 PIV Automation External Architecture Guide

***Platform Integration and Validation, Data Center Group (DCG)***

**Revision 0.3**

Table Of Contents

[Table Of Contents 2](#_Toc16518013)

[Table Of Figures 3](#_Toc16518014)

[Definition Of Terms 4](#_Toc16518015)

[1. Introduction 5](#_Toc16518016)

[Overview 5](#_Toc16518017)

[Intended Audience 5](#_Toc16518018)

[2. PIV Content Automation Methodology 6](#_Toc16518019)

[Overview 6](#_Toc16518020)

[Fully automated test suite 6](#_Toc16518021)

[Common test content and methodology 6](#_Toc16518022)

[Auto-generation of recipe/documentation 7](#_Toc16518023)

[3. PIV Test Content Architecture 8](#_Toc16518024)

[Overview 8](#_Toc16518025)

[Test Automation Infrastructure 8](#_Toc16518026)

[Source code – directory structure 9](#_Toc16518027)

[PIV-Core Repository 9](#_Toc16518028)

[PIV-Content Repository 9](#_Toc16518029)

[Top-Level Python Package Structure 10](#_Toc16518030)

[Team-Specific “tests” Package 10](#_Toc16518031)

[Configuration file management 11](#_Toc16518032)

[Sample configuration file option and usage 12](#_Toc16518033)

[Implementation structure of test 13](#_Toc16518034)

[Base Test Execution Flow 13](#_Toc16518035)

[Sample test flow 14](#_Toc16518036)

[Logging and log severity control 14](#_Toc16518037)

[Log file generation 14](#_Toc16518038)

[Exception and Error Handling 14](#_Toc16518039)

[4. Providers and Utilities 16](#_Toc16518040)

[Overview 16](#_Toc16518041)

[Providers vs Utilities 16](#_Toc16518042)

[Provider Implementation Details 17](#_Toc16518043)

[Core Providers 18](#_Toc16518044)

[5. Test Automation Framework Integration 20](#_Toc16518045)

[Overview 20](#_Toc16518046)

[Test automation framework 20](#_Toc16518047)

Table Of Figures

[Figure 2‑1 Architectural Layer Block Diagram 7](#_Toc16518002)

[Figure 3‑1 Test Automation Infrastructure 8](#_Toc16518003)

[Figure 3‑1 Repository Breakdown 9](#_Toc16518004)

[Figure 3‑2 Example "tests" package layout 10](#_Toc16518005)

[Figure 3‑3 Detailed examples of feature-specific test packages 11](#_Toc16518006)

[Figure 3‑4 Example configuration file 12](#_Toc16518007)

[Figure 3‑5 Example configuration file usage 12](#_Toc16518008)

[Figure 3‑6 Test Case Architecture Block Diagram 13](#_Toc16518009)

[Figure 3‑7 Example log file clipping 14](#_Toc16518010)

[Figure 4‑1 Architectural Layer Block Diagram 17](#_Toc16518011)

[Figure 4‑2 BaseProvider class 18](#_Toc16518012)

Definition Of Terms

|  |  |
| --- | --- |
| Term | Definition |
| API | Application Programming Interface |
| BKC | Best Known Configuration |
| CPU | Central Processing Unit |
| CR | Intel Optane DC DIMMs, also known as Crystal Ridge technology. Provides memory expansion and persistent memory capabilities. |
| DCG | Data Center Group |
| DPV | Data Center Platform Validation |
| Host | Computer attached to the target system under test used for coordinating test activities and running out-of-band commands. |
| ITP | In-Target Probe |
| OS | Operating System |
| PIV | Platform Integration and Validation |
| PI | Platform Integration |
| RAS | Reliability, Avaliability, and Serviceability |
| SUT | System Under Test |

# Introduction

## Overview

The purpose of this document is to describe the architectural and implementation details about the Platform Integration and Validation (PIV) team test content automation infrastructure. This architecture is aligned across PIV for generating fully automated test content for Platform Integration teams, Platform Validation teams, and external customers.

Today, DCG primarily does Platform Validation and Platform Best Known Configuration (BKC) releases with manually executed test content. As a result, content execution takes significant time, affecting overall validation coverage and release quality. The automated test automation methodology and architecture addresses these challenges. Intel DCG’s automated content is being architected to leverage automated validation interfaces and tools while abstracting the details of the environment, tools, and hardware. Content automation provides a potential 20X cycle time reduction (derived from multiple case studies and actual implementation of automated tests for reference platforms during Purley-Refresh platform validation cycles) in DCG's platform validation cycle and also boosts BKC release coverage.

An ideal test automation architecture needs to address the below pain points, which exist with any manual test execution process:

1. Scalability of the tests across different families, derivatives, and execution environments
2. Reusable test libraries that can be shared across multiple tests
3. Provide easy interface to control test flows and behavior
4. Easily configurable test plan - Plug and Play tests
5. Easier to debug test failures – Provide as much information about the failure as possible from the test log
6. Standard logging and on/off control – Easier to set the severity of the logs being printed and optimize time spent to execute the test
7. Disciplined log archival mechanism – Record management of the log files for understanding the evolution and life cycle of each test
8. Stable test execution and independent test sequence – Irrespective of the sequence of execution of test plan, achieving consistent results
9. Standalone test execution and report generation – Single-SUT level and lab wide automation across different geographies, independently of any automation framework.
10. Complement test development through regression and continuous integration

PIV’s automation infrastructure meets these criteria and this document explains them in detail.

## Intended Audience

This document is intended for:

* Test Content Developers developing or updating recipes
* Test Content Developers working on common utility development
* Test Content Developers working on test automation
* Validation Engineers participating in the code review and/or continuous integration process
* Validation Engineers integrating and executing test plans constructed per this architecture
* Architects developing the automation framework
* Validation Leads planning the regression and test execution cycles required for achieving milestones
* External validation partners

# PIV Content Automation Methodology

## Overview

There are three primary objectives of the PIV content automation architecture:

1. **Fully Automated test suite**

The primary objective of having fully automated tests is to have a 1:1 relation between each test from the test plan and corresponding test script. This enables single platform (standalone) level automation. Full automation comprises of performing necessary pre-test setups, triggering the main stimulus, performing complete checks for determining pass or fail status of the test, archive or dump any critical log needed to debug (if test failed) and clean up the system before exiting the test so that next test can start fresh. This also achieves 1:1 relation between the log file reference and test. The total decouple of test scripts from the test automation framework (Team City, Jenkins, etc.) enables easier and quicker integration of tests with frameworks and tools for achieving lab-wide automation.

1. **Common test content and methodology**

This objective seeks to achieve maximum reusability between tests, features, and domains a primary focus to achieve maximum reusability of the functions across tests. In addition, the architecture seeks to achieve reusable test flows between different execution environment, future generation of CPU families (with minor modifications). It emphasizes on a layered and modular approach to any test and library development; thus providing multiple level of abstractions (Platform type, Target OS running on SUT, etc.) to increase the reusability.

1. **Auto-generation of recipe/documentation**

The number of APIs to be documented will be significant due to the increased set of utility functions developed to support large sets of tests and domains. Manual maintenance of a long list of APIs is an error prone and time consuming activity. These challenges are addressed by automating the process of API, test, and test flow documentation.

Having these objectives incorporated into our methodology, we are able to achieve a **scalable, reusable, maintainable** and **releasable** automated test suite. Further sections under this chapter details each of these core objectives.

## Fully automated test suite

This methodology requires test plans comprised of completely automated tests. There should be one script for one test case which includes:

1. Setting up necessary BIOS knobs, drivers and other software/tool components on the SUT for performing the test
2. Injecting the intended stimulus
3. Capturing necessary logs from/on the SUT
4. Performing the intended pass/fail checks (make the test a self-checking one)
5. Archiving all the logs in one central directory for each test for future reference or debug
6. Performing the clean-up routines necessary to bring the SUT back to its original state in preparation for the next test

## Common test content and methodology

PIV’s automation methodology is devised considering the plurality of the server platform families, CPUs and the platform types. It achieves maximum reusability of tests across different platform families with minimal changes that are required to support running the tests on that specific product family. The automation architecture devised under this methodology has necessary abstractions and layers to achieve this. It also encapsulates the execution environment specifics into these abstractions to support varying execution environments that may have a completely different set of tools (for example, different hardware tools for controlling the power to the SUT) or interfaces (for example, SUT connected to a Host or not connected to a Host). The block diagram below depicts the layers of this architecture:

**Content Architecture**

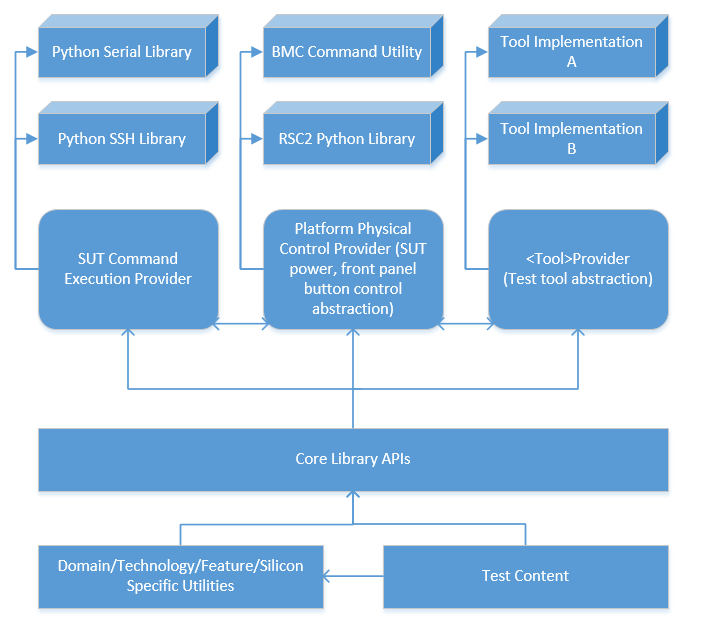


Figure 2‑1 Architectural Layer Block Diagram

Each layer is designed with standard interfaces for each module such that the implementation can be dynamically selected for any environment to achieve 100% reusability of the other layers, especially the Test Layer. The next chapter specifies the details of the content code architecture.

## Auto-generation of recipe/documentation

Documentation of the interfaces, test flow and utilities plays a vital role in achieving easier adoption and compliance to this methodology. This component of the methodology enables a run-time, easy to generate, documentation that matches the actual implementation, and API or interface. It also provides hooks to attach an integration guideline document with necessary pictures that can be attached to a specific test for detailed documentation. It supports HTML and PDF format of documents.

# PIV Test Content Architecture

## Overview

The 4 core requirements of the PIV content architecture are:

1. **Scalable**
   * The architecture must support increasing variants of server platform families with easier adoptability
2. **Modular**
   * The architectural blocks must be easily detached, controlled, or modified, without affecting other blocks
3. **Optimal**
   * Keep the execution time to a minimum to enable many tests to be run within a given time.
4. **Simple and powerful**
   * The architecture must be simple to understand for all users with different sets of skills. This is critical as it enables the adoptability and promotes user retention.

This section defines and details each component of the architecture and how it meets the core requirements.

## Test Automation Infrastructure - Yanlong

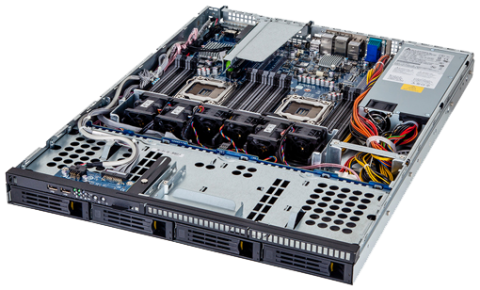
Auto Content Repo

Test Host System

Auto Content (DATF Content)

Automation API (Domain Lib)

Common Library API (DTAF Core)



CRB Platform



Commercial Platform



Simics

Test Case Database

Content SCM Tool

CI Automation Toolchain



Testing Log Database

Figure 3‑1 Test Automation Infrastructure

**Test Utilities** – Modular utility functions for building tests. These are the building blocks of the tests primarily implementing the stimulus and self-checking part of the tests. It also implements the pre-setup procedures needed for the tests.

**Tool Libraries** – Abstracted functions for using tools. These libraries acts as interface between test utilities and tools. Interface to these tools are well structured

**Tests** – Fully automated self-checking executable test scripts (1:1 relation to the each test in the plan)

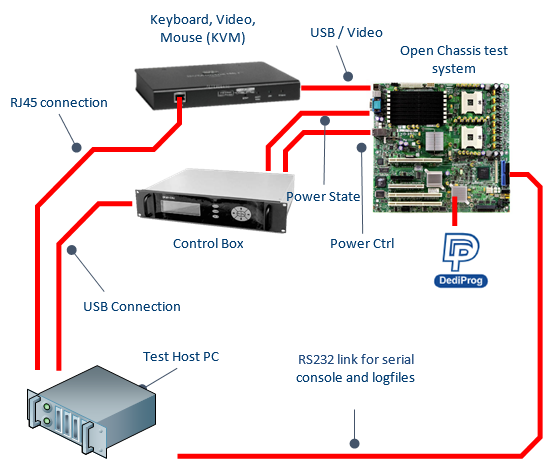
**Test Lists** – List of tests meant for specific purposes (Ex: continuous integration, basic acceptace test, release testing, etc.). These are used for standalone regression at platform level. Test plans for lab wide automation are maintained by the test automation framework.

**Miscellaneous Scripts** – Miscellaneous scripts for achieving efficient user experience. Also includes miscellaneous scripts to run tests standalone (platform level automation) and parse test results to extract human-readable data. Scripts designed for debug use only also fall into this category.

**Tools** – Packaged tools or software packages that are used for specific test purposes. They are generally installed on SUT at run time by the tests if such tool needs to be used by that test. Tools that are used by the hosts can also be kept here to avoid any complex dependencies.

**Continuous Integration** – A tool that automatically runs a subset of test content based on various triggers (see section 5). This enables early detection of issues as new code is checked in. The continuos integration tool also interfaces with code review tools to provide feedback on pull requests.

See the below figure for an example of how the host and CRB/Commercial SUT may be configured:



Closed Chassis Commercial Test System (with integrated **BMC**)



Remote test workstation

Ethernet PDU

Intranet

Figure 3‑1 Example Host/SUT configuration

## Source code – directory structure

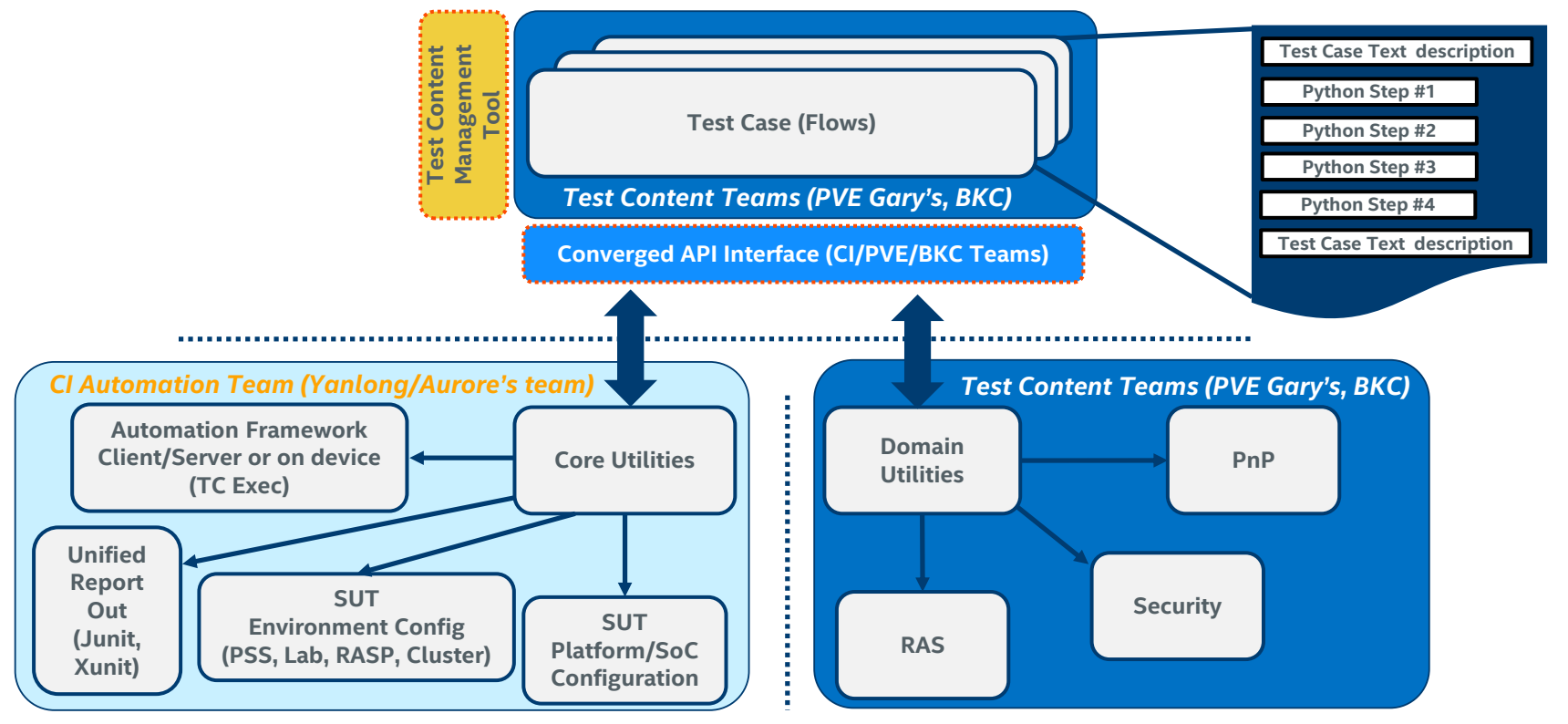


Figure 3‑1 Repository Breakdown

### PIV-Core Repository

General-purpose libraries and utilites will be stored in the piv-core git repository. This includes libraries common across all test types (libraries for working with SUTs, log file structures, etc).

### PIV-Content Repository

Domain-specific libraries/utilities and test content code will be stored in the piv-content git repository. Each content focus area will have its own Python package. In general, dependencies on other piv-common packages should be avoided except when writing interoperability tests, which by their nature will have cross-package dependencies. This approach limits the risk of interference between teams and makes the code review process easier. Code that can be shared across domains should either reside in the piv-content’s “general” sub-package (content-related), or be sent upstream into piv-core (infrastructure-related).

### Top-Level Python Package Structure

Each Python sub-package within piv-content should have the same top-level layout, consisting of **three** **packages**:

1. **lib** (Library Classes, Static Classes, Utility Functions, etc.)
2. **scripts** (Utility scripts designed for use outside of an automated environment, i.e. for test content that cannot be fully automated, or for scripts that assist with debug)
3. **tests** (In the core package, this contains the base test templates, for all other packages this contains the automated test content)

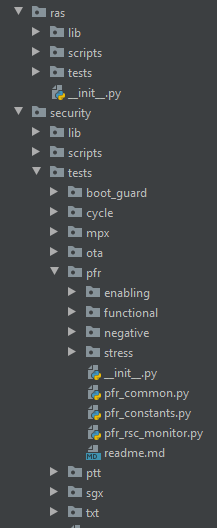


Figure 3‑2 Example piv-content layout (each section below elaborates on specific sections)

### Team-Specific “tests” Package

Any team interested in extending the test suite with content that is specific to their objectives should do so in the “tests” package. This Python package is subdivided by feature or technology, and, in general, will follow the PEP8 convention for module names (short and in lowercase with underscores). They can optionally start with an ID number depending on the team’s chosen module naming convention.

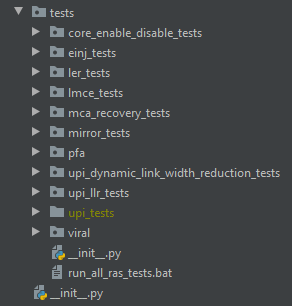


Figure 3‑2 Example "tests" package layout

Each technology/feature folder will be a valid Python package containing a <feature>\_common.py script with common functions that can be shared across all content contained in that module. It can also contain additional shared scripts or modules as necessary.

Test content and all collateral will be stored in subdirectories of the dominant feature or technology. For example, the One-Touch Activation feature uses some other security technologies, so the main content is contained in the “ota” package but imports a few functions from the “txt” package.

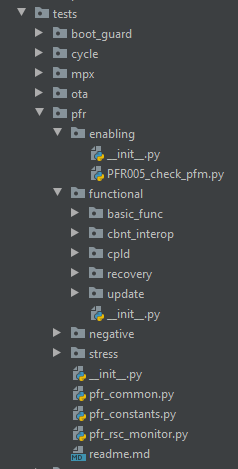
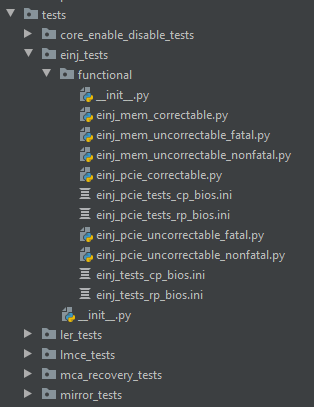
 

Figure 3‑3 Detailed examples of feature-specific test packages

The subdirectories of the feature modules should be organized by function. The table below contains the current standard for these names.

|  |  |
| --- | --- |
| **Name** | **Contents** |
| collateral | Scripts or tools designed to run on the SUT, or non-Python collateral used on the SUT or host. |
| enabling | Tests with basic feature coverage (primarily checks that a feature can be enabled or disabled). |
| functional | Functional tests for the feature (this is the most comprehensive package of coverage). |
| negative | Tests that cover negative scenarios (tests where a failure is expected). |
| stress | Content to stress the device or feature (workload-related tests can also be kept under this category). |
| stability | Tests to validate the consistency or reliability of a feature or device (generally includes cycling, repeated workloads, error injections, etc.). |

Finally, the contents of the above directories (except for collateral) should follow this naming convention: **<Test Case ID>\_**<descriptive name>.py. The ID should start with an underscore if the ID number does not start with an alphabetic character to meet Python naming requirements.

## Configuration file management - Chengming

XML, INI, or some other format that can be broken down into sections. Needs to be machine parsable into an object.

Need two config files - Core config and content configuration file. However, they must follow the same general format to allow reuse of the parsing utilities and for ease of management from the development and execution perspective.

Some configuration details can be dynamically derived at runtime.

**Content Configuration & Core Configuration:**



**Content Configuration:**

Content configuration focues on test domain specific configuration and knowledge. It will not touch the details of providers and drivers.

* + - **Common**

Common Configuration of Test Content should be put in this section. E.g.

* + - * **reboot\_time**
      * **Sut\_command\_time**
    - **Memory**

Memory Domain Specific Configuration of Test Content should be put in this section

* + - * **Platform Cycling**
    - **Security**

Security Domain Specific Configuration of Test Content should be put in this section.

* + - * **Mprime**
        + **running\_time**
    - **Ras**

**Example of Content Configuration**

<content>  
 <common>  
 <reboot\_time>600</reboot\_time>  
 <sut\_command\_time>200</sut\_command\_time>  
 </common>  
 <memory>  
 <platform\_cycler>  
 <number\_cycles>2</number\_cycles>  
 <next\_reboot\_wait\_time>60</next\_reboot\_wait\_time>  
 <dcgraceful>  
 <dc\_on\_time>400</dc\_on\_time>  
 <test\_execute\_time>160</test\_execute\_time>  
 </dcgraceful>  
 </platform\_cycler>  
 </memory>  
 <ras></ras>  
 <security>  
 <mprime>  
 <running\_time>300</running\_time>  
 </mprime>  
 </security>  
</content>

**Core Configuration:**

Core configuration specifies the configuration for providers and drivers of Core API. The content of core configuration includes general information of environment, provider configuration and driver configuration.

* + - **General Information:**

It includes log format, host information, workspace etc.

* + - **SUT Configuration:**
      * **Hardware Configuration**
        + **Memory**
        + **Expansion**
      * **Provider Configuration:**

Provider Configuration specifies the default value which is used to initiate an instance of provider before usage. In provider configuration, if a specific driver is configured, provider will use this driver as the implementation of provider APIs. (See Provider – Driver Table for the supported driver for each Provider).

* + - * + **Driver Configuration:**

Driver Configuration specifies the environment value when driver instance is created.

**Example of Core Configuration**

<?xml version="1.0" encoding="UTF-8"?>  
<core>  
 <host>  
 <logging format="jUnit"/>  
 <root>C:\workspace</root>  
 <os name="Windows" version="10">  
 <kernel version="10.0.14393" />  
 </os>  
 </host>  
 <suts>  
 <sut ip="10.190.191.196">  
 <platform type="commercial" boardname="WolfPass" name="purley" />  
 <hardware>  
 <memory>  
 </memory>  
 <cpu>  
 <family>SKX</family>  
 </cpu>  
 <pch>  
 <family>LBG</family>  
 </pch>  
 <expansion>  
 </expansion>  
 </hardware>  
 <providers>  
 <console/>  
 <sut\_os name="Linux" subtype="RHEL" version="7.6" kernel="3.10">  
 <shutdown\_delay>5.0</shutdown\_delay>  
 <driver>  
 <ssh>  
 <credentials user="root" password="password"/>  
 <ipv4>10.190.191.196</ipv4>  
 </ssh>  
 </driver>  
 </sut\_os>

**XML Node Definitions**

|  |  |  |  |
| --- | --- | --- | --- |
| **XML Node** | **Parameter** | **status** | **Description** |
| General Configuration | | | |
| core/log | path | reserved | Folder of output log |
| core/host/logging | format | reserved | Log Format |
| core/host/root | value | reserved | Workspace of DTAF Core |
| core/host/os | name | reserved | OS Name, e.g. Windows, Linux,  For User Information Only So far. |
|  | version | reserved | OS version. For User Information Only So far. |
| core/host/os/kernal | version | reserved | Kernel Version. For User Information Only So far. |
| core/suts/sut | ip | active | IP Address of SUT. Test Content Developer may use IP to pickup sut for testing. |
| Core/suts/sut/platform | type | reservied | e.g. commercial. For User Information Only So far. |
|  | boardname | reserved | e.g. WolfPass. For User Information Only So far. |
|  | name | reserved | e.g. Purley. For User information Only So far |
| Core/suts/sut/silicon/  cpu/family |  | reserved |  |
| Core/suts/sut/silicon/  pch/family |  | reserved |  |
| Providers | | | |
| Core/suts/sut/  providers/sut\_os | name | reserved | e.g. Linux. For User information Only So far |
|  | subtype | reserved | e.g. RHEL. For User information Only So far |
|  | version | reserved | e.g. 7.6 For User information Only So far |
|  | kernal | reserved | e.g. 3.10, For User information Only So far |
| Core/suts/sut/providers/  sut\_os/  shutdown\_delay |  | active | How long SUT takes to reboot/shutdown. DTAF Core will ping SUT after <shutdown\_delay> seconds. |
| Core/suts/sut/providers/  Sut\_os/driver |  | active | Backend implementation of Provider. Refer to driver section in this table and “Supported Provider and Driver” for more details. |
| Core/suts/sut/providers/  ac/timeout/power\_on |  | active | dtaf core will use this value as the default value of timeout in power\_on api if timeout is not specified explicitly |
| Core/suts/sut/providers/  ac/timeout/power\_off |  | active | dtaf core will use this value as the default value of timeout in power\_off api if timeout is not specified explicitly |
| Core/suts/sut/providers/  ac/driver |  | active | Backend implementation of Provider. Refer to driver section in this table and “Supported Provider and Driver” for more details. |
| Core/suts/sut/providers/  dc/timeout/power\_on |  | active | dtaf core will use this value as the default value of timeout in power\_on api if timeout is not specified explicitly |
| Core/suts/sut/providers/  dc/timeout/power\_off |  | active | dtaf core will use this value as the default value of timeout in power\_off api if timeout is not specified explicitly |
| Core/suts/sut/providers/  dc/driver |  | active | Backend implementation of Provider. Refer to driver section in this table and “Supported Provider and Driver” for more details. |
| Core/suts/sut/providers/  silicon\_debug |  | active | Specify silicon\_debug provider details. |
| Core/suts/sut/providers/  silicon\_debug/driver |  | active | Specify driver details of silicon\_debug |
| Core/suts/sut/providers/  Silicon\_reg |  | active | Specify silicon\_reg provider details. |
| Core/suts/sut/providers/  Silicon\_reg/driver |  | active | Specify driver details of silicon\_reg |
| core/suts/sut/providers/ uefi\_shell |  | active | APIs to interact with UEFI |
| core/suts/sut/providers/ uefi\_shell/driver |  | active | Driver details of UEFI |
| Core/suts/sut/providers/  Console\_log |  | active | APIs to read/write console log |
| Core/suts/sut/providers/  Console\_log/log\_path |  | active | Console log output path |
| Core/suts/sut/providers/  Console\_log/driver |  | active | Driver details of console log (e.g. sol or com) |
| Core/suts/sut/providers/  Bios\_setupmenu |  | active | APIs to manipulate BIOS Setup Menu UI for BIOS configuration |
| Core/suts/sut/providers/  Bios\_setupmenu/efishell\_entry |  | active | Specify the path of efishell entry in Bios menu UI |
| Core/suts/sut/providers/  Bios\_setupmenu/efishell\_entry | Select\_item | active | Menu Option Name to enter UEFI |
| Core/suts/sut/providers/  Bios\_setupmenu/  efishell\_entry/path |  | active | Path to UEFI |
| Core/suts/sut/providers/  Bios\_setupmenu/  efishell\_entry/path/node |  | active | Specify menu option name of the path to UEFI |
| Core/suts/sut/providers/  Bios\_setupmenu/  continue | select\_item | active | Menu option “Continue” on BIOS Setup Menu UI. DTAF will look for the specified option to continue the boot |
| Core/suts/sut/providers/  Bios\_setupmenu/driver |  | active | Bios setup menu driver |
| Core/suts/sut/providers/  Bios\_bootmenu |  | active | APIs to interact with BIOS Boot Menu |
| Core/suts/sut/providers/  Bios\_bootmenu/efishell\_entry | select\_item | active | Menu option name of uefishell entry. DTAF will select the specified option on bios menu to enter uefishell |
| Core/suts/sut/providers/  Bios\_bootmenu/driver |  | active |  |
| Core/suts/sut/providers/bios |  | active | Xmlcli to set bios |
| Core/suts/sut/providers/bios/  driver |  | active |  |
| Core/suts/sut/providers/  Physical\_control |  | active | Normally use control box to control sut. provide APIs to clear cmos and switch usb between host and SUT |
| Core/suts/sut/providers/  Physical\_control/usbswitch |  | active | Specify the timeout to switch usb |
| Core/suts/sut/providers/  Physical\_control/clear\_cmos |  | active | Specify the timeout to clear cmos |
| Core/suts/sut/providers/  Flash |  | active | APIs to flash image on SUT |
| Drivers (Driver section can be found under provider section, see supported providers and drivers for more details) | | | |
| Core/suts/sut/providers/  <provider>/pi |  | active | Rasper berry PI as the control box. Refer to “Supported Providers and Drivers” section for provider list which supports PI as driver |
| Core/suts/sut/providers/  <provider>/driver/pi/ip |  | active | IP address of PI. PI opens REST APIs for controlling |
| Core/suts/sut/providers/  <provider>/ driver/pi/port |  | active | Port name used by PI for rest API support |
| Core/suts/sut/providers/  <provider>/ driver/pi/proxy |  | active | Proxy may be required to access PI |
| Core/suts/sut/providers/  <provider>/driver/cscripts |  | active | Cscripts driver |
| Core/suts/sut/providers/  <provider>/driver/  Cscripts/  debugger\_interface\_type |  | active | ITP is the only option so far. |
| Core/suts/sut/providers/  <provider>/driver/  Cscripts/  Silicon/cpu\_family |  | active | Short name of CPU family. DTAF will read it to use different address mapping |
| Core/suts/sut/providers/  <provider>/driver/  Cscripts/  Silicon/pch\_family |  | active | Short name of PCH family. DTAF will read it to use different address mapping |
| Core/suts/sut/providers/  <provider>/driver/  ssh/ |  | active | Access sut via ssh |
| Core/suts/sut/providers/  <provider>/driver/  ssh/credential | user | active | User name to login SUT via ssh |
| Core/suts/sut/providers/  <provider>/driver/  ssh/credential | password | active | Password to login SUT via ssh |
| Core/suts/sut/providers/  <provider>/driver/  Ssh/ipv4 |  | active | IPv4 address of SUT |
| Core/suts/sut/providers/  <provider>/driver/  com |  | active | COM driver provides serial data for provider API. See “Providers and Drivers” section to find what providers support com driver |
| Core/suts/sut/providers/  <provider>/driver/  Com/baudrate |  | active | Baudrate of COM |
| Core/suts/sut/providers/  <provider>/driver/  Com/port |  | active | COM port.  On windows, it may be COM1, COM2  On Linux, it may be /dev/ttyS0 etc. |
| Core/suts/sut/providers/  <provider>/driver/  Com/timeout |  | active | Timeout to wait for new data. It is normally set as 5 |
| Core/suts/sut/providers/  <provider>/driver/pdu |  | active | Raritan only. |
| Core/suts/sut/providers/  <provider>/driver/pdu | brand | active | raritan |
|  | model | active | See Raritan doc to find model |
| Core/suts/sut/providers/  <provider>/driver/pdu/ip |  | active | IP address of PDU |
| Core/suts/sut/providers/  <provider>/driver/pdu/port |  | active | In most cases, PDU will use port 22 to support SSH. |
| Core/suts/sut/providers/  <provider>/driver/pdu/username |  | active | User Name to login PDU |
| Core/suts/sut/providers/  <provider>/driver/pdu/username |  | active | Password to login PDU |
| Core/suts/sut/providers/  <provider>/driver/pdu/timeout |  | active | Timeout of SSH |
| Core/suts/sut/providers/  <provider>/driver/pdu/  Outlets/outlet |  | active | Outlet number used by SUT for ac power supply. It could be multiple outlets to supply power for one SUT |
| Core/suts/sut/providers/  <provider>/driver/rsc2 |  | active | RSC2 control box. No special parameters required. E.g.  <rsc2/> is enough |
| Core/suts/sut/providers/  <provider>/driver/ipmi |  | active | Wrapper of IPMI |
| Core/suts/sut/providers/  <provider>/driver/ipmi/ip |  | active | IP address of BMC |
| Core/suts/sut/providers/  <provider>/driver/ipmi/username |  | active | User name to login BMC |
| Core/suts/sut/providers/  <provider>/driver/ipmi/password |  | active | Password of BMC |
| Core/suts/sut/providers/  <provider>/driver/ipmi/cmd |  | active | Path of ipmi tool |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Supported Provider and Driver:**

Core configuration specifies the configuration for providers

**(see more XML exapmles in system tests folder of release package)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Supported Provider** | **tag** | **Provider Pameters** | **Supported Drivers** |
| ac power | ac | Power On Timeout – default Timeout which is expected to get ac on | soundwave2k  pi  rsc2 |
| Power Off Timeout – default timeout is expected to get ac off |
| dc power | dc | Power On Timeout – default Timeout which is expected to get ac on | soundwave2k  pi  rsc2 |
| Power Off Timeout – default timeout is expected to get ac off |
| Silicon debug | Silicon\_debug | NA – No provider specific parameters | xdp |
| Silicon Register | Silicon\_reg | NA – No provider specific parameters | cscripts |
| SUT OS | sut\_os | Shutdown\_delay – delay before sut gets shutdown | ssh |
| UEFI Shell | uefi\_shell | NA – No provider specific parameters | com |
|  | sol |
| Console Log | console\_log | log\_path – folder path to save the log | sol |
|  | com |
| Bios Setting | bios | NA - No provider specific parameters | xmlcli |
| Phsical Control | physical\_control | Usb switch timeout – timeout to switch USB between host and SUT | pi |
| Clear CMOS timeout – timeout to get cmos clear |
| BIOS Setup Menu | Bios\_setupmenu | Efishell\_entry – specific the path to enter efishell  Continue – path to continue to boot os  Reset – UI path to reset | COM |
| BIOS Boot Menu | Bios\_bootmenu | Efishell\_entry – UI path to enter efishell | COM |
|  |  |  |  |
|  |  |  |  |

|  |  |  |
| --- | --- | --- |
| **Driver Name** | **tag** | **parameters** |
| Soundwave 2k | soundwave2k | Baudrate – baudrate of COM poart |
| Port – COM Port |
| Voltage Threholds – Threholds of the pin |
| Enable\_s3\_detect – depends on SUT design. Some SUTs provide pin to detect S3 |
| Raspberry PI | pi | IP – IP address of PI  Port – Port Number of PI  Proxy – proxy address. It is optional. It is mainly useful for cross-geo usage. |
| XML Cli | xmlcli | sutospath – path to sut os |
| Bios\_cfgfilepath – config file path to set bios knob |
| Bios\_cfgfilename – file name of conf |
| Ip – ip address |
| User name – user name for login to SUT OS |
| Password – password of SUT OS |
| xdp | xdp | type – only ITP supported |
| SSH | ssh | Credential – username and password to login SUT |
| Ipv4 – ipv4 of SUT |
| Serial Over Lan | sol | Address – IP address of BMC |
| Port – port number |
| Timeout – max timeout for communication |
| Credential – username and password to login BMC |
| COM Port | com | Baudrate – baudrate of COM |
| Port – COM port |
| Timeout – max timeout for communication |
|  |  |  |
|  |  |  |
|  |  |  |

## Implementation structure of test



Figure 3‑6 Test Case Architecture Block Diagram

### Base Test Execution Flow

The BaseTestCase class provides the framework for test case execution. It is extensible, so this can be altered, but the main flow is detailed below.

1. Determine the OS the script is running under
2. Parse command line arguments
3. Create a log file in the project’s log directory
4. Parse the configuration file using the host OS or command line argument override to determine the location of it
5. Determine the execution style (locally if running on the SUT, remote if running on the ITP host, etc.)
6. Construct an instance of the test class, which itself will construct any Providers or helper classes it needs to function
7. Enter the test execution context by calling the \_\_enter\_\_ method of all Providers used by the test
8. Execute the prepare method of the test class. Skip to step 10 if it fails or raises an exception
9. Execute the execute method of the test class, which has the main test content.
10. Execute the cleanup method of the test class
11. Close the log file
12. Exit the test execution context by calling the \_\_exit\_\_ method of all Providers used by the test
13. Return the test result and exit

## Logging and log severity control - Xijun

Tests will use a central logging API to send debug messages to the console and to the log file. The log file will be structured in Xunit format to allow for easy machine parsing. The console/stdout prints shall also be performed through this API, and will have independent control of the debug severity level. For example, all command output may be logged into the Xunit file, but the developer may choose to send only the pass/fail result or execution summary of said command to the console.

## Log file generation

Log files are generated for each test when the test is run. Logs will be stored in the directory specified in the configuration file, with the class name of the test and with .log extension.

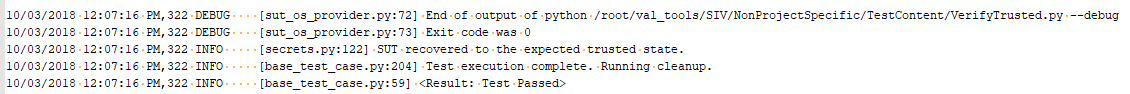


Figure 3‑7 Example log file clipping

Note that this functionality is provided by BaseTestCase’s cleanup method. Whenever a developer overrides cleanup (or prepare), they must call the superclass’ cleanup/prepare methods.

## Exception and Error Handling

BaseTestCase will catch, log, and re-raise exceptions thrown during the test. All exceptions should have a descriptive error message to assist with debugging. Avoid raising exceptions with no help text. Consider the impact to the flow. Exceptions not caught in the test will end execution. Developers should leverage “try/except” blocks when performing operations that have configuration impact to perform any cleanup to avoid affecting downstream tests. Any test case

## BaseTest Case Template Design

### Goals

* Resource Management is user-transparent.
* Exception or Error should not impact on the resource management
* Flexibility & Stablity: User can combine the usages of Provider APIs for testing flexibly inside test case template

### Overview:

Base Test Case Template maintains the context of test case. All drivers will be registered to the context. Once test case exit normally or abnormally, the context will release all the registered driver resources before exiting.

### Class Diagram:

CaseContext: work as singleton class which inherited from ExitStack. Call \_\_exit\_\_ of driver to release resource

BaseTestCase: maintain the life cycle of context.

DriverFactory: register driver to CaseContext when creating driver



### Flow of Test Case:



### Call Sequence Diagram:



that raises an exception to the BaseTestCase main method will be marked as failing.

# Providers and Utilities

## Overview

Providers abstract functionality that could potentially be replaced with a different tool or package or logic in different environments. Put simply, the usage of each Provider is the same across all execution environments, as the implementations are dynamically changed for each environment. Having implementation details wrapped up in an object oriented interface enables developers to easily implement functionality required by their own execution environment. This chapter discusses the different implementation strategies for providers in a layered approach, and lists the implementation details of a few core providers as an example.

## Providers vs Utilities

Utilities are the first level implementation that provides building blocks for the tests. Tests directly call the Utility functions. Utility functions call into Provider methods, whose behavior is controlled through abstraction and configuration file information. Providers and Utilities should be designed to abstract the following details away from the test case logic:

1. Different flows or methods used between different operating systems.
2. Silicon functionality differences between product families.
3. Tool implementation differences between execution environments.

Providers are built with fixed interfaces (method name, input arguments, and output type) for a specific purpose. The way the purpose is achieved can be changed easily by overriding the functionality by replacing it with a new provider based on the execution environment or tool used. This is the backbone of our test architecture that enables scalability and reusability with minimal changes. The below figure shows the typical usage of provider implementations to achieve reusability and scalability of a specific test between different operating systems, silicon families, and/or different tools. It shows the cascaded implementation of multiple abstraction that can be implemented depending on the level of abstraction needed. There is no fixed level of abstraction for provider implementation; the abstraction level depends on the needs of the provider. Note that what isn’t on the diagram is just as important as what is. No test or utility function (or even other provider) directly accesses external libraries and tools. All calls to those are handled by a single provider. That way, no dependency can be created between test code and external packages.

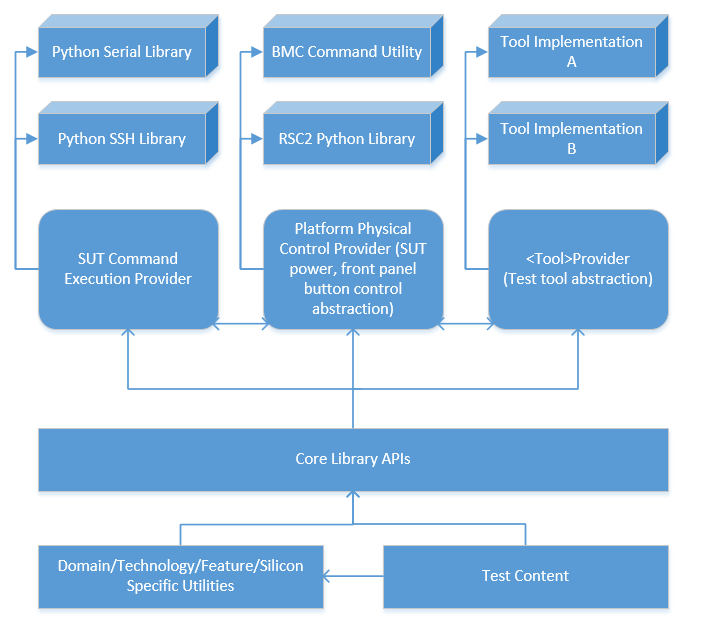


Figure 4‑1 Architectural Layer Block Diagram

Test Content – Test scripts in the framework containing the actual test logic.

Domain/Technology/Feature/Silicon Utilities – Classes and functions used to abstract specific details of a technology or product. This is a higher level of abstraction than a Provider offers.

Provider – Classes used to wrap external libraries/dependencies to provide a consistent interface to all core library APIs. This allows the implementation to be changed without any change to the bulk of the code. Providers also must abstract the execution environment details away from the user (with or without a host, or one-to-many host-to-SUT mappings).

As an example, there may be two products that need to have a register value checked. The access mechanism would be abstracted by a Silicon Abstraction Provider (OpenIPC vs. ITP). However, things like the register name, register address, and expected value will be contained in one or more Utility classes, which use the Provider interface to extract the details needed from the SUT. This prevents Providers from growing to an unmanageable size.

The Utility classes also abstract OS differences. Test content will require a wide array of operating system commands, so rather than implement everything in the SUT OS communication provider, the OS communication provider instead focuses on the communication between the script and the SUT. The commands to run will be abstracted by the Utilities as needed, to keep the test logic as OS-agnostic as possible.

## Provider Implementation Details - Brice

One of the objectives of the library is to support polymorphism for our test content: using the same test logic across types of reference platforms, commercial platforms, OEM platforms, and even across silicon generations. To facilitate this, abstract any code that interacts with dependencies outside of our direct control so that it is possible to change the implementation on the fly, transparently to the core test logic.

The BaseProvider Python class provides a framework for implementing libraries that interact with things like serial ports, ITP/Debug tools, BMC, operating systems, PythonSV, etc. Provider implementations should be in the piv-core repository.

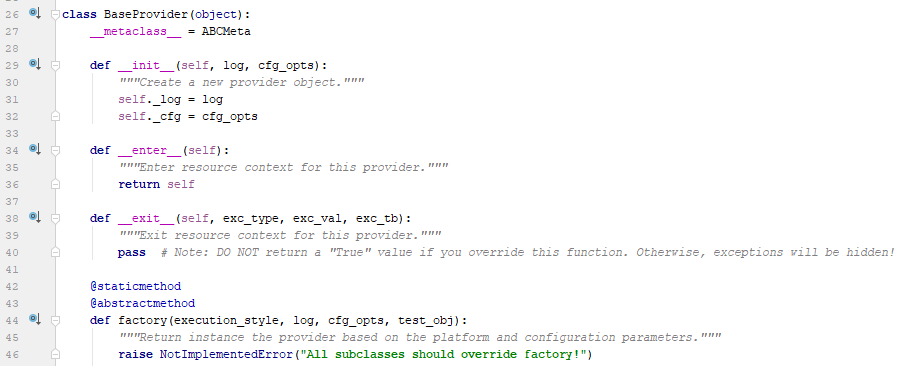


Figure 4‑2 BaseProvider class

A Provider must implement a minimum of **four** methods:

**\_\_init\_\_ -** Handles initialization of any member variables, and extracts any information needed from the configuration file (see the next section).

**\_\_enter\_\_ -** This method will be called immediately prior to test execution. This allows for deferred setup and initialization tasks. For example, only opening virtual serial ports once the Provider determines how many need to be allocated for use.

**\_\_exit\_\_ -** Called after test execution, regardless of how it finished (pass, fail, exception, etc.). This allows for graceful cleanup and release of any resources used by the Provider.

**Factory -** Called to instantiate a copy of the Provider. The key point here is that the initial class that derives from the BaseProvider class is never instantiated directly. Instead, the factory method will look at the configuration file to determine what environment it is running in, and pick from a variety of implementations that meet the interface to create. This is what provides the flexibility to run across different configurations.

Any subclass of BaseProvider is a Python Context Manager, because it has \_\_enter\_\_ and \_\_exit\_\_ methods. In short, this enables Provider objects to be added to ExitStacks or used with “with” statements to provide safe resource management. When entering a “with” block or when an ExitStack context is entered, the \_\_enter\_\_ function will be called. Then, no matter how the block is exited from (exception, function return, etc), the \_\_exit\_\_ method will be called to allow the Provider to cleanly tear down. This allows the BaseTestCase to use an ExitStack to manage Providers that the test creates in its \_\_init\_\_ block, and provides a handy way for library developers to use Providers created a runtime, through a “with” block, without manually dealing with resource allocation/deallocation.

## Core Providers

The “core” package has the basic providers that all content types use. For example, the test framework needs ways to issue commands to the SUT’s OS, the SUT’s power supplies, and the SUT’s CPU debug port.

1. Operating System Provider
   1. SSH – When running a test script remotely (from a Host), the script can issue commands to the SUT over SSH.
   2. Local Execution – When running a test script on the SUT directly, the script can use the Python subprocess module to issue commands instead.
   3. Serial – Some products may require the use of serial port communication, especially in the early stages of power-on. As such, the script can also execute commands over the serial port using a different concrete implementation of the OS command provider.
2. Platform Control Provider
   1. RSC2, Soundwave, etc. – Some execution enviornments will use sophisticated mechanisms like the RSC2 or Soundwave, which can provide AC power control, jumper control, and more.
   2. USB Power Strip – Some execution environments may use a simple USB-controlled power strip connected to the host to control AC power connections to the SUT. In this case, the other methods of the interface will need to be implemented with different mechanisms.
3. Debug Interface Provider
   1. ITP – Many Intel products use ITP as the primary CPU debug tool.
   2. OpenIPC – Newer products may use OpenIPC instead, and the Provider must support it as well.

External customers can write provider implementations for any tools used in their validation flows. If the execution environment is different, write a new subclass of the required provider using whatever tools are needed. The Providers in the “core” package also serve as a reference for implementing the interfaces. Then, update the superclass’ factory method and the configuration file to allow the test framework to select the new implementation. This way, test utilities and test logic will not need to change, as all utilities and tests code to the Provider interfaces rather than specific implementations.

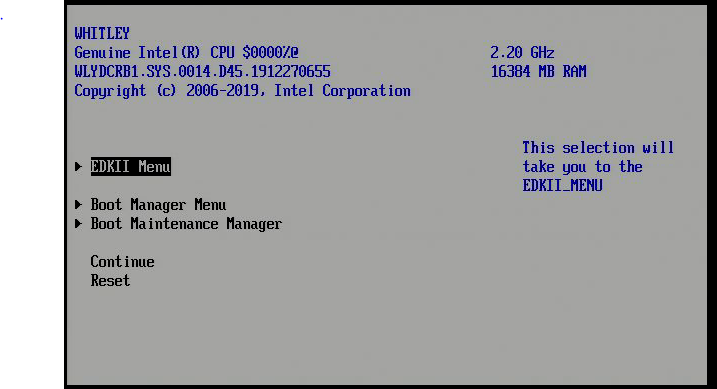
## Bios Menu Providers – Chengming

Bios Menu Providers include BIOS Setupmenu Provider and Bios Boot Menu Provider to provide APIs to work with different BIOS Menu UI.

### BIOS Setupmenu and BIOS Boot Menu

Before calling the BIOS Menu APIs, you have to understand what BIOS Setupmenu and BIOS Boot Menu are.

BIOS Setupmenu – provide various bios option to configure BIOS. In manual test, it can be accessed by pressing F6 / F7 during booting.



Bios Boot Menu – provide boot options (See below). In manual test, you can access boot menu by pressing F2 during booting



### BIOS Menu Provider Configuration

Below are two examples of BiosSetupMenu and BiosBootMenu (Refer to Configuration File for more details).

#### BIOS Setup Menu Configuration

**driver** – In latest version, BIOS Setup Menu Provider only support com as the driver.

In most cases, only port need to be changed based on what com port is used on your host.

**efishell\_entry** – specifies the path to “Launch EFI Shell” option on Menu UI. The path varies on different platform.

It is better to manualy access setup menu to confirm the path and name are correct.

**continue** – Specifies the option name to conintue the boot.

**reset** – defines how to reset in BIOS Setup Menu

<bios\_setupmenu>  
 <driver>  
 <com>  
 <baudrate>115200</baudrate>  
 <port>COM100</port>  
 <timeout>5</timeout>  
 </com>  
 </driver>  
 <efishell\_entry select\_item="Launch EFI Shell">  
 <path>  
 <node>Setup Menu</node>  
 <node>Boot Manager</node>  
 </path>  
 </efishell\_entry>  
 <continue select\_item="Continue"/>  
 <reset press\_key="\\33R\\33r\\33R" parse="False"/>  
</bios\_setupmenu>

#### BIOS Boot Menu Configuration

**driver** – In latest version, BIOS Setup Menu Provider only support com as the driver.

In most cases, only port need to be changed based on what com port is used on your host.

**efishell\_entry** – specifies the path to “Launch EFI Shell” option on Menu UI. The path varies on different platform.

It is better to manualy access setup menu to confirm the path and name are correct.

**continue** – Specifies the option name to conintue the boot.

**reset** – defines how to reset in BIOS Setup Menu

<bios\_bootmenu>  
 <driver>  
 <com>  
 <baudrate>115200</baudrate>  
 <port>COM100</port>  
 <timeout>5</timeout>  
 </com>  
 </driver>  
 <efishell\_entry select\_item="Launch EFI Shell"/>  
</bios\_bootmenu>

1. Create BIOS Menu Provider
   1. Bios SetupMenu:
2. Access BIOS Menu

To Access BIOS Menu, you need to follow below steps

1. Reboot SUT

In test case, you need to boot sut to access bios menu. DTAF provides a series of APIs to boot SUT.

e.g. call power\_off and power\_on of ac power provider

or call power off and power on of dc power provider

or call reboot API of sut\_os

etc.

1. Call wait\_for\_entry\_menu

Both BiosSetupMenu Provider and BiosBootmenu provider implemented the API.

Once SUT is booting, call the API.

1. Call API to press F2 or F6/F7 once wait\_for\_entry\_menu return
2. Call wait\_for\_bios\_menu

Once the api returns, you call call bios menu APIs to interact with UI.

1. Interact with BIOS UI
2. BiosSetupMenu API Usage.

select – this api will move the highlight to the specified BIOS Item

enter\_selected\_item – this API will enter the selected BIOS Item.

press – send key to BIOS Menu

get\_selected\_item – return the option name and value (if any) of the selected BIOS item

input\_text – input text to BIOS Menu

wait\_for\_bios\_menu – only used after press F6/F7 at entry menu

example: Please refer to tests/system/providers/test\_setupmenu.py for more details.

1. BiosBootMenu API Usage

select – this api will move the highlight to the specified BIOS Item

enter\_selected\_item – this API will enter the selected BIOS Item.

press – send key to BIOS Menu

get\_selected\_item – return the option name and value (if any) of the selected BIOS item

wait\_for\_bios\_menu – only used after press F2 at entry menu

## Simics Support - Chengming

Simics will also be a supported execution mode. It should run in a transparent manner wherever possible. Provider interfaces will also need concrete implementations supporting Simics.

* Virtual SUT Service
  + Independent Process to handle remote call from Core / Providers
  + Communicate with Simics Instance via telnet (simics interface)
  + Command execution in Simics Console
  + Reading output
  + Prepare environment for simulating
  + Use abstract layer interface for communication
  + Basic simics operation
* Interaction
  + Simics APIs in Core Layers explicitly launch/shutdown Simics Instance via remote call to VIrutal SUT Service
  + Virtual SUT Service handles remote call to launch/shut simics
  + Simic Providers to implement provider interface via command execution request to Virtual SUT Service
  + Called by Core and Abstract Layer



# Test Automation Framework Integration

## Overview

Our test architecture fully decouples the automated tests and test framework. These automated tests can be run with a single command line and used to achieve standalone regression by calling these commands in a sequence (batch file or shell script). Moreover, to enable lab wide automation and also triggering the automation across all the regression machines and deployment of latest BKCs, help of test automation framework is needed. Multiple test frameworks are available, such as Jenkins or TeamCity. Each of these automation frameworks work on the basic philosophy of enabling a continuous integration environment and at scale automation across pools of systems that can be triggered using multiple qualifiers (time based, day based, release based, code check-in based, release build available in a cloud server, and many more).

## Test automation framework

Typical automation frameworks provide support for:

1. Creating test plans
   1. Enables different test plans created for specific purposes (Basic Acceptance Tests, Nightly, Weekly, etc.)
2. Setting up automatic triggers
   1. Provides a mechanism to start the regression on multiple systems upon:
   2. New code check-in
   3. New BKC release
   4. Time based
3. Provides platform for archiving test results
   1. Establishes automatic record management for test results. Useful for future references and also for collecting historical trends of regressions for a given project milestone
   2. These artifacts can be traced from build id or test plan id or BKC id for easy access
4. Test reports
   1. Test reports are sent via email and also stored in the automation server that must be presentable report for any given run
5. Dashboard
   1. Dashboards extract statistics for test trends, progress, and system utilization