Test Development Using the UCF Framework

Richard Todd
Test Engineering Services
03.15.2023

Table of Contents

1. Purpose

This document defines the process to develop, verify and deploy an automated test station using the UniConsole Framework (UCF).

The focus of this document is primarily on using the UCF to integrate and automate the test station.

The intended audience for this document are test engineering developers using the UCF to build, automate and verify an electronic functional test station.

2. Terminology

Term	Definition
UCF	The UniConsole Framework
Test Spec (ification)	A document that defines each parameter to be tested, and the generic methodology used to create the measurement value.
Management Console	The tab/text based console that is used to monitor and control all application functionality
Test Executive	The user interface used to automatically sequence all tests and save measurement data to a local database
Go-nogo	A test sequence initiated by the Test Executive will stop on the first measurement failure
Host Computer	The computer that will control the test station's instrumentation
Profile Group	The ***'./ucfroot/opt/MyTestGroup/'*** directorycontains the 'Test Profile' subdirectory as well as a common IO configuration database.
Test Profile	The ***'./ucfroot/opt/MyTestGroup/INSTANCE'*** directory that contains 1 or more shared libraries that will be automatically loaded in order to create the test application.
Test Project Directory	The directory used to create a new test automation application. The initial contents are copied from ***./ucfroot/usr/src/_TplModule/****
Takt Time	Takt time is the average time interval between the start of production of one unit and the start of production of the next unit when items are produced sequentially.
System Variables	Text or numeric variable definitions stored in a UCF embedded database
Path-Link	Symbolic link definitions stored in a UCF embedded database
DUT	Device under test

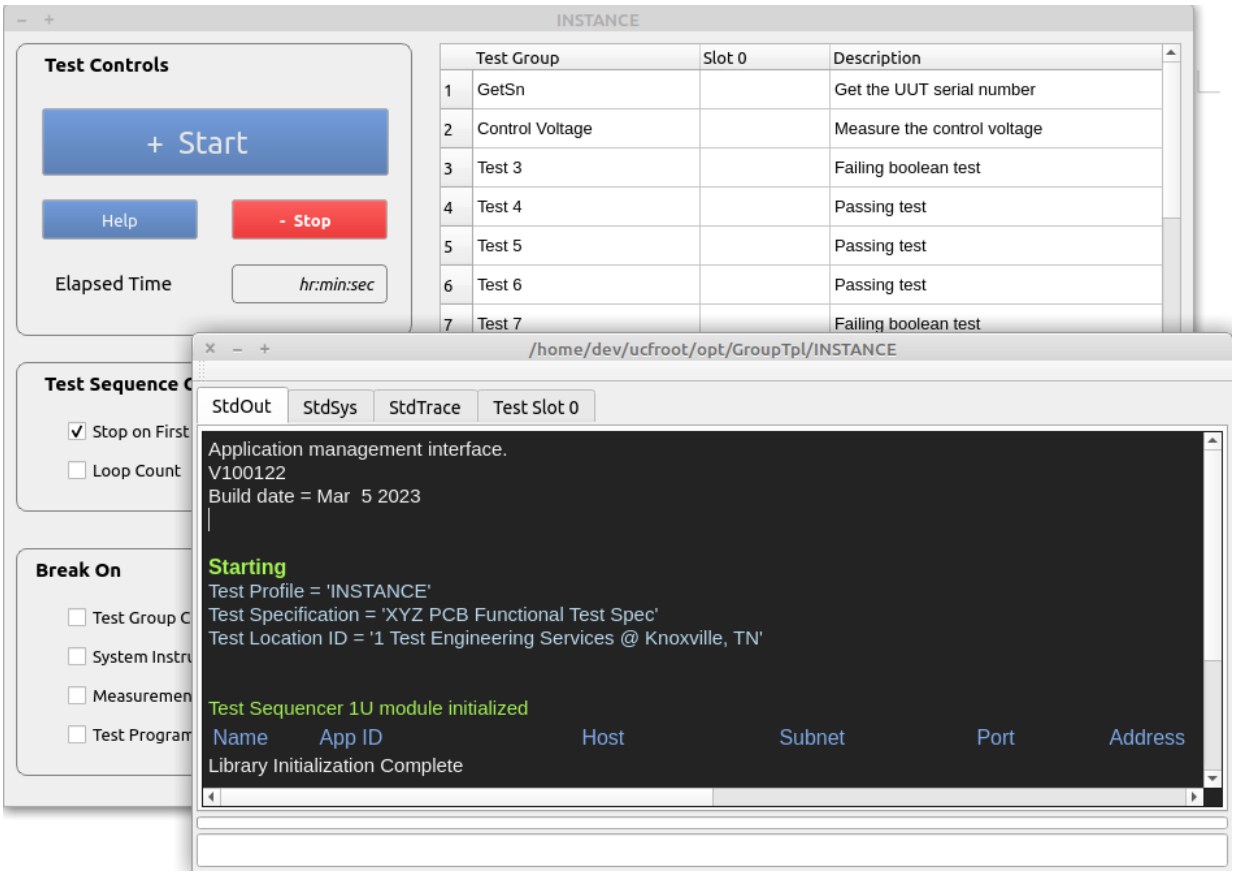
3. Related Documents

Document	Description
UCF Configuration Notes.pdf	This document defines the UCF Test Executive configuration process
'UCF System Variables Notes.pdf	This document outlines using system variables
UCF Template Module Notes.pdf	Detailed information on the structure of TplModule.cpp
TplModule.h/cpp	Module development source file(s) contained in the UCF deployment package These files are referenced by all code examples in this document.

4. UCF Overview

The UniConsole framework is a C++ bases software package used for quick-turn test development using user developed plug-in modules.

- Based on a profile name, plug-in modules are loaded on startup to create a test application
- The Management Console can be used to monitor and control all test system functionality
- A graphical Test Executive is a standard plugin that is automatically loaded for all profiles
- Test limits are defined in an embedded database and organized by test specifications and UCF profiles



4.1. UCF Startup

The UCF is started using a command-line (or shortcut) command'.

'*./ucfroot/bin/UniConsole(.exe) -p ./ucfroot/opt/PcbFunctional/TestSuite_1*'

Where

- *'./ucfroot/opt/PcbFunctional/TestSuite_1'is the path to the target profile to execute*
- *All shared libraries (*.so or *.dll) will be automatically loaded*
- *Any errors loading a shared library will terminate the sequence and lock the framework*

a) Startup Options

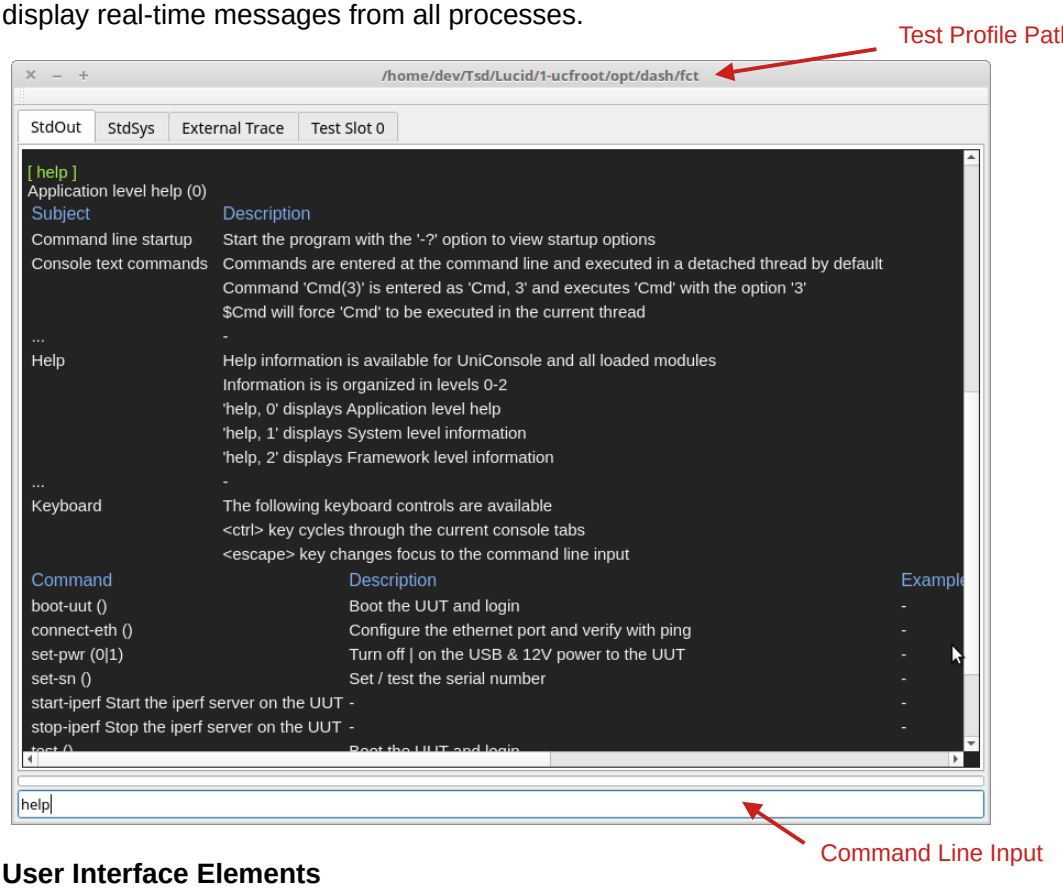
The following command line options are available for framework startup

Option	Argument	Description
-h		Show help information
-p	directory	Target profile directory
-c	directory	An alternate path containing ‘IOConfig’ and ‘StaticModules’ subdirectories. This overrides the path to ~/ucfroot/bin/..
-d	1 or 0	Enable or disable a dark Management Console background
-f	-2 to 2	Set the relative size of the Management Console text
-l	0 or 1	Disable or enable loading common profile directories
-g	0 or 1	Disable or enable ‘Debug’ mode
-w	5 to 2000	Set the horizontal size of the Management Console
-v	100 to 1000	Set the vertical size of the Management Console

Example: ‘*./ucfroot/bin/UniConsole(.exe) -p ./ucfroot/opt/PcbFunctional/TestSuite_1’*

4.2. The Management Console

The management console provides a command line interface to invoke registered commands as well as display real-time messages from all processes.



User Interface Elements

- The complete test profile path is displayed in the window's title bar
- 'StdOut' tab is used to display the responses from entered commands as well as general purpose messages.
- 'StdSys' tab is used to display system related messages
- 'External Trace' tab is used to display IO debug/trace information when tracing is enabled
- 'Test Slot 0' tab is used to display measurement results from test execution operations
- The command line input is used to enter text commands and provides 'tab completion' input capability

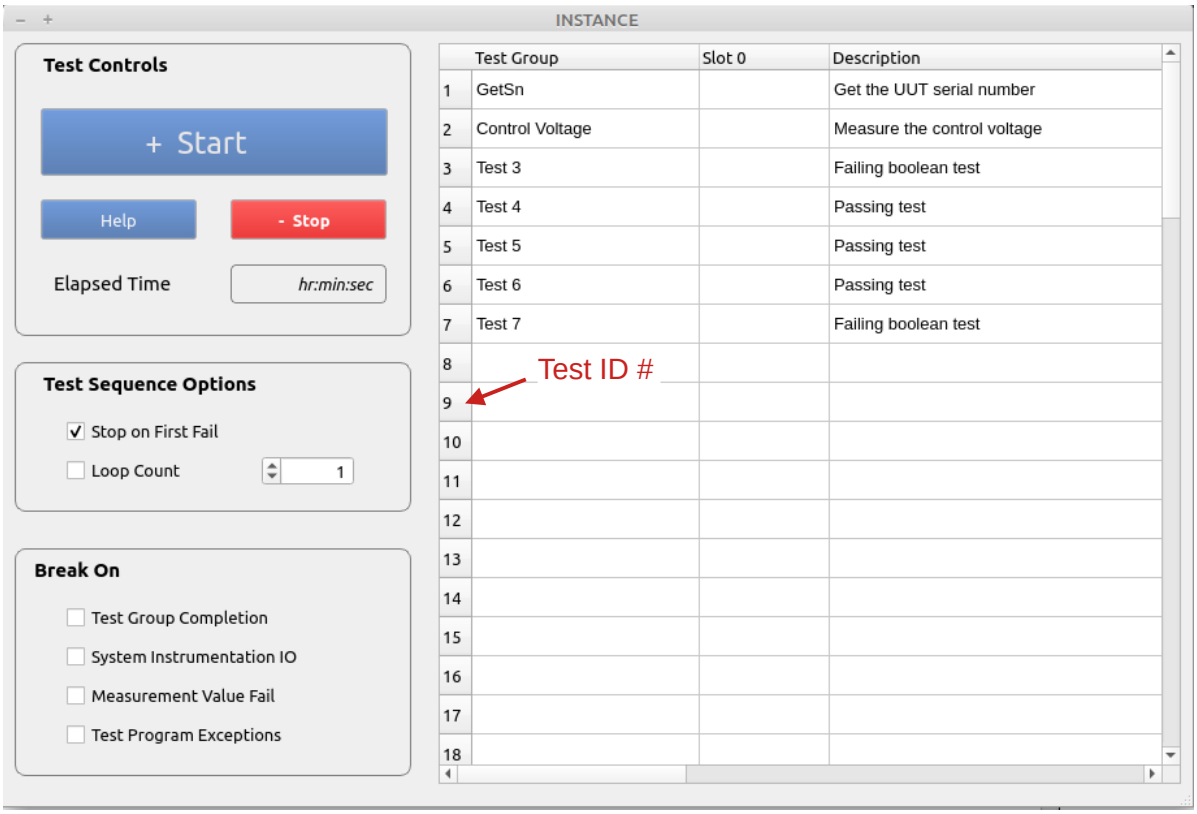
Useful Commands

- '**help**' displays all registered commands and a brief description
- '**start-test'** will execute a test session in the same manner as selecting the 'Start' button in the Test Exec
- '**start-test,1**' will execute a test session that stops on first fail
- '**start-test * 100** ' will execute a test session that loops 100 times
- '**tn**' will interactively execute a test step where '**n**' is the Test ID# from the Test Executive

4.3. The Test Executive

The Test Executive allows test sequences to be graphically executed and monitored.

This interface is designed to be used by operators in a DVT or manufacturing test environment.



Test Controls

- The '**Start**' button is used to initiate a test sequence
- The '**Stop**' button will terminate an 'in process' test sessions
- The '**Help**' button will display test procedure information
- The '**Test ID #**' is a test step identifier used with the 't' command in the Management Console

  *Example: The **'t2'** command will execute the 'Control Voltage' test group*

Test Sequence Options

- The '**Stop on First Fail**' checkbox will force a test session to stop on the first measurement failure
- The '**Loop Count**' checkbox allows test sequences to be executed multiple times

Break On

- '**Test Group Completion'** pauses execution after each test group (step)
- '**System Instrumentation IO**' pauses execution at the start of any system IO operation
- '**Measurement Value Fail**' pauses execution each time a measurement fails test limits
- '**Test Program Exceptions'** will pause each time a C++ exception is caught

4.4. UCF Configuration

The UCF is highly configurable and provides the following configuration options

See ‘UCF Configuration Notes.pdf’ for details

See ‘UCF System Variables Notes.pdf’ for details

4.5. User Interfaces

- Management Console GUI interface size and location
- Test Executive GUI interface size and location
- The initial test mode (Go-nogo)

4.6. Test Sequencing

- The manufacturer's name and location of the test station
- Test specifications, parameters and limits for each test profile
- The number of units to be tested concurrently
- Manual UUT serial number verification

4.7. System Configuration

- System Variables
- System Path Aliases

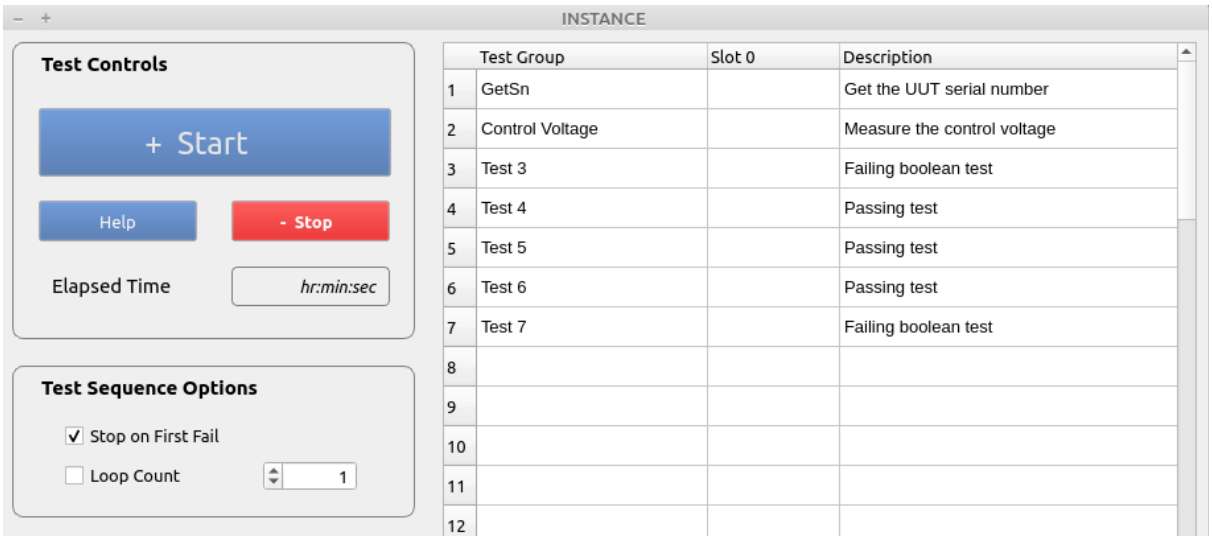
5. Test Sequences and Functions

A UUT is tested by executing a test session using either the Test Executive or the Management Console.

- A test session is a group of 1 or more test sequences.
- A test sequence is a group of 1 or more registered test functions.

5.1. Test Session Execution using the Executive

Execution using the graphical Test Executive is intuitive and is primarily intended to be used by test operators and technicians.



Options

- The test sequence is executed by selecting the '**Start'** button.
- The **'Stop**' button will synchronously terminate a test sequence
- When the **'Stop on First Fail'** checkbox is selected, execution will stop after the first failure.
- When **'Loop Count'** is selected, the test sequence will be executed the specificed number of times consecutively.

a) Test Sequence Definition

Test sequences are UCF API functions that are programmatically registered in the Test Executive.

- The order of test function registration defines the order of execution in the executive
- The first test function registered will verify the serial number that is input graphically or read from the UUT.
- When multiple sequences are executed in a session, the first test function will only be executed at the start of the first sequence.

b) Exception Handling

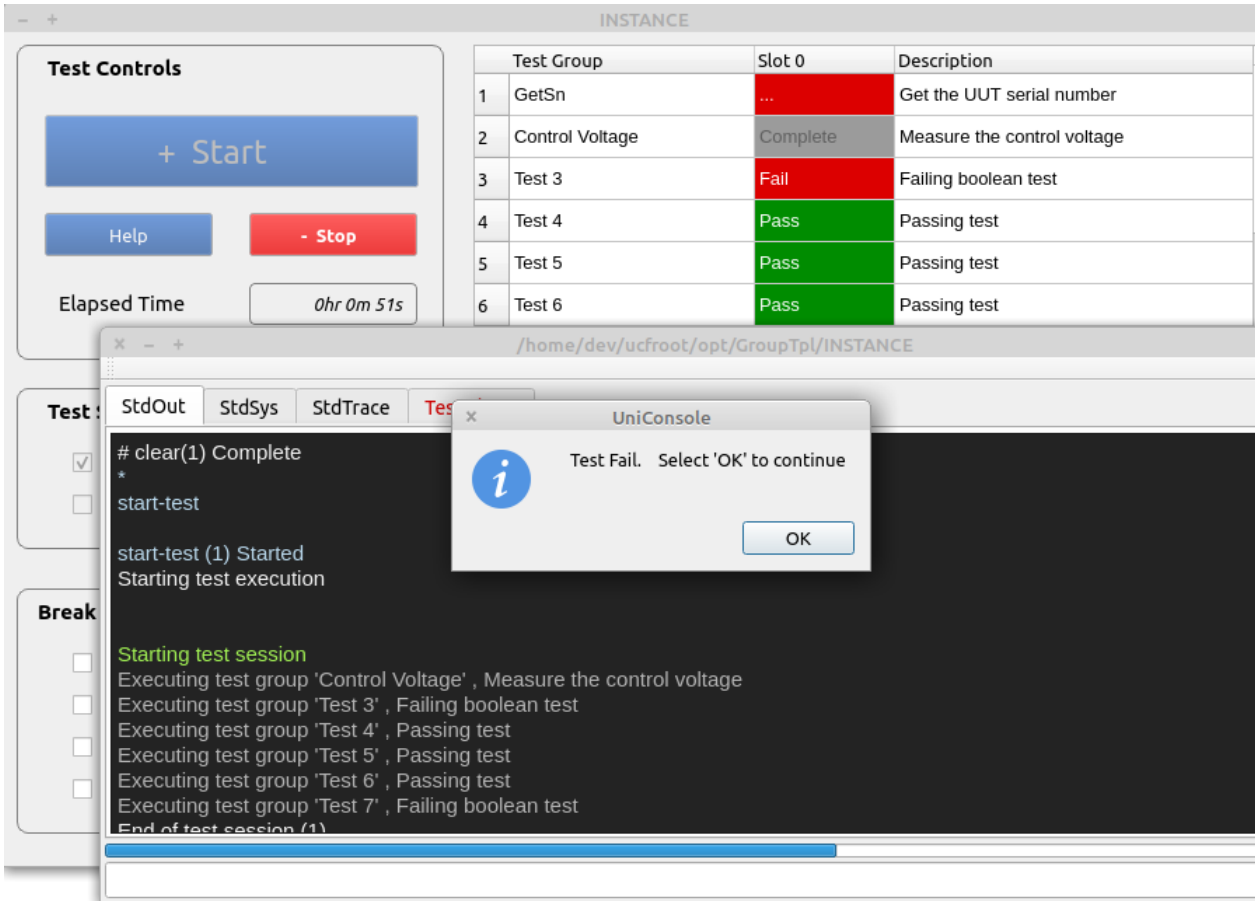
Exceptions thrown by user developed test code will be handled as follows:

- The current test session / sequence will be terminated immediately
- A dialog will be displayed indicating that an exception has occurred and the system must be restarted
- A dialog will be displayed instructing the operator to remove the UUT.

5.2. Test Session Execution using the Management Console

Test sessions, sequences and functions can be interactively executed using the Management Console command line input.

- This method provides far more flexibility than the graphical Test Executive and is intended to be used during development, validation and system debug.
- When a test sequence is executed from the console, the GUI Test Executive will be updated with in-process test results and all data is saved.



Test Session Options

- **'start-test'** will execute a single sequence that does not stop on first fail
- **'start-test,1**' will execute a single sequence that stops on first fail
- **'start-test*100'** will execute 100 consecutive sequences
- **'abort-session'** will asynchronously abort a test session at the end of the current test step
- **'stop-session'** will synchronously abort a test session at the end of the current test sequence

a) Exception Handling

Exceptions thrown by user developed test code will be handled as follows:

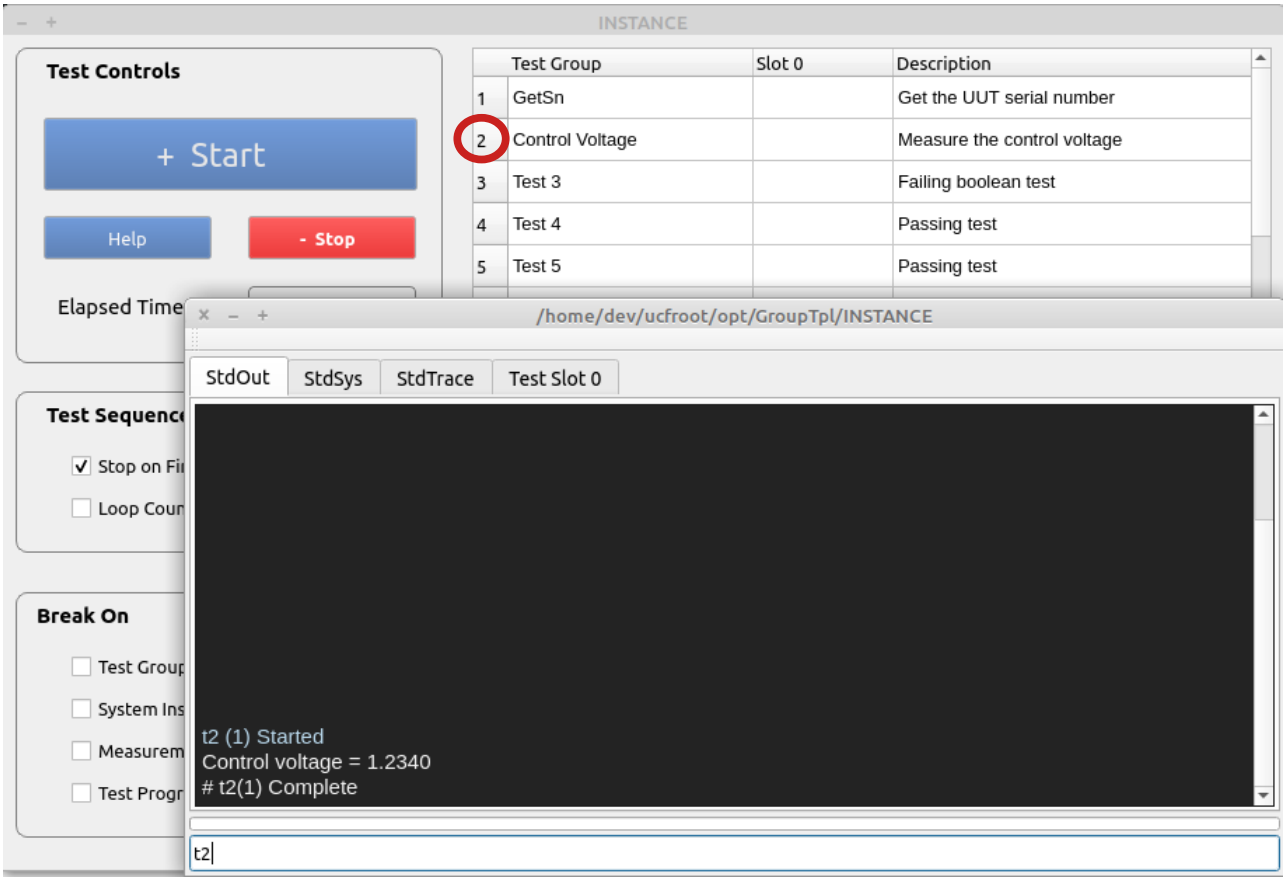
- Command execution will terminate
- An error message will be displayed in 'StdOut' using red text.

5.3. Test Step Execution using the Management Console

Interactively executing test steps (functions) from the command line is an important tool for test development and station maintenance.

This capability eliminates the need to modify code (commenting out functions, rebuilding the module, etc) in order to execute a single test.

- Data is displayed in the ‘StdOut’ tab
- The Test Executive is not updated
- The test data is not saved to the database



Command Options

- **‘t2’** will execute the registered test function #2 (‘Test Control Voltage’) in the Test Executive
- ‘**t2 *100’** will execute the registered test function 100 times

a) Exception Handling

Exceptions thrown by user developed test code will be handled as follows:

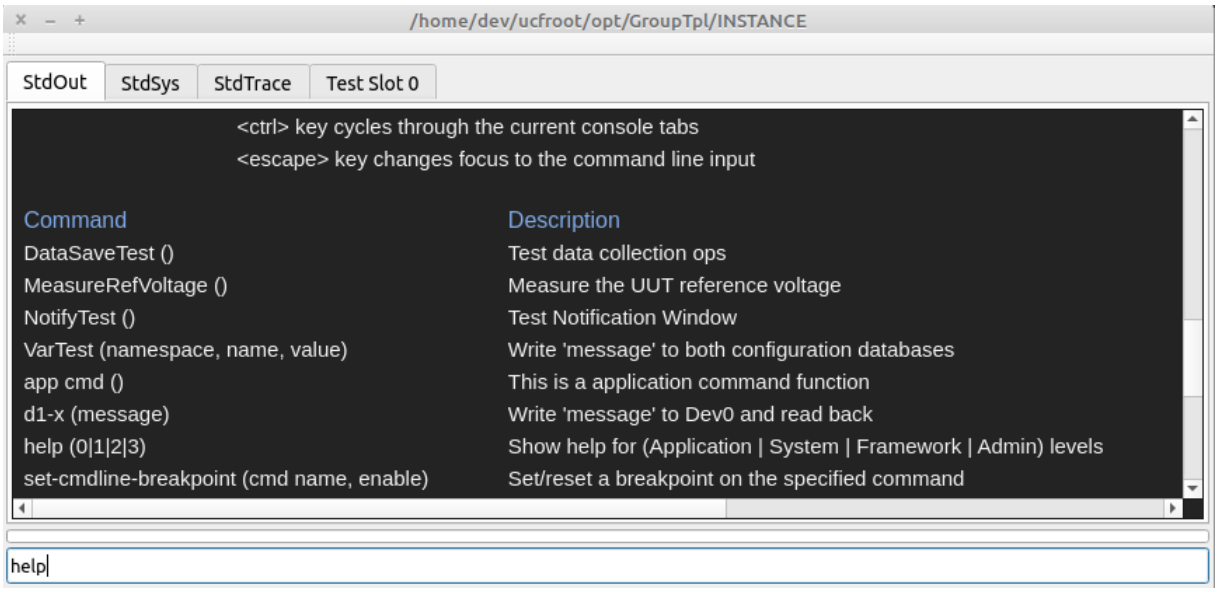
- Command execution will terminate
- An error message in ‘StdOut’ will be displayed in red text.

6. Utility Commands

Utility commands are UCF API functions that have been registered in the framework and can be executed using the command line input.

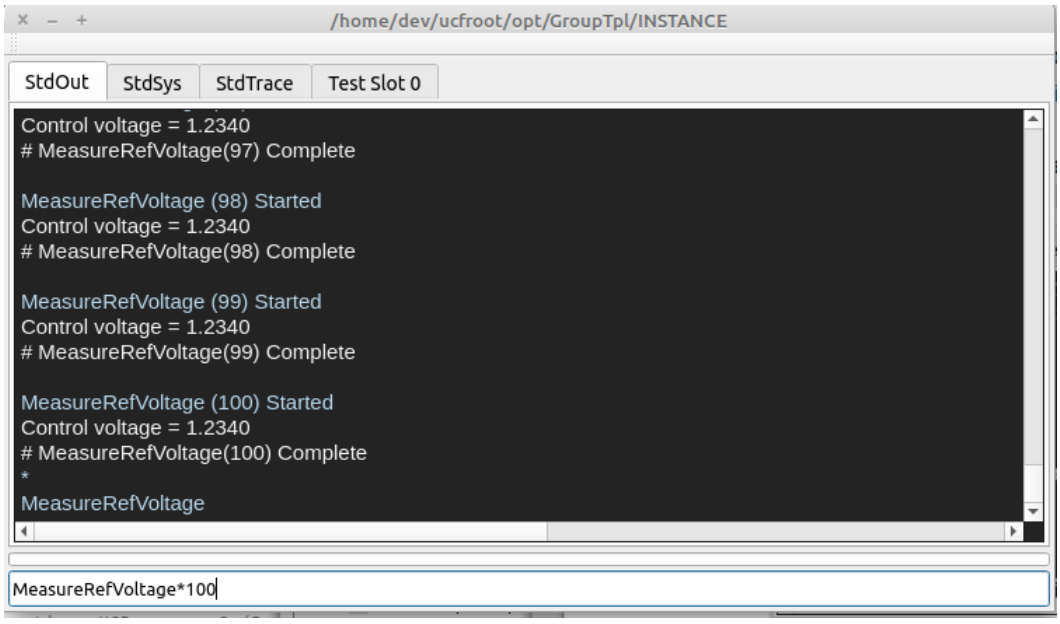
6.1. Viewing Available Commands

Entering ‘help’ and the command line input will show a list of registered commands.



6.2. Executing Utility Commands.

A registered command can be executed by entering the name in the command line input.



- ‘MeasureRefVoltage <cr>’ will execute a single instance of the command
- ‘MeasureRefVoltage * 100 <cr>’ will execute the command sequentially 100 times
- The command line input uses ‘tabbed command completion’ as necessary

7. The Development Process

This section outlines the process of creating a plugin module using the UCF software package.

- A Test profile is Created
- A C++ Software Development Project is Created
- IO Channels are Defined and Initialized
- UCF API Functions are Developed
- UCF API Functions are Registered as Commands or Test Steps
- UUT Measurement Functions are Verified
- UUT Test Sequences are Verified

8. Test Profiles

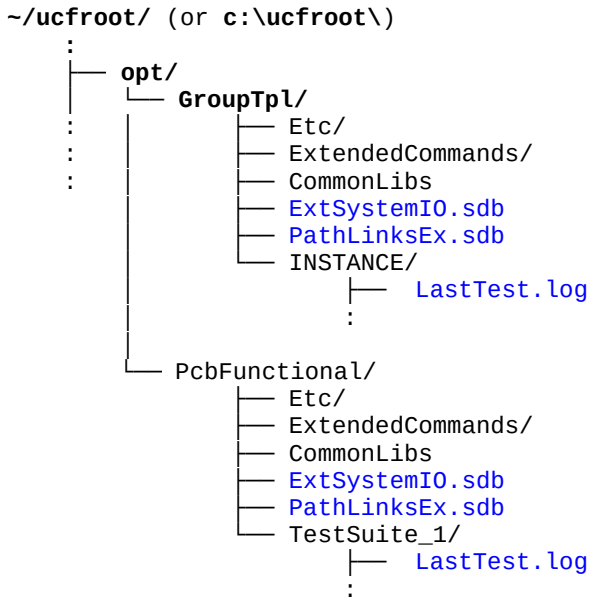
A ‘test profile’ is a subdirectory in a ‘**profile group**’ directory that contains common resources. Common resources include:

- Common utility modules (UCF plugins)
- Instrumentation addressing database
- System variable definition database
- Path-Link definition database

8.1. Creating a Profile Group & Test Profile Structure

A test group directory structure is created using a copy of the sample structure **(./ucfroot/opt/GroupTpl/)** in the UCF distribution.

- The new Profile Group’ is named ‘PcbFunctional’ and the Test Profile ‘INSTANCE’ is renamed to ‘TestSuite_1’



a) Directory and File Descriptions

Directory or Filename	**Description**
./Opt/	The directory containing 1 or more test groups
./PcbFunctional/	A profile group that contains 1 or more test profiles with their common resources
./Etc/	Misc configuration files used by all local profiles
./ExtendedCommands/	External programs and utilities used by all local profiles
./CommonLibs/	Shared libraries that are automatically loaded when executing and local profiles.
ExtSystemIO.sdb	Sqlite database containing system variable and instrument addresses used by all local profiles
PathLinksEx.sdb	Sqlite database containing directory and filename aliases used by all local profiles
./TestSuite_1/'	A unique test profile. This sub-directory is the target for Qt & Visual Studio C++ projects.

9. Software Development Project Creation

This section provides an overview of how to create a test application project using Qt or Visual Studio. The project and support files are found in the distribution directory.

```
./ucfrooot
    └── usr
            └── src
                    ├── TplModule
                    │   ├── CMakeLists.txt
                    │   └── src
                    │               ├── TplModule.cpp
                    │           └── TplModule.hpp
                    │
                    ├── UCF-<Distribution Name>
                    :   └── Api
                    :           ├── ...
                    :           :
```

9.1. Project Creation

The project is created by copying the ./usr/src/TplModule directory to the ***./ucfroot/usr/src/<your project name>*** location.

9.2. Project Configuration

The project is configured by editing the CmakeLists.txt file.

1. Configure the CmakeLists.txt to use the correct directories for all source files.
2. Configure the CmakeLists.txt to create the output in the 'TestSuite_1' profile directory (see section 8).
3. Open the CmakeLists.txt file directly with Qt, or use cmake to create the Visual Studio project.

9.3. Project Verification

1. Verify the project by 'Rebuilding All'.
2. Execute the project output using the following command line syntax:

   '***./ucfroot/bin/UniConsole(.exe) -p ./ucfroot/opt/PcbFunctional/TestSuite_1***'

10. UCF API Functions

UCF API functions are methods that have been registered as either utility commands or Test Executive steps.

Utility commands can be used to interactively control a test station independant of any test sequences.

Test Executive steps are executed as part of a test sequence, or interactively by the Management Console.

10.1. Registering a Utility Function

A utility function is registered in the TplModule.h/cpp files.

a) Example Code

```
int TplModule::Start() {
/* Register the function to be called from the command line */

    if (Cmds) {

        Cmds→RegisterApplicationCmd(
            "MeasureRefVoltage",
            new Ucfx(&TplModule::TestInstrumentControl,this),
            "(range), resolution)\tMeasure a voltage"
        );
    }
    return(1);
}
```

Where:

- "MeasureRefVoltage" is the name of the command
- **TplModule::TestInstrumentControl** is the name of the function to be registered
- "(range), resolution)\tMeasure a voltage" is the description displayed by the help command

10.2. Registering aTest Step

A test step is registered in the TplModule.h/cpp files.

Test steps will appear in the Test Executive (and execute) in the order of registration.

a) Example Code

```
int TplModule::Start() {
/* Register UCF API functions */

    if (TheTestExec) {

        TheTestExec→RegisterTestFunction(
            "GetSn",
            "()\tGet the UUT serial number",
            new Ucfx(&TplModule::Test1, this)
            );

        TheTestExec→RegisterTestFunction(
            "Control Voltage",
            "()\tMeasure the control voltage",
            new Ucfx(&TplModule::MeasureControlVoltage, this)
            );
            . . .
            . . .
    }
    return(1);
}
```

Where

- The first registered test step is to verify the serial number that was input, or read from the UUT
- **"Control Voltage"** is the name of a test step to be registered that will call TplModule::MeasureControlVoltage

10.3. Creating a UCF API Function

UCF API functions are used to create either utility or test functions. API functions

- May be executed by the Test Executive
- May be executed from the command line input.

a) Example Code

```
int TplModule::MeasureControlVoltage(CmdParam msg) {
/* Measure a voltage and display the value in the StdOut tabbed window */

    double measurement_value {1.234};
    /* The variable is for demo purposes only */

    iUcfTestExec exec(msg,ModuleId);

    measurement_value = iSystem->MeasureControlVoltage();
    /*  Demo code that will execute when invoked by the console
        and test executive
        */

    if (exec.IsValid()) {
    /*  Code that will execute only when invoked by
        1. The test executive
        2. The 'start-test' command
        */

        exec.SaveMeasurement("Test A","IsValid",measurement_value,1.0,3.0);
        /*  Save a measurement to the database and set the
            initial pass/fail limits in the test limit database on the first call
            1. 1.0 is the low test limit (value >= 1.0)
            2. 3.0 is the high test limit (value<=3.0)
            */

        exec.WriteTestOuput("A message",  <Option>);
        /* Display messages to the current test slot tabbed console output */

    }
    else {
    /* Code that will only execute when invoked by the console */

        StdOutConsole.Write(
            IOString("Control voltage = %2.4f",measurement_value), <Option>
            );
            /* Write a message to the StdOut tabbed console output */
    }

    return(1);
    /* '1' is always returned */
}
```

b) Additional **iUcfTestExec** methods

The iUcfTestExec object contains the following methods:

- ```
  iUcfTestExec* exec.GetMfgId(
        unsigned &id, imx::IOString &name,
        imx::IOString &location
        );
        /*  Get the manufacturer ID from the configuration database */
  ```

- ```
  iUcfTestExec* exec.SetSn(const char *sn);
   /*   Set the serial number for the current UUT */
  ```

- ```
  const char* exec.GetSn();
   /*    Get the serial number of the current UUT that was input using
         the test executive dialog
         */
  ```

c) **Additional SaveMeasurement(…) options**

- ```
  exec.SaveMeasurement("Test B","IsValid",measurement_value,1.0,3.0,false);
    /*  Save a measurement to the database and use the passed test limits
        only.
        The test limit database values are ignored
        1. 1.0 is the low test limit (value >= 1.0)
        2. 3.0 is the high test limit (value<=3.0)
        */
  ```

- ```
  exec.SaveMeasurement("Test A","IsValid",measurement_value,15);
    /*  Save a measurement to the database and set the pass/fail limits in
        the test limit database as a percentage of the measurement value
        on the first call.
        1. measurement_value-15% is the low test limit (value >= 1.0)
        2. measurement_value+15% is the high test limit (value<=3.0)
        */
  ```
  ```

11.IO Channels

IO channels are a software structures that define the target and any relevant communication parameters.

The UCF uses IO channels to communicate with external instrumentation and target UUT's.

11.1. Defining IO Channels

Definitions for all IO connections are defined in an embedded Sqlite database.

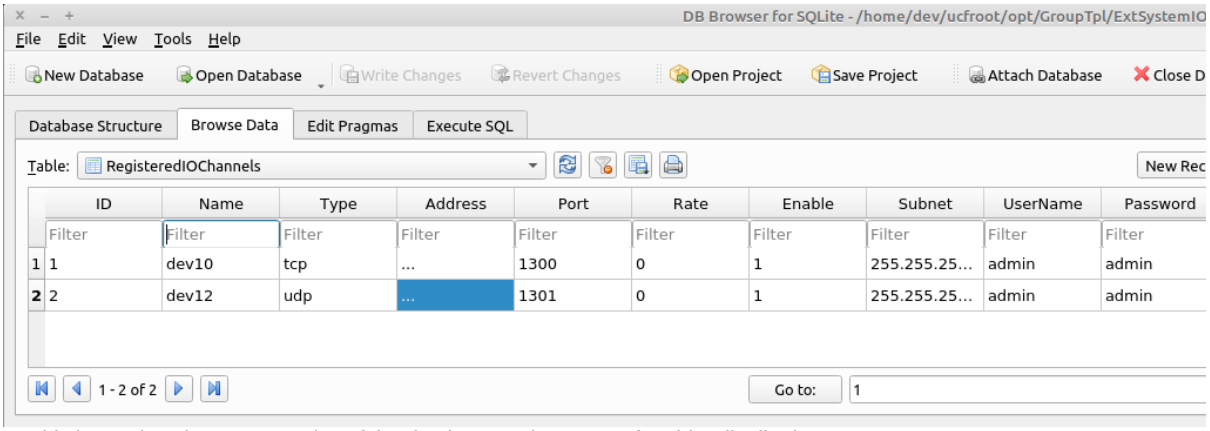
a) Database Locations

The configuration databases are located in the following directories

- **~/ucfroot/bin/<UCF Release Name>/IOConfig/BaseSystemIO.sdb** contains address information that is global to all profile groups.
- **~/ucfroot/opt/PcbFunctional/ExtSystemIO.sdb** contains address information that is local to the 'PcbFunctional' profile group.

b) Defining Communication Parameters

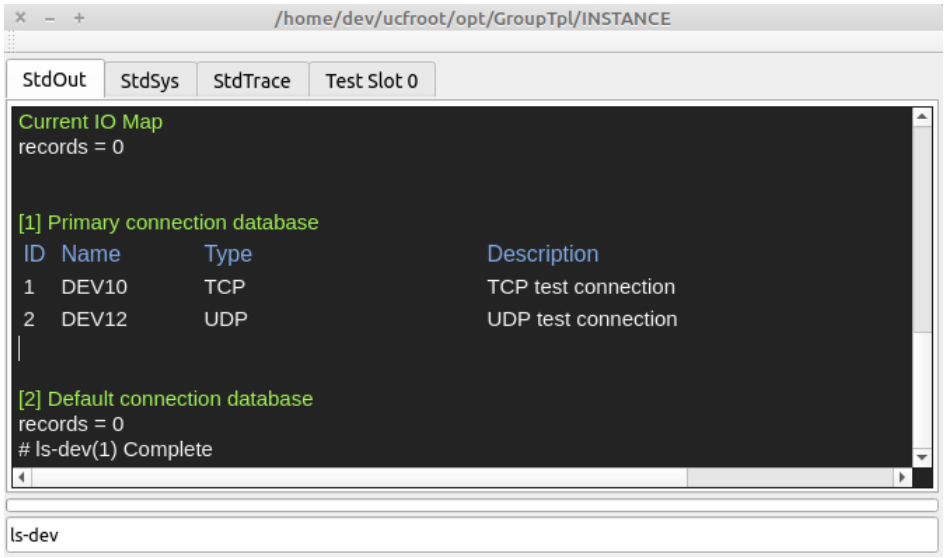
The DbBrowser for Sqlite application is used to define instrument address information.



** This image is only representative of the database and may vary for older distributions

c) **Viewing** IO Channel Definitions

The contents of both databases can be viewed using the 'ls-dev' command.



11.2. Creating IO Channels

IO channels are created on system startup and de-allocated on program exit.
Creating an IO channel is not the same as ‘opening’ a connection.

a) Code Creation Location

Connections are created in TplModule::SetStaticConfiguration(…) { . . .}

b) Creating Instrument IO Channels

TheSystem→RegisterIOChannelDevice(int device_id, char *description) is used to create instrument connections that are required to succeed for test station operation.

- The connection will be verified on creation (ethernet targets will be pinged, etc).
- The connection will be viewable using the Management Console.

Where:

- int device_id corresponds to the name (dev0-n) in the adress configuration database
- char *description is the description that will be displayed when executing the ‘ls-io’ command
- An iIOConnectionDevice* will be returned on success, and nullptr on failure

c) Creating UUT IO Channels

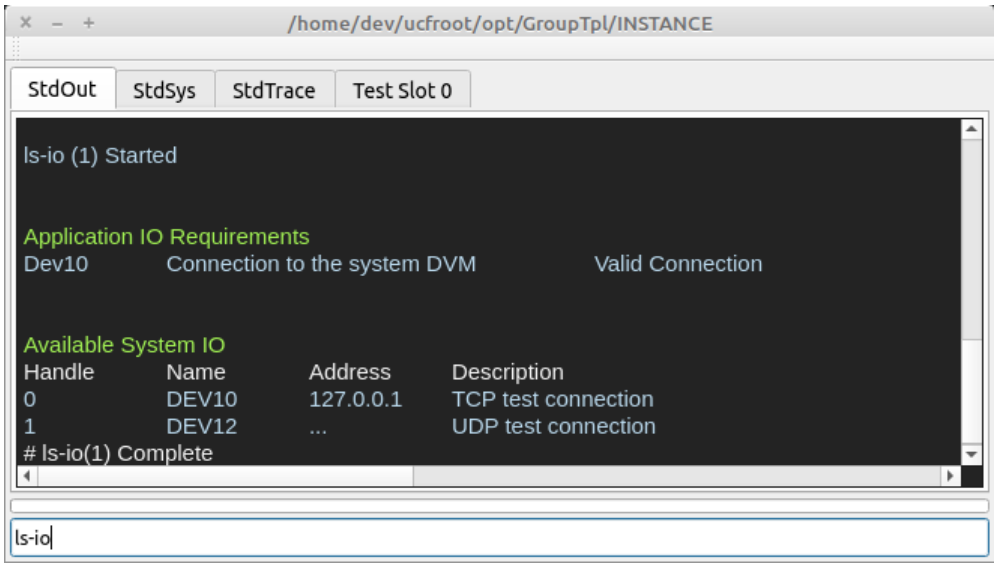
TheSystem->GetTmpIOChannelDevice(int device_id) is used to create an IO channel that will be used to communicate with a UUT.

Where:

- int device_id corresponds to the name (dev0-n) in the adress configuration database
- An iIOConnectionDevice* will be returned on success, and nullptr on failure

11.3. Viewing Instrument IO Channels

Registered IO channels can be viewed using the ‘ls-io’ command.
UUT IO channels will not be displayed.



a) Example Code

```
iIOChannelDevice *DvmIO { nullptr };
iIOChannelDevice *UutIO { nullptr };

void TplModule::SetStaticConfiguration() {
/*  set the static configuration for the system
    1.  assign system interfaces to individual instruments
    */

    DvmIO=TheSystem→RegisterIOChannelDevice(10,"Connection to the system DVM");
    /*  Create a connection to an external instrument
        1.  The database entry for ‘dev10’ is used to create the connection
        2.  “Connection…” is the description that will be displayed when using
            the ‘ls-io’ command
        3.  nullptr is returned on failure
        4.  For ethernet connections, the target is ‘pinged’ to verify
            the connection
        */
    UutIO=TheSystem.GetTmpIOChannelDevice(13);
    /*  Create a connection to the UUT
        1.  The database entry for ‘dev13’ is used to create the connection
        2.  nullptr is returned on failure
        4.  The UUT does not have to be connected when calling this function
        */
}
```

12. External Device Control

IO channels are used to write to external instruments and target UUT's, and read back results when available.

Operations include

- Initialization
- Written to/read from
- Configuration to an idle state on test sequence end, and program shutdown

12.1. IOChannel Open/Close Guard

IO channels can be automatically opened at the start of a function, and closed when the function ends or an exception is thrown.

- **iIOChannelDevice::Guard dvm_guard(iIOChannelDevice *device)** will open 'device' on declaration and close it when 'dvm_guard' goes out of scope.

12.2. Initialization

Instrumentation is initialized and configured at startup to minimize test times and to identify any related system issues as soon as possible.

a) Initialization Code Location

Code is inserted in the TplModule::ThreadedStartup(…) function which executes on startup, in a seperate thread in order to keep the user interface responsive.

- The progress bar will be active while the 'ThreadedStartup(…)' function is executing
- Real-time status information will be displayed in the Management Console

b) User Notification on Completion

The operator should be notified when system initialization is complete using a call to **TheShell->UserSync(…).**

c) Exceptions

All exceptions that are thrown in the thread are caught at the end of the ThreadedStartup(…) and will lock the Test Executive to prevent test operations after a failed instrument configuration process.

- The system locked state can be cleared using the 'lock,0' command.
- The current system lock status can be viewed using the 'lock' or 'status' command

d) Example Code

```
void TplModule::ThreadedStartup(iIXmgr *ix) {
    try {
        TheAppEngine->iErrorManager()->LockEngine(true);
        TheAppEngine→IncrementActiveThreadCount();
       iIOChannelDevice::Guard dvm_guard(DvmIO);
       /* Open the connectin and throw an exception on failure */
       DvmIO->Write (“CONF:VOLT:DC 10,.001);
       /* Set the full scale range and resolution */
       ...
       ...
       TheShell→
       UserSync(true,false,"Initialization is Complete","Select ‘OK’ to continue");
       /* Notify the operator that initialization is complete */
       while(TheShell->GetButtonSelection()==false) {
       /* loop until the 'OK' button is selected */
           std::this_thread::sleep_for(std::chrono::milliseconds(1000));
       }
       TheShell->UserSync(false);
       /* Hide the notification */
        TheAppEngine->iErrorManager()->LockEngine(false);
    }
    catch(IOString msg) {
        TheAppEngine->iErrorManager()->SetHardError(msg);
    }
    catch(...) {
        TheAppEngine
           →iErrorManager()
           →SetHardError("Unknown exception while initializing the system");
    }
    TheAppEngine->DecrementActiveThreadCount();
}
```

12.3. Instrument & UUT Control

Instruments can be controlled by UCF API functions using registered & tmp connections.

a) Example Code

```
int TplModule::ReadDvm(CmdParam msg) {
/* Configure and control an Agilent 34461 DMM */

    std::string response;
    float measurement_value {0};

    try {

        iIOChannelDevice::Guard dvm_guard(DvmIO);
        iIOChannelDevice::Guard uut_guard(UutIO);
        /*  Open instrument & UUT connections
            1.  An exception will be thrown if the connection has not been defined
                or cannot be opened
            */

        dvm_guard->Write (“READ?”);
        TokenString instrument_result(dvm_guard->GetReadBuffer(),',')
        /* Read the input voltage */

        instrument_result.GetToken(1,measurement_value);
        /* Parse the result */

        StdOutConsole.Write(
            IOString("Control voltage = %2.4f",measurement_value),
            iStdOut::Text1
            );
            /* Display the result to the StdOut console */

        uut_guard->Write (“<command>”);
        TokenString result(uut_guard->GetReadBuffer(),',')
        /* Write a command to the UUT and read the result */

        result.GetToken(1,response);
        /* Parse the response */

        StdOutConsole.Write(
            IOString("Command response = %2.4f",response.c_str()),
            iStdOut::Text1
            );
            /* Display the response to the StdOut console */
    }

    catch(iIOChannelDevice::Guard connection) {
    /* If dev13 does not exist in the configuration database */

        StdOutConsole.Write(connection→GetErrorMsg()), iStdOut::Error);
    }

    catch(iIOChannelDevice::Guard connection) {
    /* If the connection could not be opened */

        StdOutConsole.Write(connection→GetErrorMsg()), iStdOut::Error);
    }

    catch(iIOChannelDevice::Guard connection) {
    /* Any other exception*/

        StdOutConsole.Write(“An . . . error occurred”), iStdOut::Error);
    }

    return(1);
}
```

12.4. UUT Configuration on Test Session End

In many cases, a UUT must be set to an idle state at the end of a test session.

a) Example Code

```
int TplModule::ManageTestSequences(TestXtor xtor) {
/*  manage sequence and session construction/destruction
    throw(IOString msg) to terminate the sequence
    */

    int ret {1};

    switch(xtor.GetXtorType()) {

        case(TestXtorType::SequenceCtor):
        /*  Execute at the start of a test sequence */
            break;

        case(TestXtorType::SessionCtor):
        /*  Execute at the start of a test session */
            break;

        case(TestXtorType::SessionDtor):
        /*  Execute at the end of a test session */

            /*******************************
             Insert custom code here
            *******************************/

            break;

        case(TestXtorType::SequenceDtor):
        /*  Execute at the end of a test sequence */
            break;

    }

    return(ret);
}
```

12.5. Instrument Configuration on Shutdown

Instrumentation should be set to an ‘idle’ state on program termination.

The TplModule::Shutdown() function is automatically called on program termination.

a) Example Code

```
void TplModule::Shutdown() {
/*  Shutdown the lib on program termination
    1.  Set any instrumentation to an idle/safe state
    2.  De-allocate as necessary
    */

    iSystem→DisableSystemPowerSupplyOutput(…);
    /*  This is a demo function to disable the system power supply’s output */
}
```

13. Plugin Modules

The UCF framework is designed to use plugin modules that export virtual interfaces to be used by other modules.

- Instrument control modules can be updated without requiring recompilation of UUT test code
- Test algorithm modules can be updated without requiring recompilation of system control code
- Multiple stations that use slightly different instrumentation can execute the same test code without requiring re-writes or re-compilation.

13.1. Creating a Plugin Module Project

A plugin module project is created a software development project as outlined in section 9.

- The output of the project will be in the ***./ucfroot/opt/PcbFunctional/CommonLibs/*** *directory*

13.2. Creating an Export Interface

An export interface is a pure virtual interfaces to an object that is instantiated in the plug-in module.

a) Example Code

```
class iSystem {
    public:
    virtual void ConfigureVoltageSource(iIOConnectionDevice *ps)=0;
    /*  Set the IO Channel connection to the system voltage source
        */
    virtual double SetOutputVoltage(double voltage, double current_limit)=0;
    /*  Set the system voltage source output voltage and current limit
        1.  Return the current output
        */
    virtual ~iSystem() {}
};
class System:public iSystem {
    public:
    void ConfigureVoltageSource(iIOConnectionDevice *ps) override {
    /*  Set the IO Channel connection to the system voltage source
        */
        ThePowerSupply=ps;
    }
    double SetOutputVoltage(double voltage, double current_limit) override {
    /*  Set the system voltage source output voltage and current limit
        1.  Return the current output
        */
        :
        :
        return(output_current);
    }
    ~System() {…}
    private:
    iIOConnectionDevice *ThePowerSupply {nullptr};
};
```

13.3. Exporting Interfaces

Interfaces are exported to the framework by registering a virtual pointer in the GetEntryPoint(…) function.

Once exported, a virtual interface to 'TheSystem' can be imported into any number of additional modules.

a) Example Code

```
System TheSystem;

iSharedLibFramework<iIXmgr>* GetEntryPoint(*local_ix, unsigned module_id) {
/*  The module is entered here and a reference to the lib's framework is returned
        1.  register all exported interfaces into the module-local repository
        2.  register all exported commands into the module-local repository
        3.  'ix' is the interface manager local to this module
    */

    local_ix->iSet<iSystem>(&TheSystem);
    return(&TheModule);
}
```

13.4. Importing and Using Interfaces

Virtual interfaces that have been exported can be imported and used in any module.

a) Example Code

```
#include <iSystem.hpp>
/* The declaration of the iSystem virtual interface */

iSystem *TheCurrentSystem {nullptr};

int TplModule::Initialize(
    iSharedLibFramework<iIXmgr>::ModuleStates level,
    iIXmgr *global_ix
    ) {
    /*  Initialize the module
    1.  use the interfaces previously registered by the framework and other modules
    2.  the 'ix' interface provided to this function is the global interface
    */
    using state=iSharedLibFramework<iIXmgr>::ModuleStates;
    try {
        switch(level) {
             case(state::GetInterfaces):
             /* use the interfaces previously registered by the framework
                and other modules
                */
                TheCurrentSystem=global_ix->iGet<iSystem>();
                break;

            :

            :
           }
    }
    catch(...) {

            HandleModuleException("Initialization");
    }

    return(1);
}
```

```
int TplModule::TestFunction(CmdParam msg) {
/*  Make a call to the 'System' object that was instantiated in another module */

    TheCurrentSystem→SetOutputVoltage(5.0, 3.5);

    return(1);

}
```

14. Validating Measurement Accuracy

Measurement accuracy must be validated and be within the accuracy and repeatability requirements of the Test Specification.

14.1. The Process

1. Create utility command for each system measurement function that displays the measurement value.
2. For each utility command, execute 100 times using the Management Console ('command*100).
3. Copy and past the results from the StdOut console into a spreadsheet (and format as needed).
4. Calculate min/max/mean values, and determine if the measurements will be satisfy the needs of the test specification.

15. Validating Test Sequence Reliability

Test sequence reliability must be validated in order to be released to production.

15.1. The Process

1. Execute the 'start-test*100' command in the Management Console to test a passing UUT.
2. Verify the accuracy of all measurements
3. Verify that there are no false failures.
4. Execute the 'start-test*100' command in the Management Console to test a failing UUT.
5. Verify the accuracy of all measurements
6. Verify that there are no false 'passes'.
7. Repeate steps 1-6 for 5 passing UUT's and 5 failing UUT's.
8. For all steps 1-7, verify that there are no program crashes or unexpected behavior.

16. Misc Development Topics

16.1. Writing to Management Console Tabs

The following methods are used to write to Management Console Tabs.

- StdOutConsole.Write("A message", <option>);
  /* Write to the StdOut tab */
- StdSysConsole.Write("A message", <option>);
  /* Write to the StdOut tab */
- StdTraceConsole.Write("A message", <option>);
  /* Write to the StdOut tab */

Where <option> is defined as one of the following values:

```
iStdOut::Pass              // green text
iStdOut::Warning           // yellow
iStdOut::Error             // red text

iStdOut::SystemSoftError   // yellow text
iStdOut::SystemHardError   // dark red text

iStdOut::Text1             // normal text
iStdOut::Text2             // normal text-1 point size
iStdOut::Text3             // normal text-2 point size
iStdOut::GrayText          // gray text
iStdOut::ItalicText        // white italic text

iStdOut::Directory         // blue text
iStdOut::Filename2         // light blue text
iStdOut::Filename3         // yellow text
iStdOut::Filename3         // white text
```

16.2. User Notifications

User notifications are used to display messages to the user. Program execution can be halted until the ‘OK’ button is selected, or until a polled event occurs.

- The notification will be displayed on top of all other windows and is thread-safe.
- Only one notification can be displayed at the same time.
- The title label is programmable.
- Any number of lines can be displayed. Scroll bars will be displayed as necessary.
- The message window will resize based on the number of message lines (1-20).

a) Example Code

```
bool is_an_error {false};
bool open {true};

TheShell→ UserSync(true,is_an_error,"Title","message line1\n message line2\n");
/* Display the notification */

while(TheUutIsNotCalibrated()) {
/*  loop until the operator has completed a manual step, that is verified by
    the call to ‘TheUutIsNotCalibrated()'
    -   Exit when the 'OK' button is selected
    */
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    if (TheShell->GetButtonSelection()==true) break;
}

** or **

while(TheShell->GetButtonSelection()==false) {
/* loop until the 'OK' button is selected */
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
}
TheShell->UserSync(open=false);
/* Hide the notification */
```