stilreader** -target 93000SMT8 <stil filename>
- Check the messages written into the terminal from “stilreader”.
- Start SmarTest and import the generated project with the name “project”.
- Verify the setups by checking the appropriate files:
  - *DUT board description file*
  - *Specification file* defining signal groups
  - *Specification file* defining levels
  - *Specification file* defining timing
  - Testflow file
  - *Operating sequence file*
  - Pattern file
  - ....