NOKIA TSM V0.96

Sponsored in conjunction with The University of Texas at Dallas Senior Design Program.

Developed by Design Team #904

Written by Cody Hatfield

Table of Contents

Introduction

Welcome to the user manual for the Nokia™ Test Suite Manager v0.96. This document will serve as a brief overview on utilizing the Nokia™ TSM as a general purpose automated testing platform, as well as how to extend the platform with additional testing modules. This document is divided into four sections: the Introduction, Quick Start, Architectural Concepts, and Module Development Guide.

The Quick Start gives a brief set of steps to follow to get the TSM up and running, for either demo purposes or to give a general idea of how the application flows.

Architectural Concepts gives a broad look into the theory and concepts behind how the Test Suite Manager is structured, both programmatically and philosophically. The purpose of this section is not to exhaustively study every line of code or examine every file, but to give future potential developers a better understanding on how everything is put together.

The Module Development Guide takes an in-depth look at the most import features of the Nokia™ Test Suite Manager – the extensibility of the TSM via custom test modules. The majority of the application serves solely as a general purpose user-interface and automated sequencer for these modules, but it is the modules themselves which package the data for transmission to the test bench and return the results of the tests back to the application for viewing/saving. As such, a recipe for module development would be most prudent in increasing the lifetime of the application.

In Memory of Tabithat the Tubby Cat.

You will be missed.

Quick Start

This section will describe how to get your Test Suite Manager going out of the box, using an automated installation process and the Demo Suite that comes pre-packaged with application.

System Requirements

To run the Nokia™ Test Suite Manager, you will need to have Python 3.7.0 installed and configured with the tkinter, pyvisa, and pyvisa-py libraries. An installer is included in this TSM distribution, and is located at 'Python/python-3.7.0-install.exe'. An internet connection is required to install the 3rd party pyvisa libraries, while tkinter usually comes default with python.

An automated install script is included with this distribution, which handles the python installation and library dependency downloads automatically. Run the file 'reinstall.bat', and the automated script will do its job. Rerunning the script after an install will result in a fresh reinstall procedure, with the previous python instance being uninstalled automatically.

> **Note:** If you wish to uninstall/reinstall the local python instance, please **do not** delete the files in the Python/ folder manually, rather use the 'reinstall.bat' script. Failing to do so and deleting the instance manually may result in the installer becoming corrupted.
>
> If this occurs, there are four different ways to recover:
>
> a) Replace the deleted files back in the Python/ folder
> b) Redownload/Checkout the git instance of the application (specifically, the Python/ python-3.7.0-install.exe file)
> c) Repair the python instance by manually double clicking on the installer at 'Python/python-3.7.0-install.exe', and then choosing the "Repair" option.
> d) Redownload the original python offline installer from the python main site, and rename it to 'python-3.7.0-install.exe' while also placing the redownloaded installer in the Python/ folder.
>
> If you prefer, you can run and manage your own python instance in the Python/ folder, either with the provided installer or through your own methodology. This automated distribution was provided mainly as a byproduct of the development process.

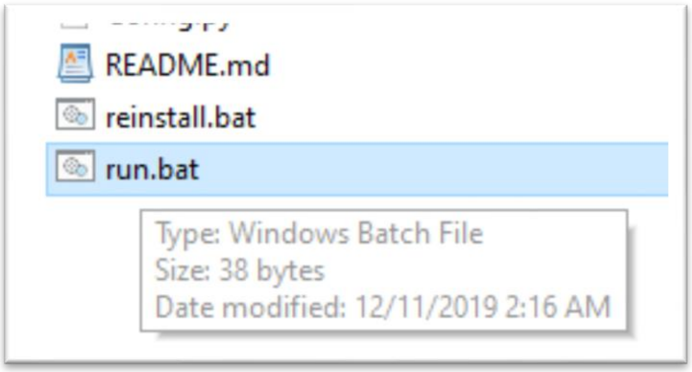
NOTE:

> If you are developing a module, it is recommended that you do so within an IDE. This will grant you access to critical debugging functions, as the exception handling of the application is limited in scope to exceptions that may be encountered during normal runtime operations.

Demo Quickstart

Make sure you have python installed, as detailed in the previous section.

While most of the modules developed thus far for the application were created to connect to test equipment and gather data, some basic dummy modules without these resource requirements were also created to demo the application. This section will walk you through running this demo, optimizing speed over content.

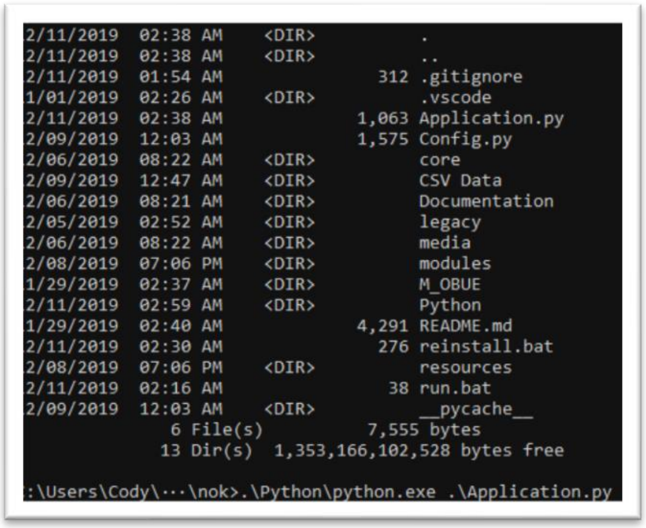


README.md
reinstall.bat
run.bat

Type: Windows Batch File
Size: 38 bytes
Date modified: 12/11/2019 2:16 AM

1. Run Application:

You can open the Test Suite Manager by double clicking the ‘run.bat’ file in the application directory.

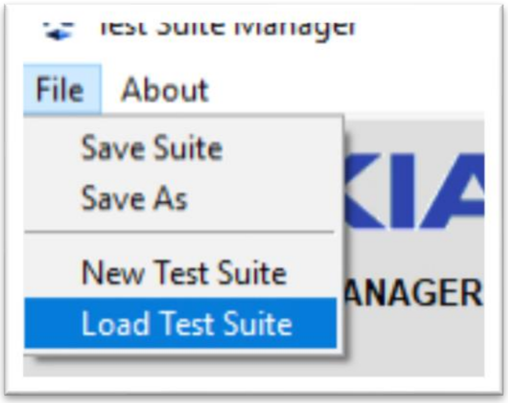


```
2/11/2019  02:38 AM    <DIR>          .
2/11/2019  02:38 AM    <DIR>          ..
2/11/2019  01:54 AM               312 .gitignore
1/01/2019  02:26 AM    <DIR>          .vscode
2/11/2019  02:38 AM             1,063 Application.py
2/09/2019  12:03 AM             1,575 Config.py
2/06/2019  08:22 AM    <DIR>          core
2/09/2019  12:47 AM    <DIR>          CSV Data
2/06/2019  08:21 AM    <DIR>          Documentation
2/05/2019  02:52 AM    <DIR>          legacy
2/06/2019  08:22 AM    <DIR>          media
2/08/2019  07:06 PM    <DIR>          modules
1/29/2019  02:37 AM    <DIR>          M_OBUE
2/11/2019  02:59 AM    <DIR>          Python
1/29/2019  02:40 AM             4,291 README.md
2/11/2019  02:30 AM               276 reinstall.bat
2/08/2019  07:06 PM    <DIR>          resources
2/11/2019  02:16 AM                38 run.bat
2/09/2019  12:03 AM    <DIR>          __pycache__
               6 File(s)          7,555 bytes
              13 Dir(s)  1,353,166,102,528 bytes free

:\Users\Cody\⋯\nok>.\Python\python.exe .\Application.py
```

1b. Alt Run Application

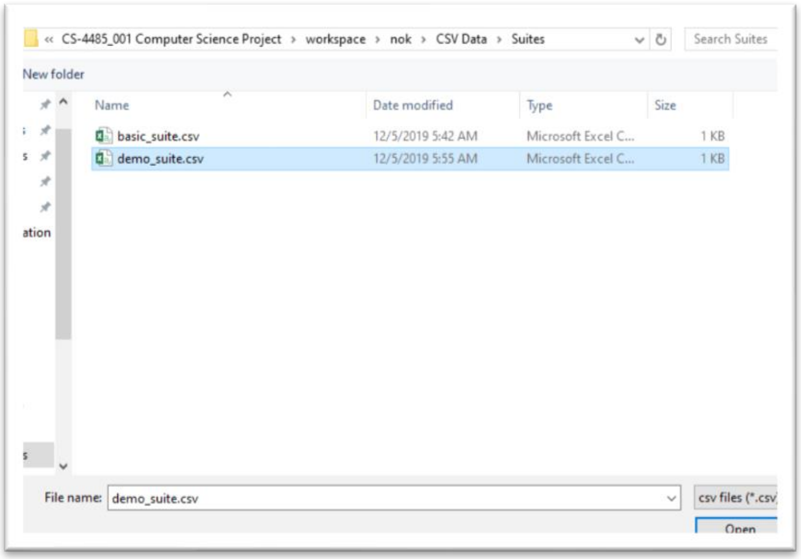
…or you can open the command line and enter:

‘{path-to-python}\python.exe .\Application.py’



2. Open File Browser

With the application open, navigate on the file menu to ‘File’ > ‘Load Test Suite.



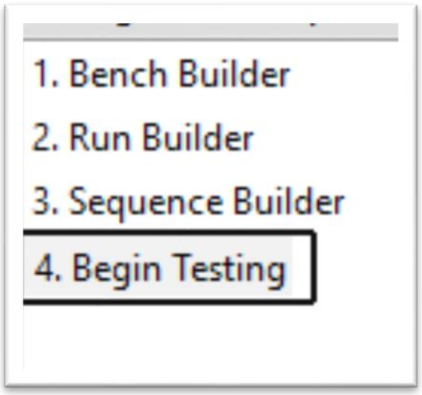
« CS-4485_001 Computer Science Project › workspace › nok › CSV Data › Suites

Name	Date modified	Type	Size
basic_suite.csv	12/5/2019 5:42 AM	Microsoft Excel C...	1 KB
demo_suite.csv	12/5/2019 5:55 AM	Microsoft Excel C...	1 KB

File name: demo_suite.csv
csv files (*.csv

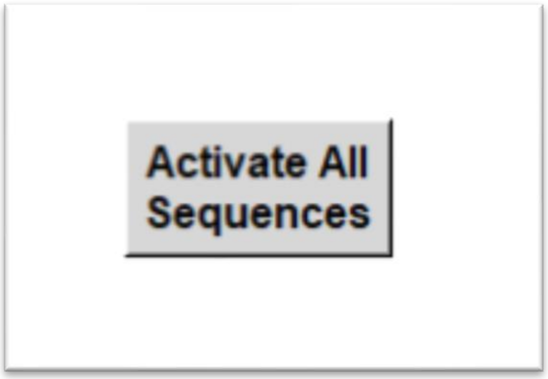
3.Select Suite File

Choose the file ‘demo_suite.csv’ from the folder ‘CSV Data/Suites/’.



4. Navigate

Use the Navbar in the lef column, and select Step 4. 'Begin Testing'

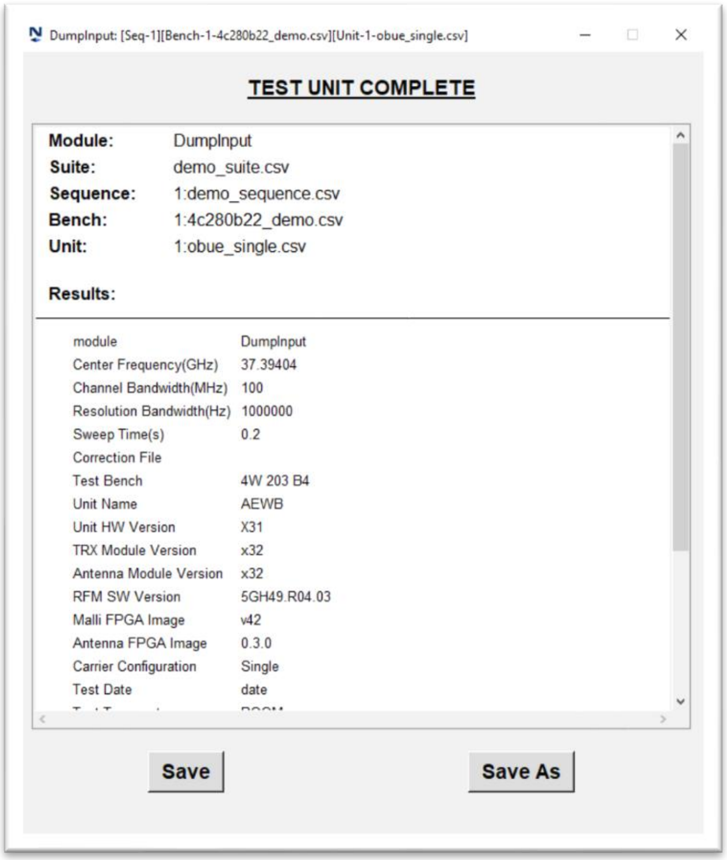


5. Activate

On the Testing page, click on the button titled "Activate All Sequences"

6. Wait

The Demo takes approximately 12 seconds to run.



DumpInput: [Seq-1][Bench-1-4c280b22_demo.csv][Unit-1-obue_single.csv]

TEST UNIT COMPLETE

Module:	DumpInput
Suite:	demo_suite.csv
Sequence:	1:demo_sequence.csv
Bench:	1:4c280b22_demo.csv
Unit:	1:obue_single.csv

Results:

module	DumpInput
Center Frequency(GHz)	37.39404
Channel Bandwidth(MHz)	100
Resolution Bandwidth(Hz)	1000000
Sweep Time(s)	0.2
Correction File	
Test Bench	4W 203 B4
Unit Name	AEWB
Unit HW Version	X31
TRX Module Version	x32
Antenna Module Version	x32
RFM SW Version	5GH49.R04.03
Malli FPGA Image	v42
Antenna FPGA Image	0.3.0
Carrier Configuration	Single
Test Date	date

7. Finished

The test units have finished running, and there should be a results window displayed for the four demo test modules.

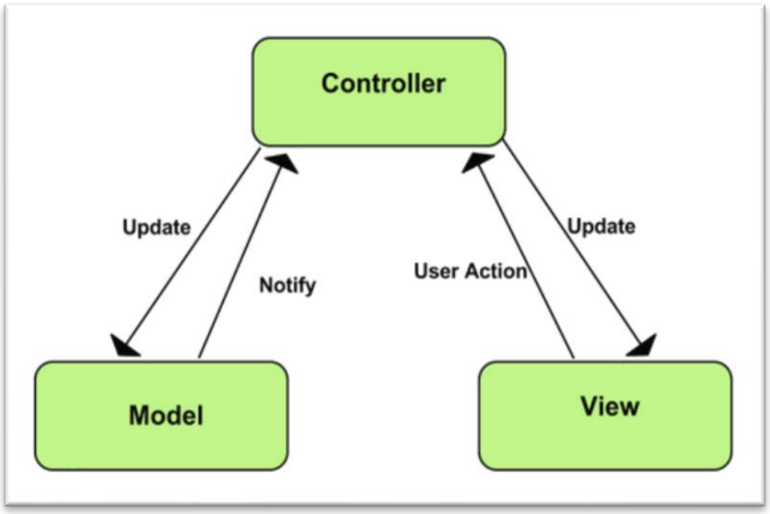
Architectural Concepts

This section will seek to elucidate the philosophical organization of the Test Suite Manager's code, so as to give future module developers a better understanding of how to integrate their test modules with the application.

The TSM is built primarily upon the Tkinter UI library that comes pre-packaged with Python. It is a popular and widely supported library, which allows for ample 3rd party documentation to guide development.

MVC Architecture

The Nokia™ Test Suite Manager is based primarily upon the MVC architecture design pattern. This paradigm defines three types of components: a Model, a View, and a Controller.



The Model serves as a general purpose data structure that can access and represent stored data – any file data that must be read into or written out of the application must first pass through a Model, which enforces the business logic of its data storage mechanism. This creates a standard of interaction that ensures data stability.

The View takes data that is passed to it, and formats it into a presentable interface for the user. It does little if any data processing on its own, as its primary responsibility is presenting data to and getting data from the user.

The Controller directs the flow of the application, serving as a coupling between the Model and the View. It is responsible for creating and implementing the business logic of the application, and handling runtime errors in a graceful manner.

Due to how Tkinter structures itself, the line between the View and the Controller can become blurred, and the components can sometimes merge. This variation of the MVC architecture is occasionally referred to as M/VC architecture, however the general ideas that guide the two are essentially the same.

Care was taken to try and keep the separation between the View and Controller clear, but there were some instances (such as the Builder) where this was simply not an option without descending into a problem of infinite regression. Regardless, it is still useful to view the components as distinct from one another, and the remainder of this documentation will do as such.

Conceptual Building Blocks

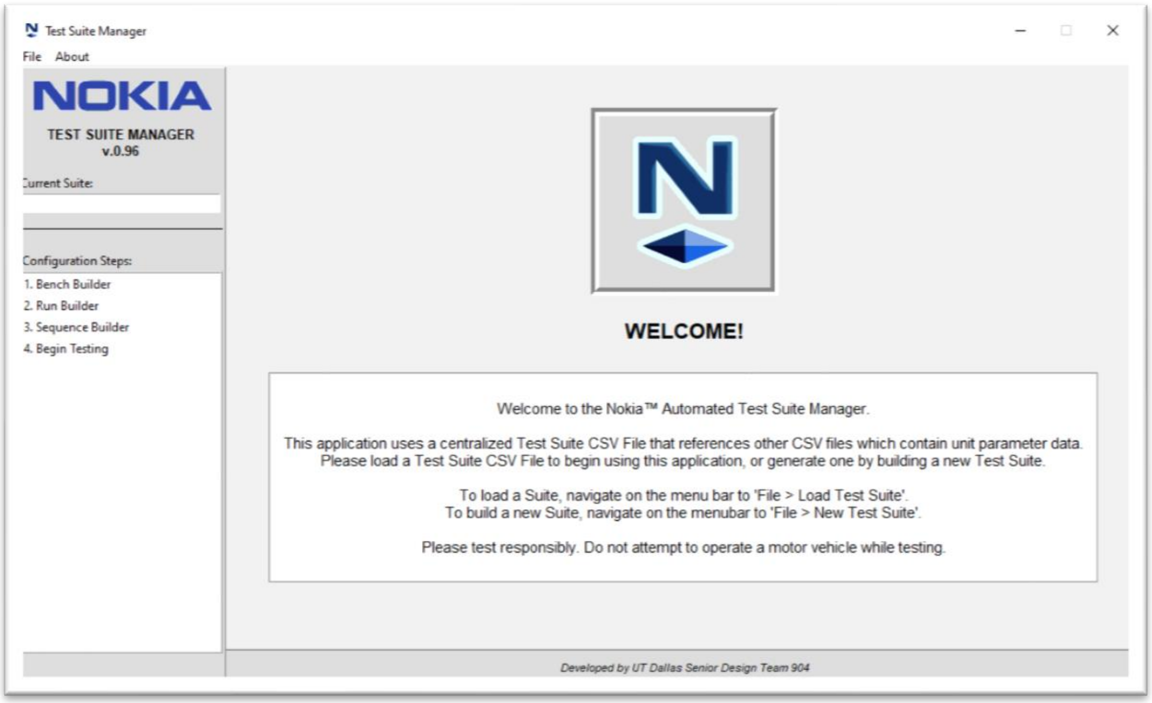
While the MVC architecture is a guiding principle of the application, it serves primarily as a foundation from which the more specific behaviors of the application can arise. These behaviors are expressed using specific configurations of the foundational architecture, implemented into either child or sibling classes which interact with the MVC building blocks.

The result of this is a modular and highly extensible framework that organizes itself into layers of interaction, with specific data pathways defined which dictate the legal ways that objects can interact. There is some deprecated code still present in some of these implementation and class definitions, as it was replaced with more efficient functionality or its responsibility was abstracted away to another class. This deprecated code still functions however, and could potentially be repurposed in the future if it is desired.

Thus, the following sections and subsections will illuminate these layers and their building blocks, while maintaining a top level approach to their descriptions.

Suite Manager

The Suite Manager serves as the root of the application, tying together the various UI elements and facilitating communication between them. It is the first object instantiated by the application upon runtime, and performs rudimentary file-save checking before terminating the application.



Welcome to the Nokia™ Automated Test Suite Manager.

This application uses a centralized Test Suite CSV File that references other CSV files which contain unit parameter data. Please load a Test Suite CSV File to begin using this application, or generate one by building a new Test Suite.

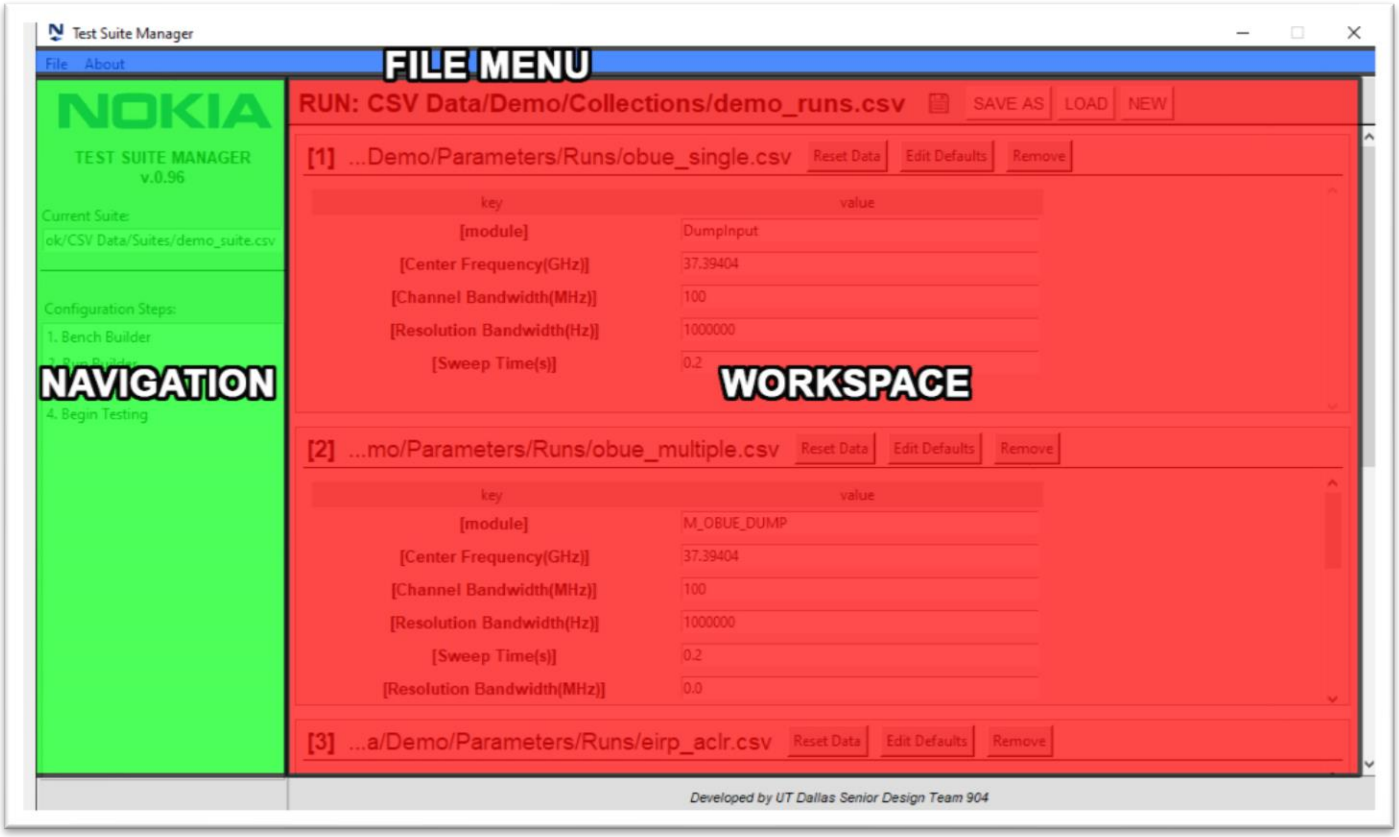
To load a Suite, navigate on the menu bar to 'File > Load Test Suite'.
To build a new Suite, navigate on the menubar to 'File > New Test Suite'.

Please test responsibly. Do not attempt to operate a motor vehicle while testing.

Developed by UT Dallas Senior Design Team 904

The Suite Manager can conceptually be broken down into three main areas of interaction for the user:

- **File Menu:** A small menu bar at the top of the application window which performs save/load operations for the Suite Manager's CSV data. Can also be used to generate a new Suite from the ground up, or display a fun 'About' window.
- **Workspace:** This is the container where a workspace can be added to the application. A workspace includes the various renderers and sequence control fields of the application, as well as any interface frame meant to change, manipulate, or otherwise do something useful with data. The suite manager features four different workspaces out of the box:
    - *Bench Builder – Creates and manages the benches of the loaded suite.*
    - *Unit Builder – Creates and manages the test units of the loaded suite.*
    - *Sequence Builder – Collects the loaded benches and units, and pairs them to create test sequences.*
    - *Begin Testing – The sequencer control page, which can be used to activate a sequence and begin testing.*
- **Navigation:** A navigation column that displays information about the currently loaded suite, and provides buttons to switch between the various workspaces.



Current Suite:
ok/CSV Data/Suites/demo_suite.csv

Configuration Steps:
1. Bench Builder
4. Begin Testing

RUN: CSV Data/Demo/Collections/demo_runs.csv

[1] ...Demo/Parameters/Runs/obue_single.csv

key	value
[module]	DumpInput
[Center Frequency(GHz)]	37.39404
[Channel Bandwidth(MHz)]	100
[Resolution Bandwidth(Hz)]	1000000
[Sweep Time(s)]	0.2

[2] ...mo/Parameters/Runs/obue_multiple.csv

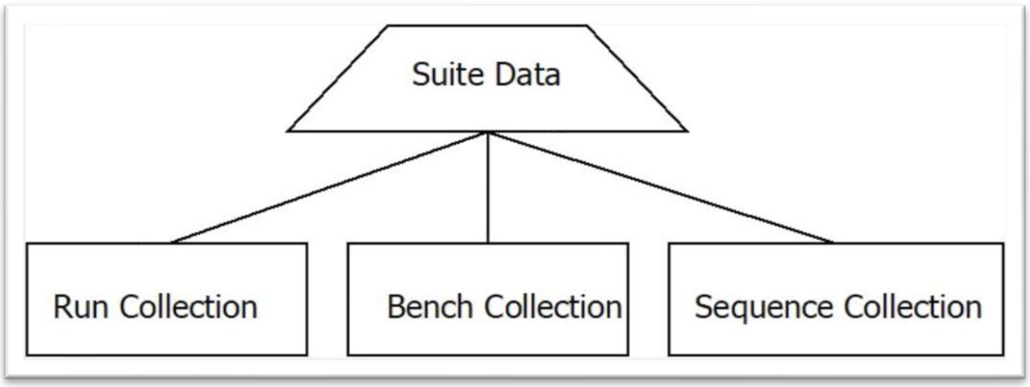
key	value
[module]	M_OBUE_DUMP
[Center Frequency(GHz)]	37.39404
[Channel Bandwidth(MHz)]	100
[Resolution Bandwidth(Hz)]	1000000
[Sweep Time(s)]	0.2
[Resolution Bandwidth(MHz)]	0.0

[3] ...a/Demo/Parameters/Runs/eirp_aclr.csv

Developed by UT Dallas Senior Design Team 904

The Suite Manager contains an internal model called a Suite Model, which it uses to save its data. Whenever the application updates itself, this model must be saved to ensure that the data persists.

The Suite model contains three strings populated with paths to CSV files. Each of the three CSV file paths point to a saved "CSV Collection", also just called a Collection, who's functioning is explained later. All that is currently necessary to know, is that the Suite model must refer to a saved Collection file for each builder in order for the Suite Manager to activate the test sequence.

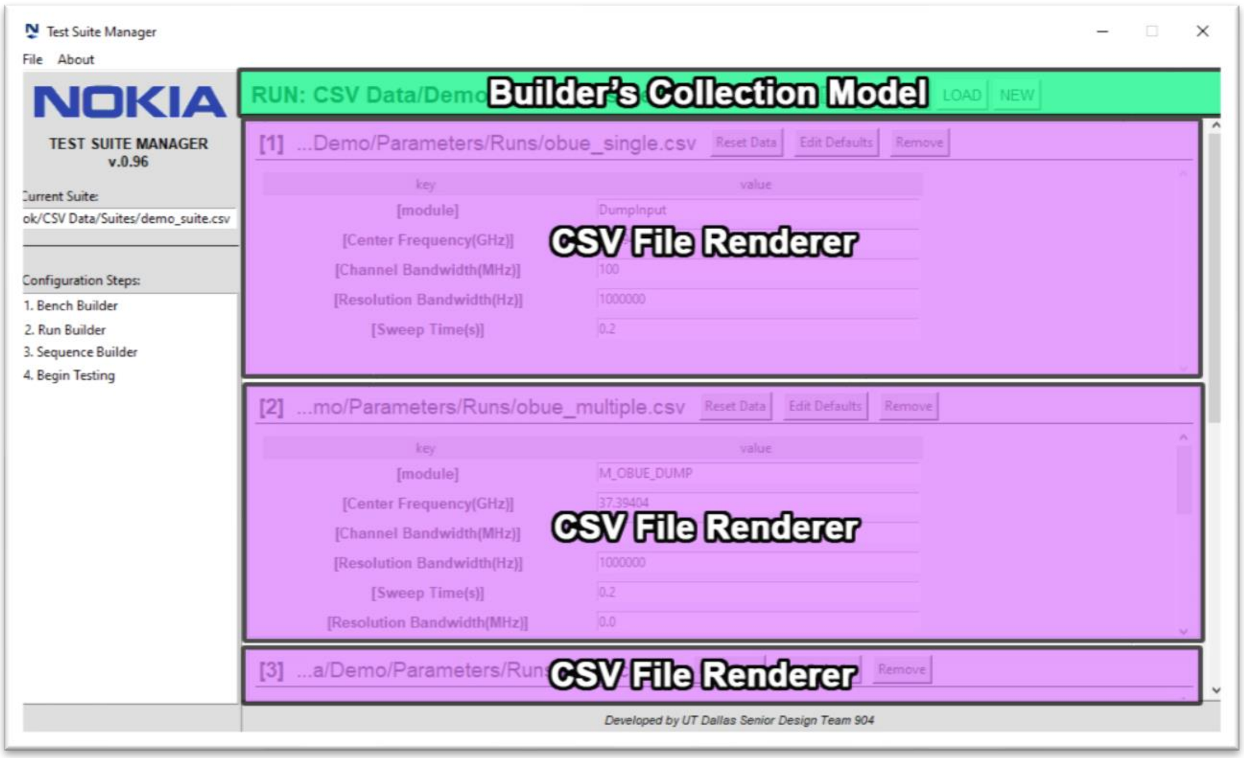


Builder

A Builder is an interface frame which can load, manipulate, and return multiple CSV files' data to the application when called upon. It uses the Suite Model data to load itself with its corresponding CSV Collection, which is used to load an arbitrary series of CSV files into the Builder.

Once the files are loaded into the Builder, it feeds their data into a series of CSV renderers before populating itself with the rendered data.

All Builders define a universal "compileData( )" method in their class definition, which is used to collectively gather all data currently loaded in the application.



key	value
[module]	DumpInput
[Center Frequency(GHz)]	[illegible]
[Channel Bandwidth(MHz)]	100
[Resolution Bandwidth(Hz)]	1000000
[Sweep Time(s)]	0.2

key	value
[module]	M_OBUE_DUMP
[Center Frequency(GHz)]	37.39404
[Channel Bandwidth(MHz)]	[illegible]
[Resolution Bandwidth(Hz)]	1000000
[Sweep Time(s)]	0.2
[Resolution Bandwidth(MHz)]	0.0

Model

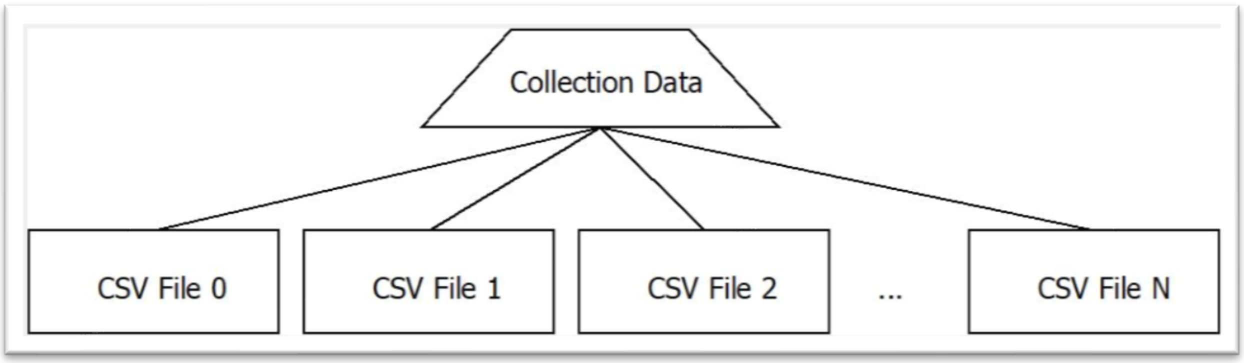
Following the archetype set by the MVC design pattern, a Model fetches and saves data for other classes in the application. It maintains a path reference to its CSV file for future use, and contains a list of dictionaries which represent the data of the file. This data can be saved from/loaded into at will by the Model, with child classes extending this functionality.

Suite Model

The Suite Model stores the data used by the Test Suite Manager. It is simple, and composed of only three rows and two columns. Each row is a relative path to a Collection CSV file stored elsewhere in the application directory, with each Collection corresponding to one of the Builders instantiated in the Suit Manager. One of the columns is the key that indexes the path.

Collection Model

The Collection is a model used by the Builder, and it is the crux of its power – because the application is reading/writing exclusively to CSV's, there is no opportunity to store multi-dimensional data within the files used by the application. All data that is saved must be flat, and attempting to extend the UI markup language to support structured data would have left the CSV files unreadable by humans.



Thus, the Collection serves as an intermediary file, which points to N other CSV files that contain the actual desired data. This allows the Suite Manager to specify a finite number of collections for its builders, configure the builders to use these collections, and in return the builders can accommodate a near-infinite number of references to other stored data structures of different types. This allows for the enforcement of Soft Data Structures, which are explained later.

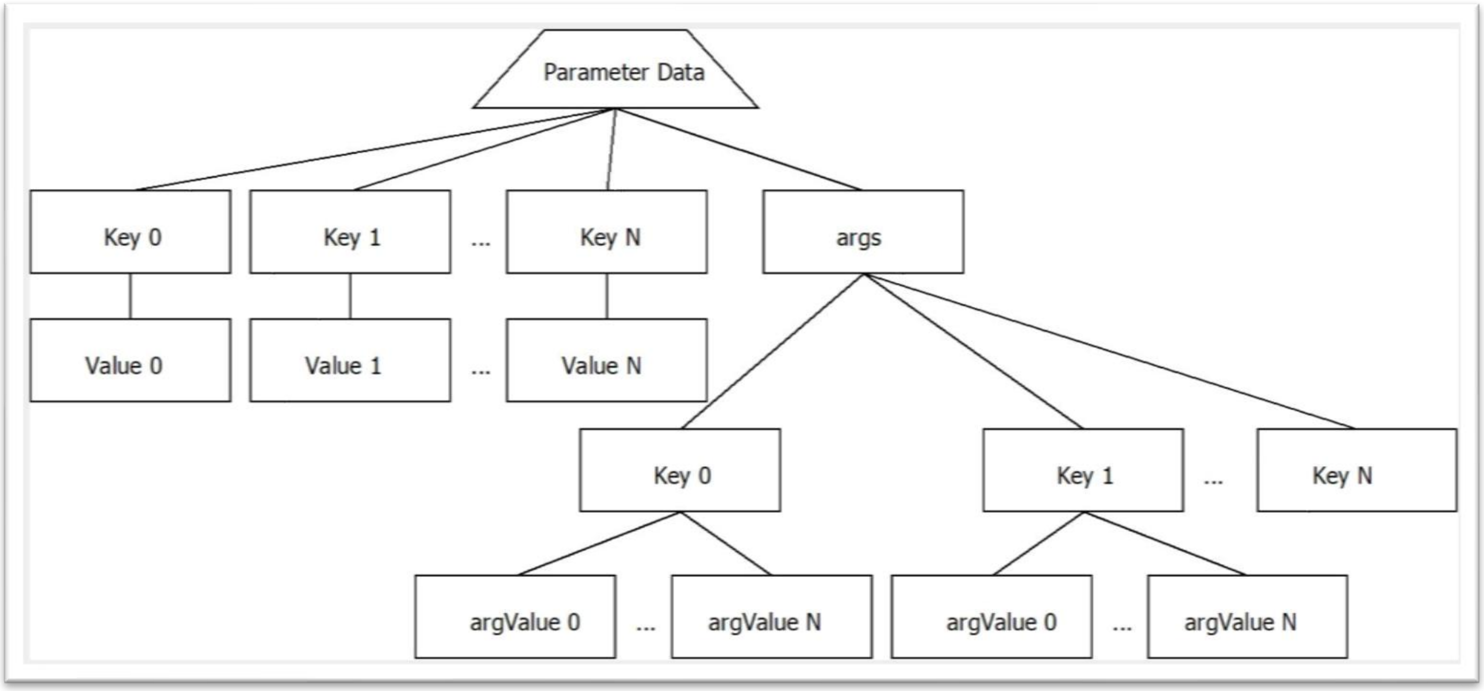
Parameters Model

The Parameter Model is a Model used by the CSV Render/Editor to utilize individual parameters CSV files. A parameters CSV file is the most fundamental unit of storage in the application, as it is where the actual data points which are sent to the test module are defined.

The Parameters Model has the same flat data element as the regular Model, but also supports an additional dictionary, which the Parameters Model attempts to populate with extracted indexed data from the flat CSV data.

Because CSV data cannot be guaranteed to be indexable, it support optional configurations to ensure a direct set of column pairs from the flat data is used as key/value pairs. If this configuration is not present, then the Parameters Model attempts to make a best guess.

If there are more than two columns of data in the CSV file, then the Parameters model will store the extra data from each row in an additional args data member, which is structured in such a way that the additional row data for an indexed row can be extracted. If there is no extra row data, then args is empty.



The Parameters Model is used to save/store the data for each individual Bench and Unit. While some of the indexing capabilities are now deprecated, the Parameters Model offers a way to quickly load data and get it indexed if you know that the data is indexable to begin with. The addition of the args data member further extends the applicability of the Paramerters Model, as they allow for the setting of additional flags or other necessary values on a row-by-row basis for each data point.

Note on Sequence Data

Sequences are not implemented using the Parameters Model however, and in fact are not given their own specialized model at all. Their data is simple pairings between Benches and Units, and thus the parent Model class is perfectly acceptable to use as a structure to represent the sequence data.

Due to this, the sequence data can be easily extended in the future by modifying the configuration values in the Sequence Builder. One potential avenue to explore is storing average runtimes for a sequence pair in the data, so that the application can predict how long the currently selected sequence will take.

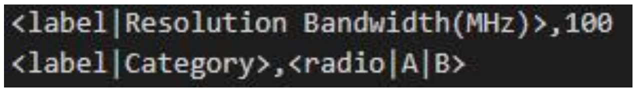
Interface

An interface is analogous to the view component of the MVC architecture. Any frame of interaction can be considered an interface, with interfaces capable of using other interfaces as their root frame or instantiating their own self-contained pop-up window.

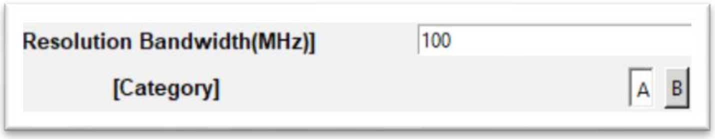
There is an Interface class structure defined in the core/ directory of the source code. Not every interface is directly related to this class, but they all share similar properties with it.

Input Factory & Widgets

Whenever a CSV file is loaded by a Builder, its path is passed to a CSV renderer which reads all text data at the path and builds a formatted representation of said data based on a simple markup language defined within the application.



```
<label|Resolution Bandwidth(MHz)>,100
<label|Category>,<radio|A|B>
```



The intended purpose of the markup language and the rendered formatting is the creation of "guided" test suites, where bench and test unit parameter data can be displayed to the user in a useful manner while also limiting the user's ability to corrupt the integrity of the CSV data.

For instance, examine the sample markup in the image on the left, which is rendered as the UI widgets in the image on the right. Anything surrounded by pairs of '<' and '>' brackets the renderer will attempt to interpret as a markup command, where vertical pipes '|' delineate the arguments of the markup.

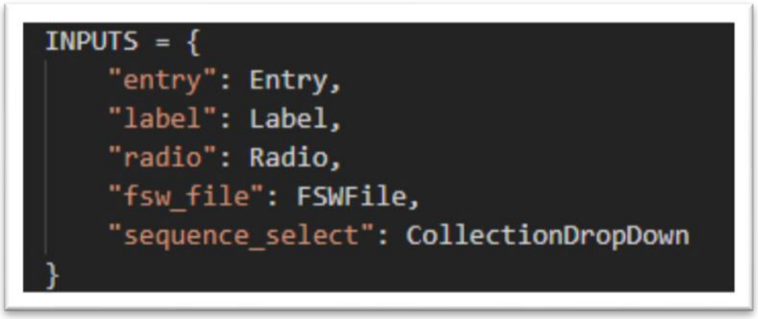
The first argument specifies which kind of render widget to display, whereas the rest of the arguments are passed into the instantiated render widget. In the sample markup from above, a label widget is being instantiated with the text value of "Category", and a radio widget is being instantiated with potential options of 'A' and 'B'.

These formatted UI widgets are useful, as labels make perfect widgets for CSV data you don't want the user changing, and radio buttons allow for structuring the choices a user can make for a data field.

If improper markup is supplied, or if markup is not detected, then a default text entry widget is instantiated, with the value of the CSV data item being used to populate the default value of the entry widget. You can see this in the sample markup above, and the result on its right with the number "100".

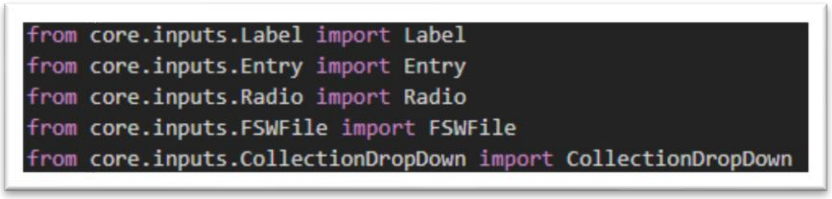
To determine which input widget to instantiate based on a raw markup string, the CSV renderer passes the string to an Input Factory. This Input Factory is responsible for parsing the markup, and determining which input widget to create.

An exhaustive list of the supported markup widgets is seen to the right, where the array key should equal the first markup argument found in the string.



```
INPUTS = {
    "entry": Entry,
    "label": Label,
    "radio": Radio,
    "fsw_file": FSWFile,
    "sequence_select": CollectionDropDown
}
```

This registration dictionary can be found in the file 'core/inputs/InputFactory.py'. If you want to add your own markup widgets, simply add your own custom key/value pair to this dictionary, where the value is the name of your widget class. Make sure to import your class at the top of the definition file before you do however.



```
from core.inputs.Label import Label
from core.inputs.Entry import Entry
from core.inputs.Radio import Radio
from core.inputs.FSWFile import FSWFile
from core.inputs.CollectionDropDown import CollectionDropDown
```

Data Controller

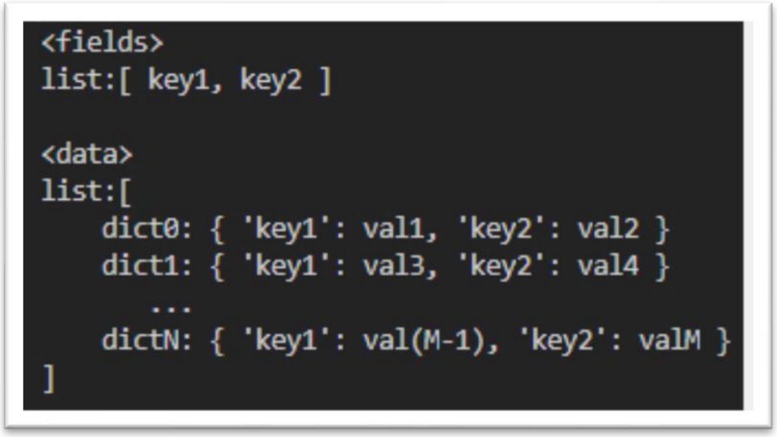
The Data Controller is quite likely inappropriately named – a better name would be the Data Driver. However, it has stuck thus far in the development cycle, and so the name will stay.

The Data Controller is a global helper class that provides a standardized mechanism for reading/writing CSV files. While all data must pass through a Model so that it can enforce the business logic of the storage mechanism, all models make use of the Data Controller's static helper methods that directly write to the CSV files.

When loading/saving with the Data Controller, it is passed a path to the file it is operating on, and a class of type CSVObject. The CSVObject class holds the data which is to be written by the Data Controller, and serves as a standardized coupling for classes that wish to save CSV data.

The CSVObject maintains two list data structures – the fields and the data. The data is a list of dictionaries, and the fields is a list of the keys used by the dictionaries. The keys in the dictionaries must match the entries in the field list during normal Save( ) operations, however this requirement can be bypassed by invoking the Data Controller's SaveSloppy( ) method.

A pseudocode example of the field and data structures, and how they are to be defined.



```
<fields>
list:[ key1, key2 ]

<data>
list:[
    dict0: { 'key1': val1, 'key2': val2 }
    dict1: { 'key1': val3, 'key2': val4 }
      ...
    dictN: { 'key1': val(M-1), 'key2': valM }
]
```

Data Structures

There have been numerous references to data structures and their definitions throughout this document, but not all of these structures have clear or explicit definitions within the source code of the application. While this might be confusing at first, this is intentional, and helps to clearly illustrate the differences between the Hard and Soft data structures used in this application.

Hard Data Structures

Hard Data Structures are explicitly defined in the source code. They have customized classes or other programmatic structures built just for them, and have their type definitions rigorously enforced by the application.

The Builder, Model, Parameters Model, and Interface classes are examples of Hard Data Structures used in this application.

Soft Data Structures

Soft Data Structures usually lack explicit definitions in the source code, and instead are a result of emergent or directed behavior exhibited by Hard Data Structures. They can exist in concept only, or they can be specified by CSV data read by the application which influences the behavior of the TSM.

A Sequence Collection is an example of a CSV data specified Soft Data Structure, while a Bench and a Unit are examples of conceptual Soft Data Structures that are implemented via a Parameters Model.

CSV File Hierarchy

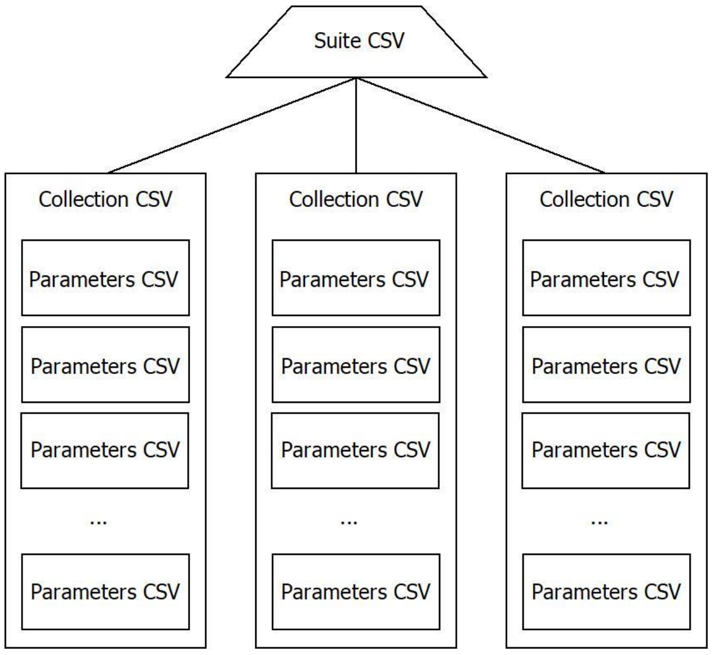
While the usage of linked CSV files allows the application to store multi-dimensional data between its runs, it does necessarily add a modicum of complexity to the storage paradigm.

It might be helpful then, to view the data structure in terms of the actual CSV files rather than models.

Each Test Suite has an associated Suite CSV file, which has three rows, with each row containing a file path to a Collection CSV.

Each Collection CSV contains an arbitrary number of rows, with each row containing a file path to a Parameters CSV files.

The Parameters CSV files possess the target data of the application, as they contain the actual data points and numerical testing parameters which are passed to the test module.



Module Development Guide

The heart of this application is its module registration and activation system. Much of the Test Suite Manager exists solely as a universal interface for the test modules, so as to allow engineers to focus on developing their SCPI and other associated testing scripts.

This section will provide a step-by-step guide for this module development process, and give explicit instructions on building, registering, and activating a very rudimentary module.

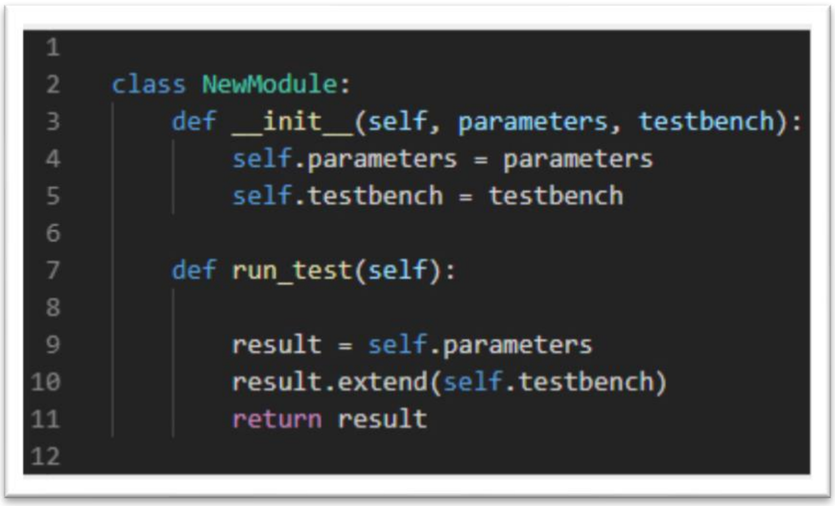
It is recommended that you use an IDE during this process.

Create Module

To begin, create a file named “NewModule.py” at the following location:

- {Install Dir}/modules/NewModule.py

And then create a class definition for this module like so:



```

class NewModule:
    def __init__(self, parameters, testbench):
        self.parameters = parameters
        self.testbench = testbench

    def run_test(self):

        result = self.parameters
        result.extend(self.testbench)
        return result

```

This definition contains some configurations that are worth explaining. All modules used in the Test Suite Manager must configure their __init__( ) constructor to accept an argument named parameters and an argument named testbench. When the module is instantiated by the application, it will pass the compiled parameter and testbench list data structures needed for the module through these arguments, and their names must match exactly to ensure this process is successful.

In order to fulfill the contract the module has with the application, it must also define a run_test( ) method which returns the result of this module’s test. For the purposes of this demonstration, this NewModule class simply concatenates its parameters list structure with its testbench list structure, and then returns the result back to the module.

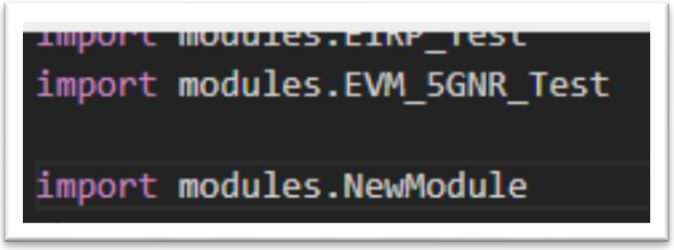
This is an excellent way to begin the module development process, as it ensures a proper connection to the application.

Register Module

Before the newly created module can be used in a test unit, it must be registered with the application. This registration process allows the Test Suite Manager to keep track of which modules are active and ready for testing, and provides a centralized path resolution system by abstracting the import statements for classes into a single global _CONFIG_ file.

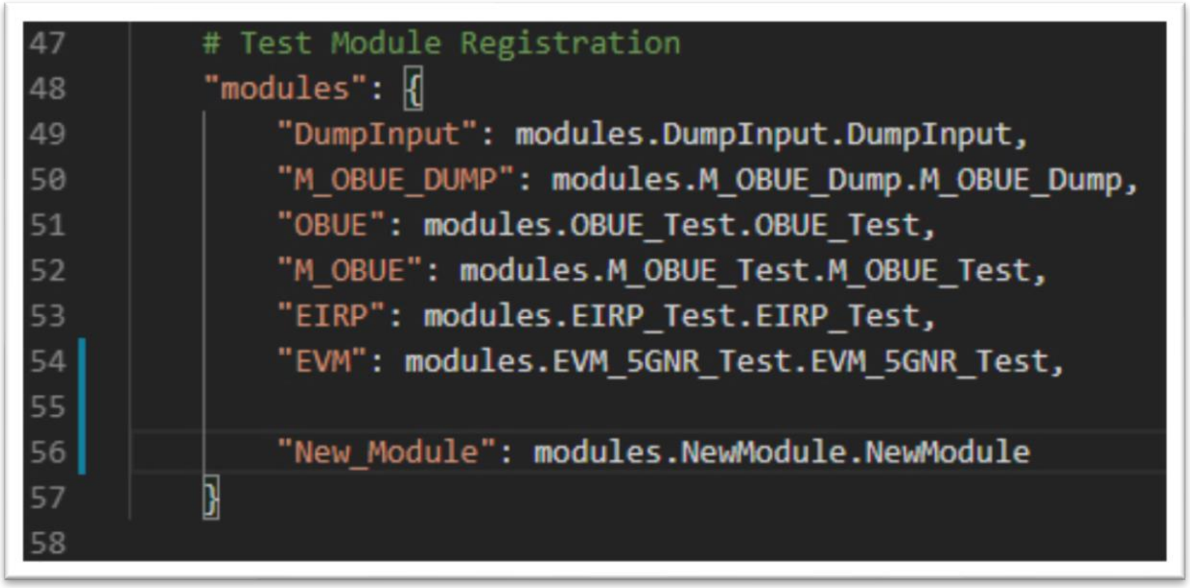
The _CONFIG_ file is a global dictionary that the entirety of the application can import and access. It contains a wide array of configuration options for the Test Suite Manager, of which the “modules” sub-dictionary is our chief target.

Open the file ‘Config.py’ located in the root of the application folder, and then add an import statement for the new module alongside the current module imports. This import statement corresponds to the path that the NewModule.py file is located at (/modules/NewModule.py).



```
import modules.EIRP_Test
import modules.EVM_5GNR_Test

import modules.NewModule
```

With the module now imported into the _CONFIG_ file, scroll down to where the “modules” sub-dictionary is defined, and add the module that was just imported.



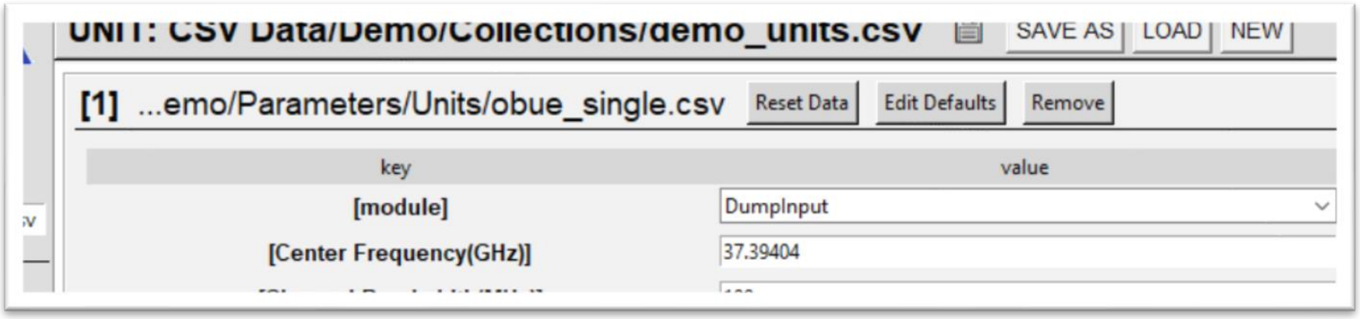
```
    # Test Module Registration
    "modules": {
        "DumpInput": modules.DumpInput.DumpInput,
        "M_OBUE_DUMP": modules.M_OBUE_Dump.M_OBUE_Dump,
        "OBUE": modules.OBUE_Test.OBUE_Test,
        "M_OBUE": modules.M_OBUE_Test.M_OBUE_Test,
        "EIRP": modules.EIRP_Test.EIRP_Test,
        "EVM": modules.EVM_5GNR_Test.EVM_5GNR_Test,

        "New_Module": modules.NewModule.NewModule
    }

```

Choose a unique key for the sub-dictionary, and then save the _CONFIG_ file.

Assign Module

Now that the module has been registered, it's time to assign it to a test unit. Run the 'run.bat' batch file to compile and open the application. If there are any issues with the module definition or how it was imported, then the Test Suite should throw an error and exit the startup process. If this occurs, run the application using step 1b) from the Quick Start guide in this document to gain access to command line debugging messages.

Otherwise, load the suite file 'demo_suite.csv', and navigate to the 'Unit Builder'.

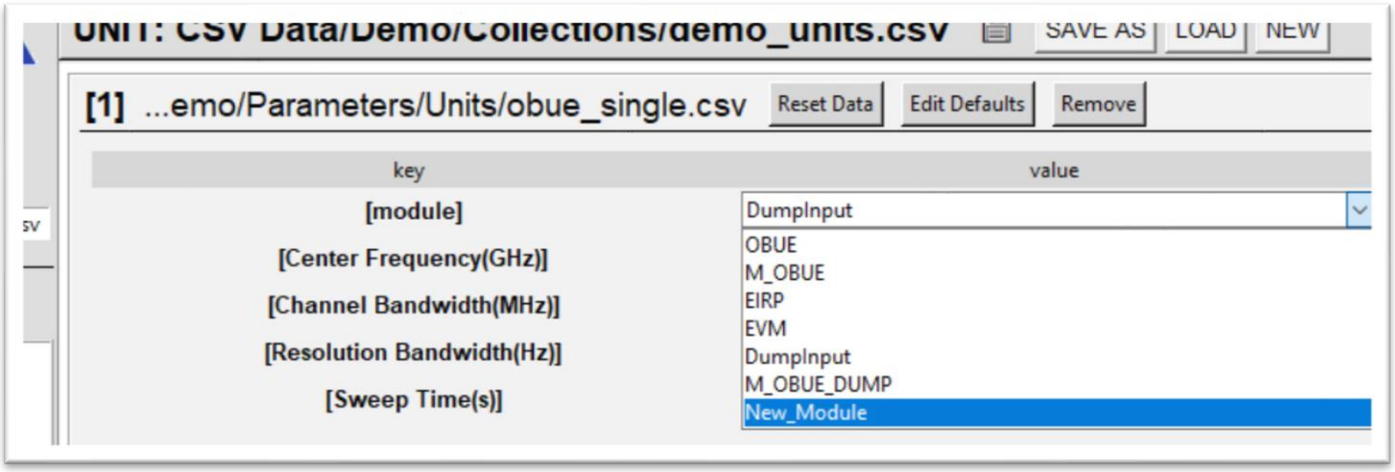


[1] ...emo/Parameters/Units/obue_single.csv

key	value
[module]	DumpInput
[Center Frequency(GHz)]	37.39404

We can repurpose the first configured test unit. In the corresponding "module" field, replace the word "DumpInput" with the dictionary key we used to register our module in the _CONFIG_ file. There is no need to save the data – the Test Suite Manager operates off of whatever is presently loaded in the application. You should only be overwriting CSV files if you are generating a new test suite, or if you want to change the default values of the test suite.

The dictionary key the module was registered with is known as the "Module Key", and it is imperative that the module key entered in the _CONFIG_ file match the module key entered in the "module" field of a test unit, otherwise the application cannot determine the correct module to instantiate.

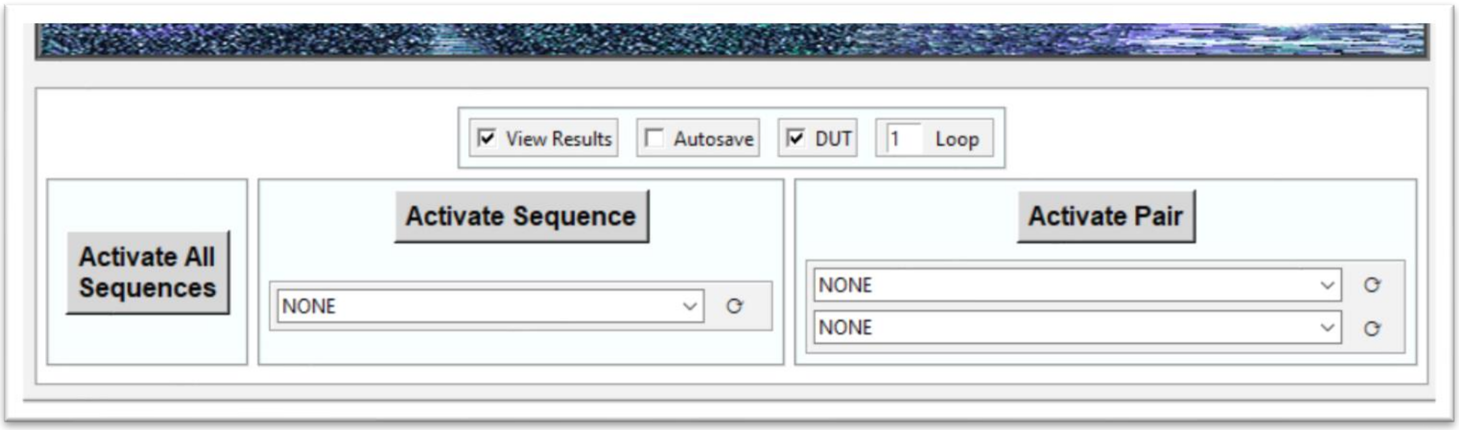


[1] ...emo/Parameters/Units/obue_single.csv

key	value
[module]	DumpInput
[Center Frequency(GHz)]	
[Channel Bandwidth(MHz)]	
[Resolution Bandwidth(Hz)]	
[Sweep Time(s)]	

Activate Module

The module is created, registered, and assigned. All the remains is to activate it. This can be accomplished by navigating to the 'Begin Testing' page.

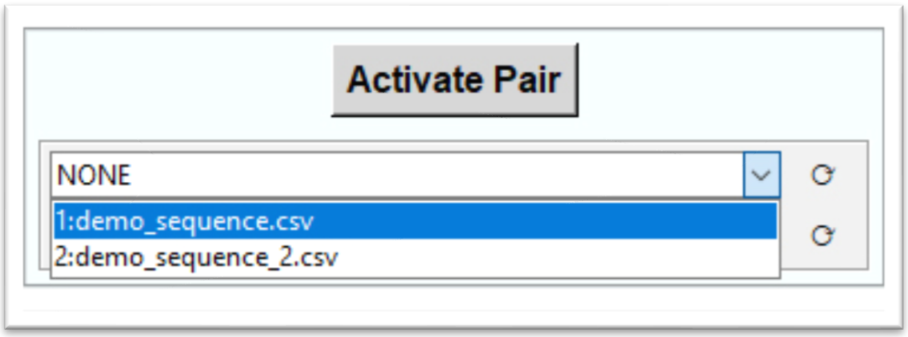


This page is divided into two sections, each with a specific purpose:

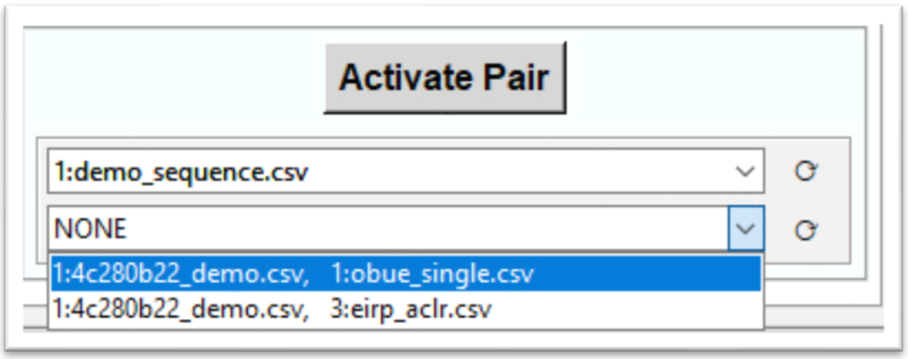
- **Configuration Panel:** Applies various configurations to the test sequence.
  - ***View Results:*** Instantiates a results window for every sequence pair that successfully completes its test, so that the results of the test can be reviewed by the user in real time. Active by default.
  - ***Autosave:*** Automatically saves the results of each test into a new CSV file. Autosaved files do not overwrite each other, potentially allowing for the gathering of mass runtime statistics. Deactivated by default.
  - ***DUT:*** The "Device Under Testing" implementation, which requires the user interact with a pop-up window before every sequence pair can proceed. Active by default.
  - ***Loop:*** The number of times to loop the entire sequence. Set to '1' by default.
- **Activation Panel:** Activates the sequencer to begin performing tests on loaded data, according to what kind of test sequence the user would like to perform.
  - ***Activate All Sequences:*** Activate every single sequence and sequence pair currently loaded in the application's "Sequence Builder" workspace.
  - ***Activate Sequence:*** Activates a specific sequence from the list of sequences currently loaded into the "Sequence Builder" workspace. Use the refresh button to reload potential sequence choices, and use the drop down list to select the specific sequence to activate.
  - ***Activate Pair:*** Activates a specific bench-unit sequence pair that is loaded into the application. The top drop down menu chooses the sequence a pair will be taken from, and the bottom drop down menu chooses the pair from that sequence. Use the refresh buttons to refresh the selection list with newly loaded sequences and pairings.

For the purpose of this exercise, we will use "Activate Pair", which will allow us to avoid activating all the sequences in lieu of the one sequence we're interested in.

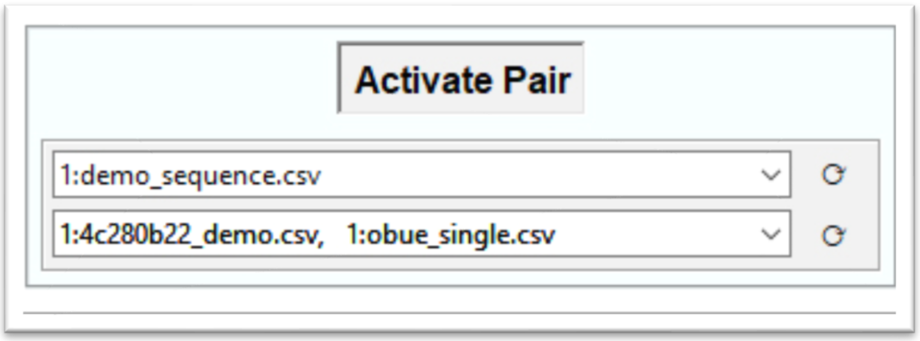
Click the refresh button next to the top drop down list, and then select the first sequence from the drop down list.



Click the refresh button next to the bottom drop down list, and then select the sequence pair which matches the test unit we just configured (should also be the first one).

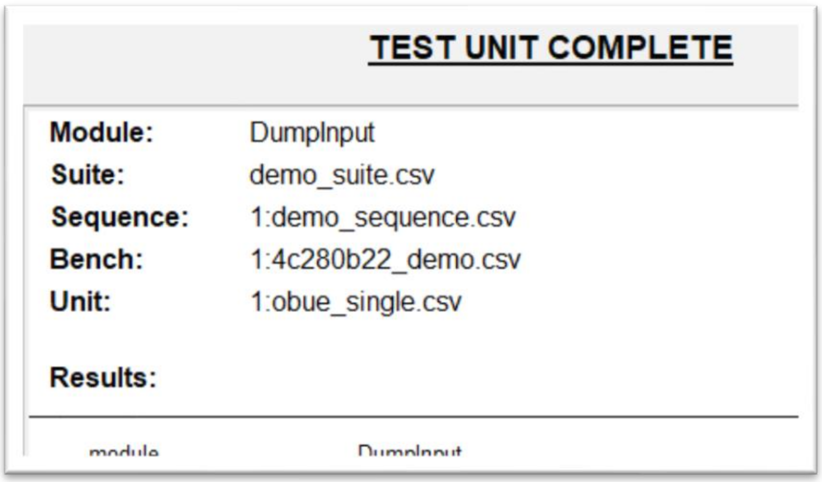

Activate Pair
1:demo_sequence.csv
NONE
1:4c280b22_demo.csv, 1:obue_single.csv
1:4c280b22_demo.csv, 3:eirp_aclr.csv

Now finally, click the “Activate Pair” button, which should activate the sequence and the new test module we just made!



Note: If you receive a “module not found error”, but did not receive an import error when starting up the Test Suite Manager, check the spelling of the module key, or try restarting the application.

If you left the “DUT” option active, simply click “Continue” to proceed with the test. After a few seconds, the dummy test should be complete, and the results window should be displayed.



TEST UNIT COMPLETE

Module:	DumpInput
Suite:	demo_suite.csv
Sequence:	1:demo_sequence.csv
Bench:	1:4c280b22_demo.csv
Unit:	1:obue_single.csv

Results:

You can use this window to ensure that the proper module was run by checking that our new module key is being used in the “Module” field of the Results Window.

And there you have it! The entire process for module development, from registration to activation. With this knowledge, you can now begin implementing your own host of test suites at your leisure.

Notes On Module Chaining & Import Resolution

Due to how the module registration and activation system is structured, it is more than possible to chain multiple modules together. One need only create a parent module class, register and assign it, and then have the parent class instantiate the chained modules before activating them as need be. Indeed, theoretically entire application windows could be instantiated from within a module.

While such endeavors could result in powerful test units, it is worth remarking however that great care must be taken when importing and chaining anything other than modules. The namespace resolution used by Python can be a bit spotty at times, and the language absolutely falls apart when faced with a recursive/circular dependency.

As such, it is recommended that developers interested in this possibility define intermediary classes that can help resolve these potential import issues, and to take the utmost caution when saving over CSV data.

Nonetheless, it could be quite interesting to see how this might be used. Theoretically, one could use the Test Suite Manager as a CSV configuration manager, and the module can instantiate additional sets of UI inputs for missing parameter data.

CSV Data Processing

While much of the previous sections focused on an abstract view of data storage via the Models design pattern, this section will describe the same data storage mechanisms but using the actual CSV files written/read by the application.

Some time will also be spent going over how this data is compiled by the application, and how it is sent to the test modules which drive the testing process.

CSV Structures

Models serve as a layer of abstraction between the Test Suite Manager and the raw CSV text data that the TSM uses to function. This abstraction is useful, as it allows for components of the applications to easily read/write data to storage without concerning themselves with how the storage mechanism is implemented.

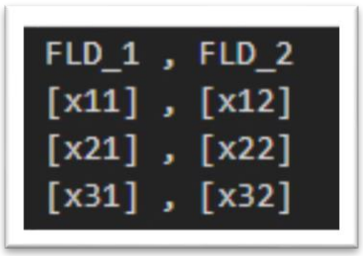
To accomplish this, the Models use a helper class called Data Controller, which contains a series of static methods that allow a Model to read/write its data from/to storage. The Data Controller makes some assumptions about how CSV data is structured, however some of these assumptions can be modified by changing the global Config file.

- Assumption 1: All data is text data
- Assumption 2: Data can be organized into rows and columns
- *Assumption 3*: A newline '\n' or '\r' marks the end of a row *(configurable)*
- Assumption 4: A comma ',' separate every column
- Assumption 5: Files that are to be loaded will have their first row specify the fields of the CSV data, and extraneous columns will be dropped.

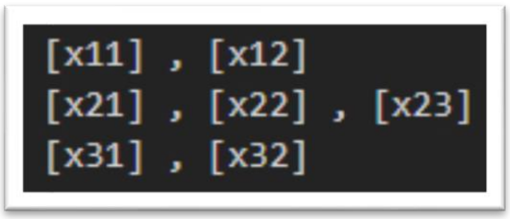
Assuming all assumptions hold true, even if a large CSV file is read without newlines or commas, it can still be interpreted by the Data Controller as one long row that is used as the only field of the column data, with zero rows of data.

Notice how there is no assumption that the number of columns of each row match each other while saving data. The Data Controller implements a Save( ) method which enforce such business logic, and it is used extensively by the Models to ensure their data integrity.

However, through the use of the SaveSloppy( ) method, the Data Controller can save arbitrary CSV data of any kind. Thus, the CSV data specified by both of the images below on the left and the right can be written by using the Save( ) and SaveSloppy( ) methods respectively.



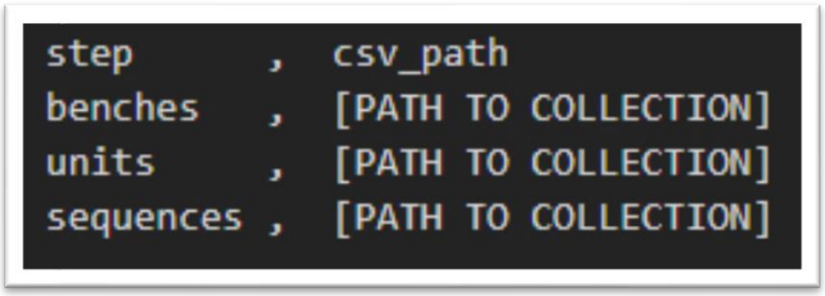
```
FLD_1 , FLD_2
[x11] , [x12]
[x21] , [x22]
[x31] , [x32]
```



```
[x11] , [x12]
[x21] , [x22] , [x23]
[x31] , [x32]
```

Suite

A Suite's CSV file, which is maintained by the Suite Model, follows this pattern:



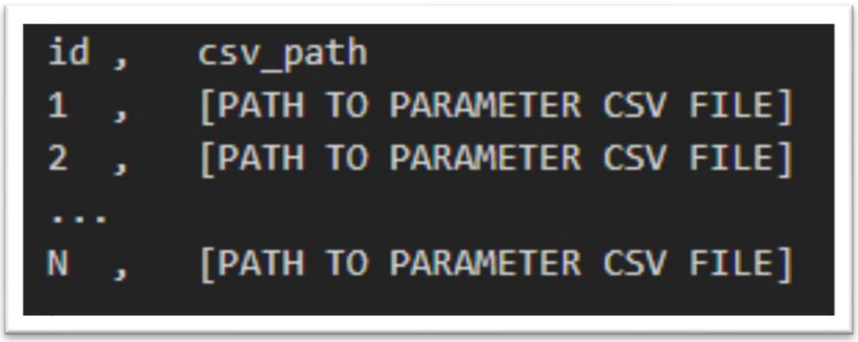
```
step      ,  csv_path
benches   ,  [PATH TO COLLECTION]
units     ,  [PATH TO COLLECTION]
sequences ,  [PATH TO COLLECTION]
```

Where the step column specifies a step of testing configuration, and the csv_path column specifies a path to a data collection which contains references to the parameters of the testing configuration.

Here, the "benches" row specifies a csv_path to the collection used by the Bench Builder.

Collection

A Collection's CSV file, which is maintained by the Collection Model, follows this pattern:

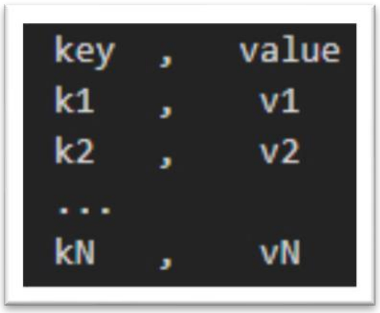


```
id ,   csv_path
1  ,   [PATH TO PARAMETER CSV FILE]
2  ,   [PATH TO PARAMETER CSV FILE]
...
N  ,   [PATH TO PARAMETER CSV FILE]
```

Where the id column serves as a numerical index for the loaded parameters CSV file, and the csv_path column refers to the path to the CSV file. The numerical indexing allows for multiple instances of the same CSV file to be loaded into a collection, while still keeping each row differentiable.

Parameters

The Parameters Model is used to store the indexed data of the Benches and Units, and utilizes the following pattern:



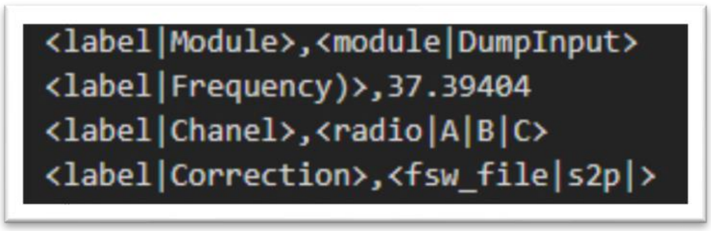
```
key  ,   value
k1   ,    v1
k2   ,    v2
...
kN   ,    vN
```

Note that extra columns can be specified in these files, and they will be read as additional arguments into the Parameters Model data structure as specified in the section Architectural Concepts > Model > Parameters Model.

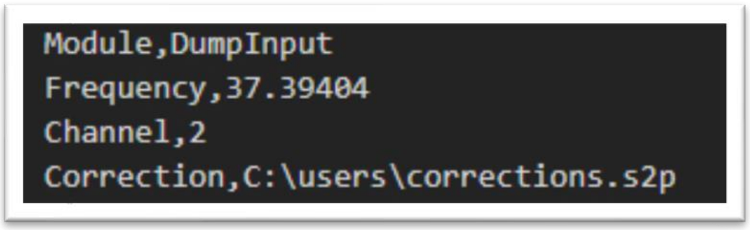
Parameters and Formatted Data

Throughout the current CSV files, there are ample instances of the application's CSV Renderer markup language. This markup is completely optional, and was only implemented to illustrate the Test Suite Manager's capabilities for Guided Suites.

Functionally, the two CSV files on the left and the right can be compiled by the Test Suite Manager into the exact same data representation, based on how the user interacts with the CSV Renderer. The markup only changes what is presented to the user by the CSV render, which can change the behavior of the user but does not alter the overall structure of the compiled data which is sent to the test unit.

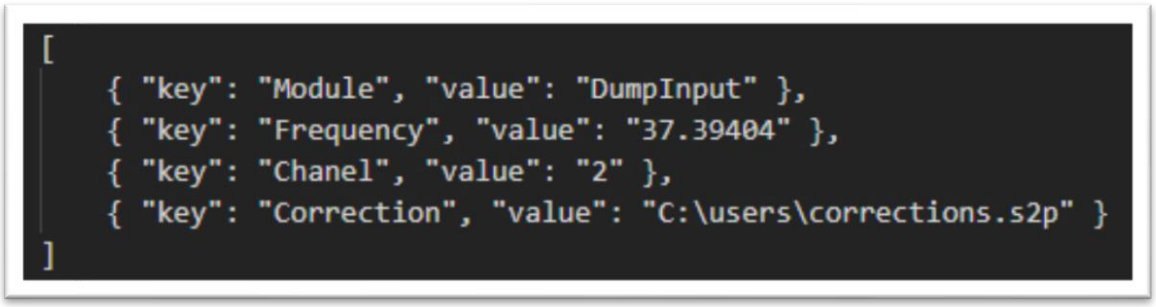


```
<label|Module>,<module|DumpInput>
<label|Frequency)>,37.39404
<label|Chanel>,<radio|A|B|C>
<label|Correction>,<fsw_file|s2p|>
```



```
Module,DumpInput
Frequency,37.39404
Channel,2
Correction,C:\users\corrections.s2p
```

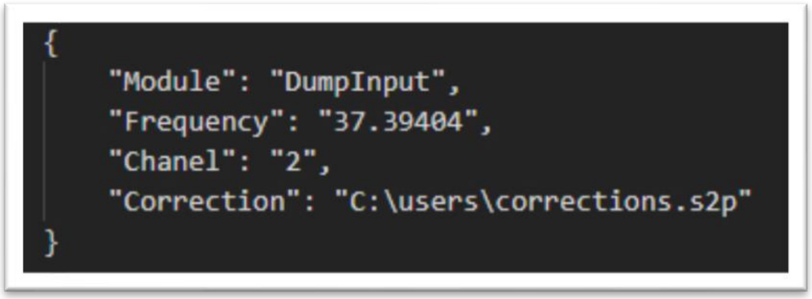
Assuming that the user interacts with the CSV Renderer that displays the markup on the left, where they select radio option 'B' and then uses the FSW File browser to get the path of the correction file 'C:\users\corrections.s2p', then the application will compile the data into a list of dictionaries that is passed to the module. This list of dictionaries will be identical to the one produced by the CSV file on the right, and will have the form:



```
[
    { "key": "Module", "value": "DumpInput" },
    { "key": "Frequency", "value": "37.39404" },
    { "key": "Chanel", "value": "2" },
    { "key": "Correction", "value": "C:\users\corrections.s2p" }
]
```

This raw, flat data is passed to the modules in the event that they find it to be a more useful representation of the data than the parameter's traditional indexing methodology. If the module however requires that the data be indexed in a dictionary, then it can utilize the Data Controller's static helper function GetDictionary ( ), which converts this list of dictionaries into an indexed dictionary.

Examples of doing this exist in the previously developed modules, and passing the flat data into the GetDictionary( ) method will result in the following dictionary structure:



```
{
    "Module": "DumpInput",
    "Frequency": "37.39404",
    "Chanel": "2",
    "Correction": "C:\users\corrections.s2p"
}
```

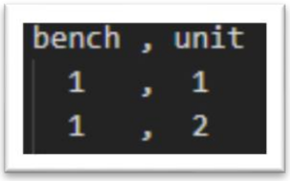
Unlike the parameters model, this dictionary indexing of flat data does not preserve extra columns as additional arguments. Rather, the GetDictionary( ) method just drops extraneous columns, keeping only the data contained in those used as the key/value fields.

Sequences

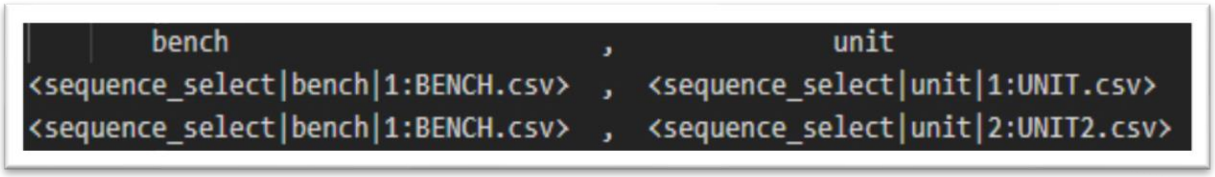
Sequences do not use the Parameters CSV paradigm, as the sequence pairings are not indexed. Instead, sequences make use of the original Model class, which operates solely on flat data.

However, to dramatically improve the user experience, by default sequences will auto populate their CSV data with markup to the sequence_selector widget.

While the markup for the sequence selector widget contains more data than a plain text entry would, it was deemed that UI experience would be prioritized in this instance. However, it is still possible to feed raw text-data into the sequence, it is just harder to read. The two CSV instances below will be compiled by the application in the same way.



```
bench , unit
  1   ,  1
  1   ,  2
```

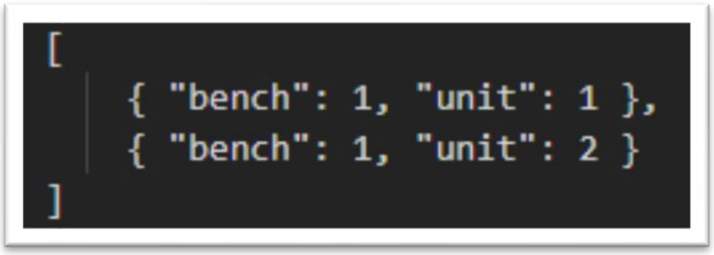


```
          bench                        ,              unit
<sequence_select|bench|1:BENCH.csv>   ,   <sequence_select|unit|1:UNIT.csv>
<sequence_select|bench|1:BENCH.csv>   ,   <sequence_select|unit|2:UNIT2.csv>
```

The number in the raw text CSV on the left is the numerical index of a Bench/Unit in the associated Collection used by the Suite. The markup on the right instantiates a sequence drop down which takes two arguments. The first argument specifies which data collection the drop down will fetch its values from, and the second specifies the default value of the drop down. Notice the corresponding numerical indexes at the beginning of the second argument strings.

When the drop down is instantiated, it uses the numerical index to gain more information about the suite, so as to make the drop down menu more readable for the user. This is why the default values have such extraneous data.

Nonetheless, when the sequence drop down returns the value of its current selection, it returns the numerical index, not the full user-friendly text. Thus when the data is compiled, it will follow this pattern:

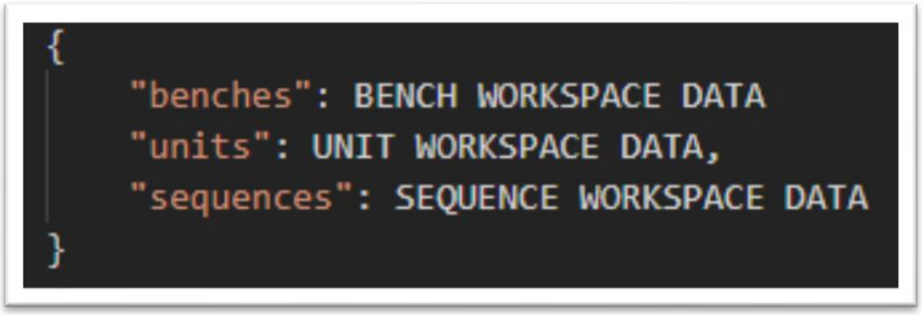


```
[
    { "bench": 1, "unit": 1 },
    { "bench": 1, "unit": 2 }
]
```

Data Compilation

Whenever the sequencer is activated, it compiles all of the data currently loaded in the application. Using the sequence data passed to it, such as a sequence index or a bench/unit pair, the sequencer extracts the relevant information from the compiled data, and passes it to the module specified by the unit.

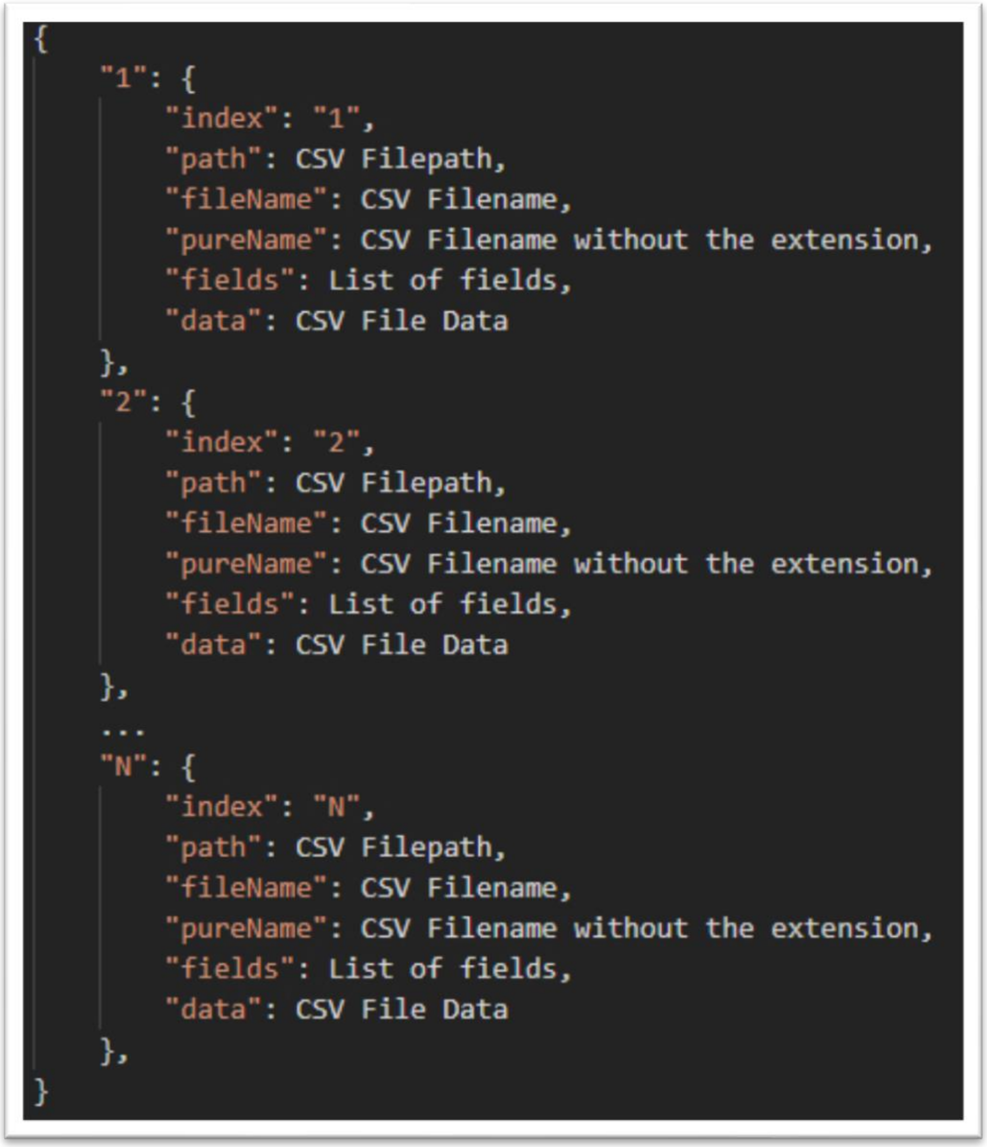
This compilation starts in the Suite Manager, which defines a dictionary to contain its child data:



```
{
    "benches": BENCH WORKSPACE DATA
    "units": UNIT WORKSPACE DATA,
    "sequences": SEQUENCE WORKSPACE DATA
}
```

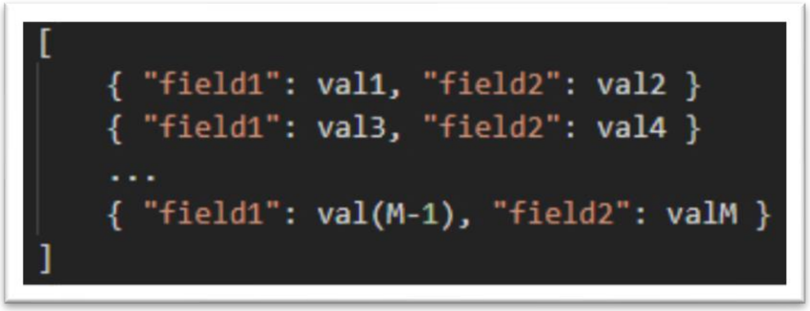
The Suite then calls on each of the workspace Builders to compile their own data. Each Builder then instantiates a dictionary, and populates it with the data that the Builder presently has loaded.

The Builder indexes its dictionary based on the numerical indexing of each CSV file in the collection, and points that index to a dictionary filled with meta data about that CSV file. After filling in the meta data, the Builder then calls upon each CSV renderer to compile its data, and places that data in the value of its corresponding dictionary entry.



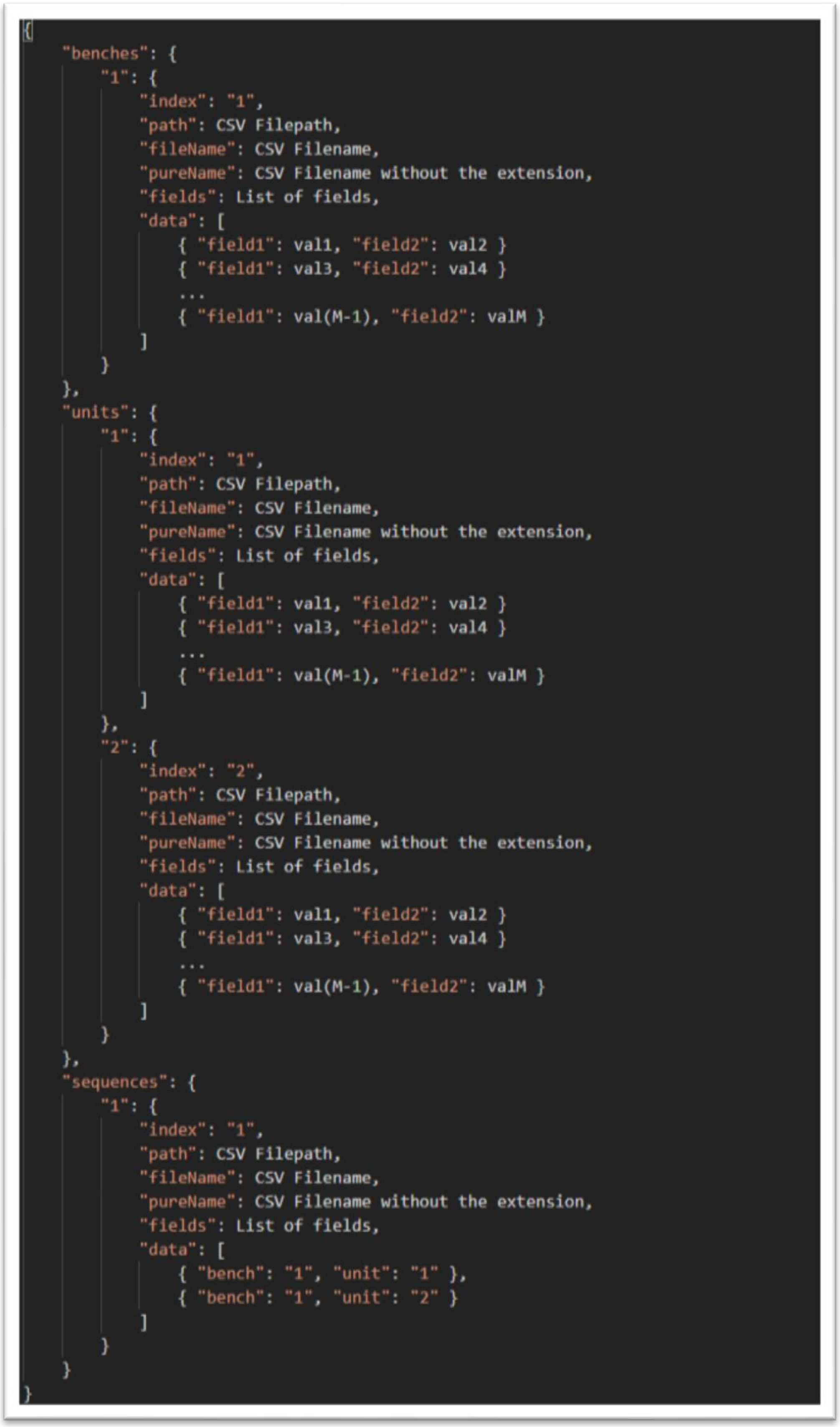
```
{
    "1": {
        "index": "1",
        "path": CSV Filepath,
        "fileName": CSV Filename,
        "pureName": CSV Filename without the extension,
        "fields": List of fields,
        "data": CSV File Data
    },
    "2": {
        "index": "2",
        "path": CSV Filepath,
        "fileName": CSV Filename,
        "pureName": CSV Filename without the extension,
        "fields": List of fields,
        "data": CSV File Data
    },
    ...
    "N": {
        "index": "N",
        "path": CSV Filepath,
        "fileName": CSV Filename,
        "pureName": CSV Filename without the extension,
        "fields": List of fields,
        "data": CSV File Data
    },
}
```

The CSV Renderer in turn compiles its data, returning a flat list of dictionaries holding the data for the CSV file. Each dictionary corresponds to a row of data in the CSV file, with the keys of the dictionaries being the fields of the CSV file.



```
[
    { "field1": val1, "field2": val2 }
    { "field1": val3, "field2": val4 }
    ...
    { "field1": val(M-1), "field2": valM }
]
```

Thus, if a suite were composed of 1 bench, 2 units, and 1 sequence, where the sequence has a pair for each Unit-Bench combination, it's compiled data might look like:

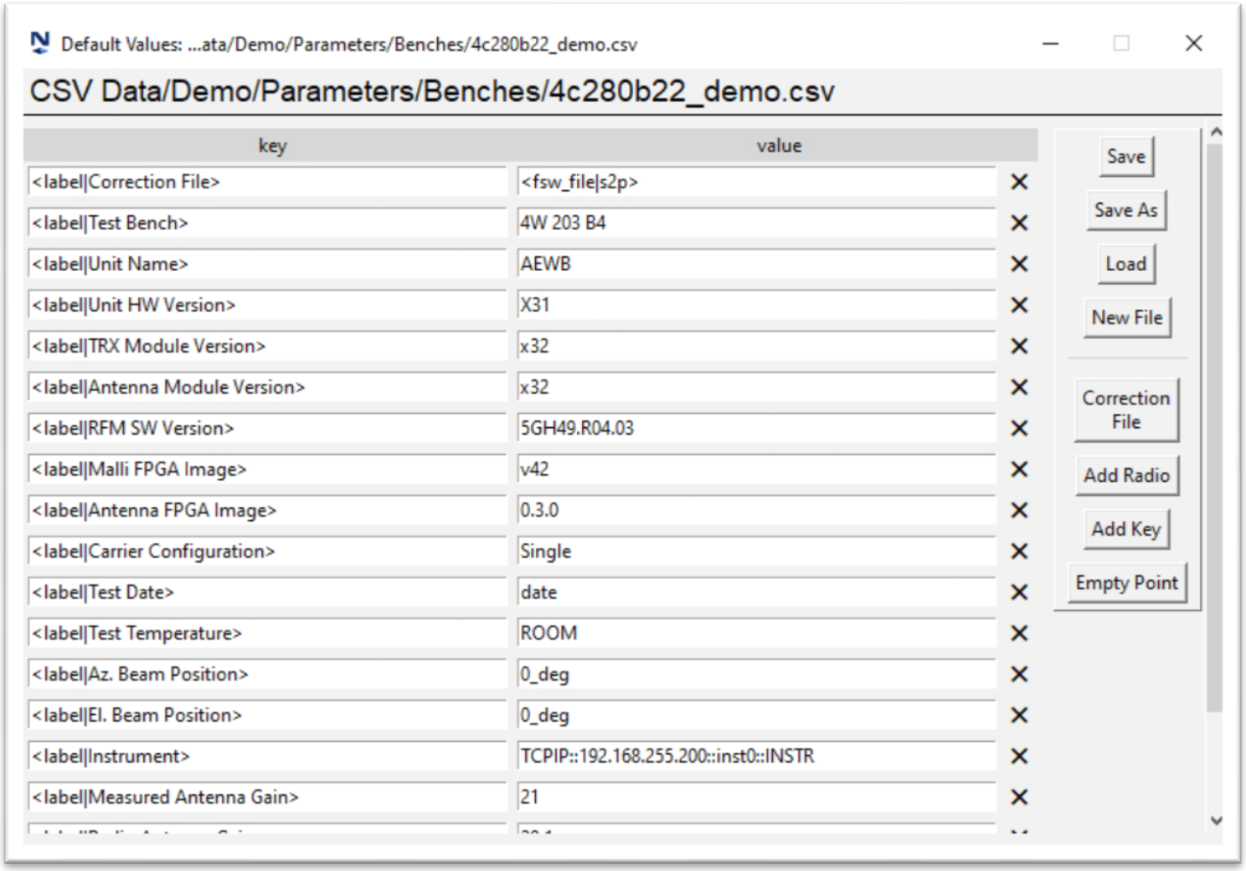


```
{
    "benches": {
        "1": {
            "index": "1",
            "path": CSV Filepath,
            "fileName": CSV Filename,
            "pureName": CSV Filename without the extension,
            "fields": List of fields,
            "data": [
                { "field1": val1, "field2": val2 }
                { "field1": val3, "field2": val4 }
                ...
                { "field1": val(M-1), "field2": valM }
            ]
        }
    },
    "units": {
        "1": {
            "index": "1",
            "path": CSV Filepath,
            "fileName": CSV Filename,
            "pureName": CSV Filename without the extension,
            "fields": List of fields,
            "data": [
                { "field1": val1, "field2": val2 }
                { "field1": val3, "field2": val4 }
                ...
                { "field1": val(M-1), "field2": valM }
            ]
        },
        "2": {
            "index": "2",
            "path": CSV Filepath,
            "fileName": CSV Filename,
            "pureName": CSV Filename without the extension,
            "fields": List of fields,
            "data": [
                { "field1": val1, "field2": val2 }
                { "field1": val3, "field2": val4 }
                ...
                { "field1": val(M-1), "field2": valM }
            ]
        }
    },
    "sequences": {
        "1": {
            "index": "1",
            "path": CSV Filepath,
            "fileName": CSV Filename,
            "pureName": CSV Filename without the extension,
            "fields": List of fields,
            "data": [
                { "bench": "1", "unit": "1" },
                { "bench": "1", "unit": "2" }
            ]
        }
    }
}
```

CSV Editor

To help in the development process, a CSV Editor was included in this application. It's primary function is to aid in building new Test Suites from scratch, as it comes with many useful buttons that auto-populate its CSV data with markup. Its secondary but still useful function is to edit the default values a test suite uses natively in the application, theoretically eliminating the need for 3rd party CSV editing software.

To open the CSV editor, open or build a suite, find one of the CSV files currently opened in the suite, and then click the "Edit Defaults" button.



CSV Data/Demo/Parameters/Benches/4c280b22_demo.csv

key	value
<label|Correction File>	<fsw_file|s2p>
<label|Test Bench>	4W 203 B4
<label|Unit Name>	AEWB
<label|Unit HW Version>	X31
<label|TRX Module Version>	x32
<label|Antenna Module Version>	x32
<label|RFM SW Version>	5GH49.R04.03
<label|Malli FPGA Image>	v42
<label|Antenna FPGA Image>	0.3.0
<label|Carrier Configuration>	Single
<label|Test Date>	date
<label|Test Temperature>	ROOM
<label|Az. Beam Position>	0_deg
<label|El. Beam Position>	0_deg
<label|Instrument>	TCPIP::192.168.255.200::inst0::INSTR
<label|Measured Antenna Gain>	21

You may notice that the available buttons in the CSV Editor can change based on which Builder it is instantiated from. This is intentional, as the application is able to configure what auto-populate buttons are available to a CSV Editor on a Builder basis. There is no need for a Bench Builder to add MOBUE channel configurations, and the CSV Editor it can instantiate reflects that.