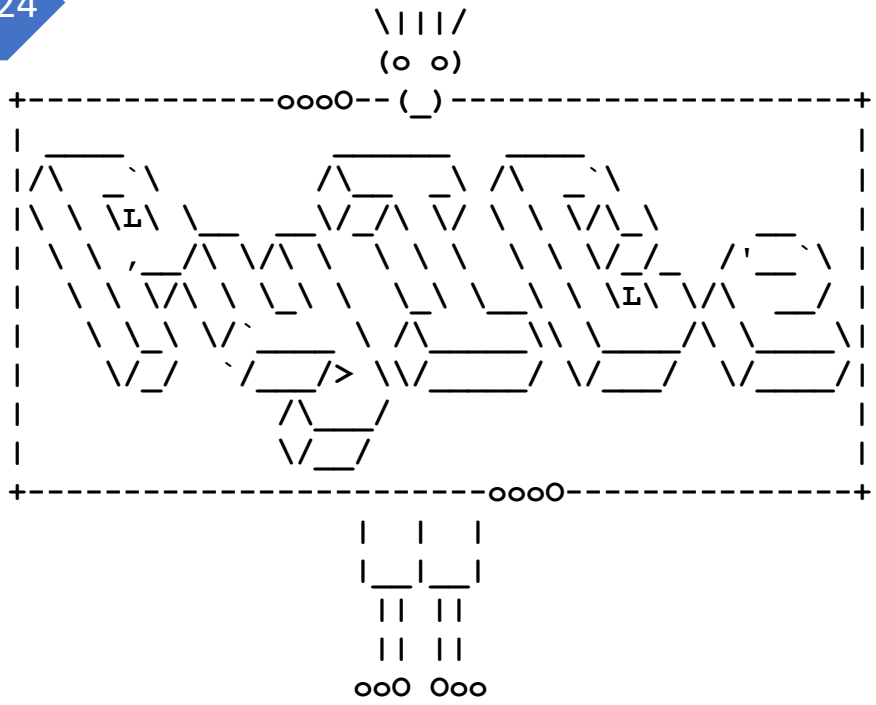
July 12, 2024



Infrastructure Extensions

Russell McFadden & Steven Martin
ANALOG DEVICES, INC.

Contents

CHAPTER 0 - INTRODUCTION

Introduction

PyICe (**Py**thon **I**ntegrated **C**ircuit **E**nvironment) is an automation tool that streamlines verification of any product, circuit board, integrated circuit, etc. through a powerful, unified, abstraction layer. It puts all bench resources, from the target IC or board to the laboratory equipment, etc. on an equal footing into a common *namespace*. In a sense, PyICe achieves in Python what HPIB/GPIB/IEE-488/SCPI attempted and failed at so many times before. Building on PyICe's groundwork, with some organization, planning and user input, aspects of an entire project can be unified across multiple team members, automated and simplified significantly. Some features of adopting this workflow are outlined below. These features will be referred to as the **P**yICe **I**nfrastructure **E**xtensions (P.I.E.).

Unified Test Parent Class

Under the proposed workflow, all test scripts, i.e. scripts that are intended to measure a specific performance metric from the device under test, will be added to an instance of a PyICe ***Plugin_Manager*** object. By adopting this approach, methods that are needed to manage tedious aspects of a large project can be offloaded onto PyICe, absolving the evaluation team from having to code or repeat those tasks over and over for each test. A prescribed set of methods will be required for each test, unifying the test procedure across and within the project and its team of contributors.

Setup Automation

PyICe uses the concept of a ***channel_master*** to aggregate the *location handles* it needs to access all of the registered bench resources, including the DUT. Once a channel master is created, a PyICe logger then provides the service of collecting the values of all known channels of the channel master each time the user denotes a time point of interest (by calling the PyICe logger's `log()` method). In the proposed workflow, both a master and a logger object will automatically be generated in a higher layer of PyICe, and the channels will be automatically populated using the instruments identified for the specific test computer. This feature absolves the project leader

of having to create their own PyICe master and logger and provides high level services such as creating a SQLite database file, running test suites over temperature and archiving test results, etc.

Temperature On the Outer Loop

When evaluating device performance over a range of temperatures, it is quite burdensome to wait out the slew and soak times of the temperature forcing instrument (environmental chamber or ThermoStream™ for example). When running a collection of tests, or *test suite*, over temperature, this burden is even more untenable. PyICe now provides a mechanism to run a suite of tests at a given temperature before moving on to the next temperature, as opposed to running all temperatures for one test then all temperatures again for the next test and so on. This saves a tremendous amount of time waiting for the temperature to settle as well as saves on temperature controlling resources such as liquid nitrogen, electricity, time, and untoward noise.

Bench Instrument Abstraction Layer – Unified Namespace

Assembling a coordinated evaluation team that can share code, edit, and run each other's tests and review each other's test results and methods provides untold benefits to quality and productivity. To achieve this, all members of the team must share a common code base. This generally means that a centralized software version management system should be deployed such as SVN, GIT, Perforce, etc. If the team is not familiar with good version management and data/code collaboration tools, it behooves the team leader to arrange a few training sessions.

Notwithstanding a common code base, having various members collaborate and share code at physically different sites, with physically distinct instruments, requires some planning and abstraction. The key to successful disambiguation of the physical instruments is to store the instrument address information (per user, bench, or site) in a common location and with a structured, common format.

Exploiting the bench abstraction workflow proposed herein separates the numerous test scripts from the instruments to be controlled, meaning that a test can be run on any one of a myriad of benches/sites with entirely different pieces of equipment and without a single changed line of test code. The lightweight, user specific, equipment abstraction code is reduced to a single small file per user in a common folder within the project.

Data Management

The proposed project infrastructure methodology comes with a built-in data management plan. For each test script written, generally targeting the measurement of a single parameter of interest or a collection of related parameters, a single ***SQLite*** file is generated. This file will be deposited in a scratch area of the test's folder and be amended with a time-stamped table of the logged data upon each run of the target script. All data operations, from curve fitting to plotting to interpolation to extrapolation to limit testing, etc. will originate from that file using easy to learn SQLite queries or Python transformations. The days of Excel row and column manipulation, and confusing schema-less .csv files, are over.

Traceability Automation (optional)

In addition to the data logger, a second, meta-data, table in the SQLite database can be made to automatically log various meta-data information about the test at the onset of data collection. The project can now store what connections have been made for the test, what specific DUT was used, the serial numbers of the instruments, who the operator was, and whatever else the project manager deems relevant or valuable.

Special Channel Actions (optional)

During certain circumstances of testing, it might be desirable to alert a human or take another special action when a particular event has occurred. Examples might be a voltage going out of range, a device stops responding correctly or an environmental chamber won't regulate. Using the *special channel action* feature, trigger conditions can be assigned to a channel and an action function can be triggered at each logging of the database. For example, it can ensure that the temperature sense value of a temperature control chamber is within an acceptable error band and shut down the test if it is not. Alternately, it can help with debugging DUT problems by immediately sending out an email or SMS notification with the present test conditions if a parameter ever exceeds a particular threshold.

Cleanup Automation

At the onset and end of each test (script), it is desirable to put the test equipment and DUT into a known state so that the next test will run correctly, or the bench can be safely abandoned. By inheriting the test master class and local instrument *mini-driver*, or driver wrapper, methodology, there is no need to include shutdown actions at the end of every test. Rather, local instrument driver wrappers control what the project deems appropriate for shutdown actions. These actions are then executed automatically upon the conclusion of each test. This feature returns the entire bench to a safe and neutral state, ready for the next round of tests. Declaring the shutdown state of each resource will occur one time by the project leader as part of the project specific infrastructure. Of course, this does not preclude a custom cleanup routine from executing within any given script if desired.

Notifications (optional)

The notifications feature of PyICe can be used to send emails and/or texts to interested parties when a test is started, completed, a temperature is completed, or a test crashes. Likewise, team members can get test results delivered immediately upon a test successfully completing. Those results could include details such as a pass/fail declaration of the data collected and/or the actual plots generated by the test script.

Post Collection Data Archival (optional)

Upon the successful completion of a test script, a SQLite file is deposited in a scratch area within the test script's folder. This file contains *all* runs conducted on that test for *this* computer in time-stamped tables. By adopting the data archive extension, a copy of the most recently collected data (table) will be deposited in an automatically generated archive within its own folder, titled after the time and data of the run. The data can be compared to a test's established limits provided elsewhere and can generate fresh plots as needed. Those results can then be added to the version management system demonstrating a running regression of the given parameter. Once the data is archived, the outputs it creates, pass/fail results, correlation results, plots, etc. can all be *regenerated* post-facto. These post processing actions are intentionally separated from the data collection activity. They can be run individually on archived data to quickly refine plots or reestablish a pass/fail status after changes are made to the limits.

Bench Connection Compatibility Check (optional)

Physical test benches consist of many instruments and boards linked together in a variety of ways. To adopt as much automation as possible, it is desirable minimize the number of instrument connection *configurations*. Consequently, suites of tests, while powerful, can only exist if the equipment connections are consistent for all tests within a test suite. Therefore, it behooves the evaluation project leader to adopt as many inter-connection consistencies as possible. There are physical tools available to help this automation consistency effort. These may

include a permanently connected high-channel-count scanning voltmeter, relay or multiplexing resources such as individual ammeters, RF Muxes, Multi-channel dataloggers, etc. Notwithstanding a robust effort at physical automation, planning and consistency, some human interaction resulting in connection variation is likely to remain.

Variations in interconnectivity means that some groups of tests just can't be run as a suite due to a conflict of connections that prevent proper operation of the device (board, IC, product) under test. PyICe, along with some user input on which instruments are present and how they are connected, can be used to prevent incompatible tests from being run together. If adopted, it will alert the user of the discrepancy, saving time spent gathering error prone data. Fringe benefits of adopting this feature include automatic electronic logging of how the bench was configured and even an auto-generated diagram of the bench equipment and connections (see Bench Visualization). This data can be very powerful for conducting audits and reviews well after the test is completed and the physical bench connections disassembled.

Bench Visualization (optional)

If the Bench Connection Compatibility Check feature is adopted, a list of bench connections is created that can be parsed and logged. For humans that better grasp concepts through visual aid, a tool is provided to generate an image of the lab bench based on the instruments used and the connections made between them. This feature is optimum for including in reports and for easy test reproduction in a collaborative environment. It can also prove invaluable for audits and reviews.

CHAPTER 1 – PROJECT STRUCTURE (FOLDERS)

Introduction

It is strongly recommended to have the following folder hierarchy in place. The PyICe infrastructure extensions will look specifically for some of these folders by name. You will also need to add the folder *above* the project's main folder to the PYTHONPATH environment variable.

For example: PYTHONPATH = **C:\users\<*yourname*>\projects**

```
project_name
    ├── eval_plan (highly recommended)
    │       └ eval_plan.xls
    │
    ├── infrastructure
    │       │
    │       ├ benches
    │       │       ├ bench_file1.py
    │       │       ├ bench_file2.py
    │       │       ├ ...
    │       │       └ bench_filen.py
    │       │
    │       ├ hardware_drivers
    │       │       ├ power_supply.py
    │       │       ├ oscilloscope.py
    │       │       ├ electronic_load1.py
    │       │       ├ electronic_load2.py
    │       │       └ ...
    │       │
    │       └ plugin_depdendencies
    │               ├ metadata_gathering_fns.py
    │               ├ project_settings.py
    │               ├ test_template.py
    │               └ ...
    │
    └── tests
            ├ test1
            │       ├ test.py
            │       ├ run.py
            │       ├ debug.py
            │       └ plot.py
            │
            └ test2
                    ├ test.py
                    ├ run.py
                    ├ debug.py
                    └ plot.py
```

eval_plan

PyICe, itself, will not look for the ***eval_plan*** folder but it is highly recommended that the project's evaluation plan be documented. The evaluation plan could be something as simple as an Excel sheet or a .csv file but simply *having* an evaluation plan can keep the evaluation project focused and on track. If the evaluation plan is more structured, such that it has identifiable specification ***keys*** for example, it should be possible to reference the plan from with the Python test scripts to apply limit-testing against documented limits, etc. This can be done by importing the plan into the project using Pandas or any

other Python importation method. The evaluation module will provide the limit testing services against which measured data will be compared and subsequently generate a compliance report.

infrastructure

The ***infrastructure*** folder contains project-specific code needed to connect *this project* to PyICe.

The folders: ***benches***, ***hardware_drivers*** and ***plugin_dependencies*** must be present under ***infrastructure***. Other project specific folders and code may be added to the ***infrastructure*** folder as needed.

benches

The ***benches*** folder contains all of the unique bench files needed for each physical lab-bench involved in the project. These files, an abstraction layer of device addresses, represent the entirety of the *user-specific* unique code that should be needed on a multi-teammate project. All of the other project code should be 100% shareable by all team members. Generally, there will be one file per bench (i.e. computer) as the instrument addresses tend to be computer specific.

hardware_drivers

PyICe has a substantial set of detailed instrument drivers in its ***lab_instruments*** folder. These drivers map the instrument specific instructions (usually SCPI or even… 🤮 binary) to PyICe *channels*, thereby creating a consistent, user-friendly, abstraction layer.

On any given project, it will be necessary to include *yet another* abstraction layer defining the PyICe channel names for this project as well as selecting the appropriate instrument drivers for each piece of bench hardware. The totality of the project-assigned channel names from all hardware drivers forms the project's channel *namespace*.

plugin_depdendencies

The folder ***plugin_dependencies*** must contain at least one file that declares a dependency class to be inherited by *each* test script. That template class must, in turn, inherit global methods from a centralized PyICe class. Together, this chain of dependencies unlocks the power of the PyICe Infrastructure Extensions workflow. Additionally, there must be a file that contains a dictionary of settings for the project which will be passed on in helper scripts.

tests

This is where the project's tests should reside. A test, in this sense, is a script (file) intended to capture a specific behavior, characteristic or parameter of the device under test, possibly over a wide range of adverse operating conditions such as temperature, voltage, current, frequency, etc. A test can result in one or more plots, an evaluation pass/fail report, bench connection diagram, etc.

The grouping of tests is left to the project manager, but it is strongly recommended that there be a single folder per test. Each test folder will contain the main script for that test along with a small set of helper scripts to kick off that test. The list of files in each specific test folder will be something like:

```
── tests
    ├ test1
    │    ├ test.py
    │    ├ run.py
    │    ├ debug.py
    │    └ plot.py
    │
    └ test2
         ├ test.py
         ├ run.py
         ├ debug.py
         └ plot.py
```

CHAPTER 2 – BENCHES

Each file in ***benches*** must be named identically to the computer's name to which the test equipment is connected. The computer name can be found using the **System Information** query in Windows, for example (note that dashes must be replaced with underscores).

Each file contains a single function: **`get_interfaces()`**. This function returns a dictionary with project defined *keys* corresponding to the instruments used on the bench. The corresponding values are the PyICe interfaces used by the instruments to communicate with the computer for *that* particular bench. For more information on the PyICe interface_factory, see the tutorials within the PyICe repository,

```
from PyICe.lab_interfaces import interface_factory

def get_interfaces():
    my_if = interface_factory()
    return {'CONFIGURATOR': my_if.get_visa_serial_interface('COM8', baudrate=57600),
            'BKCHMIDDLE'  : my_if.get_visa_interface('USB0::0x2EC7::0x8800::802197042757410034::0::INSTR')
           }
```

CHAPTER 3 – HARDWARE DRIVERS (MINI-DRIVERS)

The files in the ***hardware_drivers*** folder (AKA *mini-drivers*) will find the bench file for *this computer* (matching file name to computer name) and look for the interface key associated with a given instrument. If found, it will add the instrument's channels to the PyICe channel-master.

The files in the ***hardware_drivers*** folder will be automatically scanned and in each file a specific function, **`populate()`**: will be executed.

Below is an example of a hardware mini-driver:

```
from PyICe.lab_instruments.bk8600 import bk8600

def populate(self):
    if 'BKCHMIDDLE' not in self.interfaces:
        return {'instruments':None}
    load = bk8600(self.interfaces['BKCHMIDDLE'], remote_sense=False)
    channel = load.add_channel('iload', add_extended_channels=True)
    return {'instruments':[load], 'cleanup_list':[lambda : channel.write(0)]}
```

Notice that the populate method looks for the key by name in **self.interfaces** that was declared in the bench files as described earlier.

Also notice that, from and within the plurality of many instrument mini-driver files, the project specific PyICe channel *namespace* (e.g. 'iload', 'tdegc', etc.) is collectively created.

The mini-driver must return a dictionary with a key of ***instruments*** consisting of a list of channels to be aggregated.

It is also possible to return dictionary keys of ***cleanup_list***, ***special_channel_actions, startup_list, temperature_run_startup_list***, ***shutdown_list***, and ***temp_control_channel***. These keys instruct P.I.E. how to behave with regard to special instrument channels and will be explained later.

Of these keys, only the ***instruments*** key is absolutely required. Its value can be Python ***None*** but the key must be present. If the instrument's key is None, then none of the optional keys will be acknowledged.

The associated values required by these keys are as follows:

"instruments":
The instrument driver may return a Python list of PyICe channel objects, an instrument (AKA channel group) or a mixed list of instruments, channel groups and channels. Optionally the **"instruments"** key may contain a Python value of **None** but the key itself must always be returned.

"startup_list":
The instrument driver may optionally return a list of Python functions to be used to setup instrument channels to appropriate values that may not be their default. The startup order will be the order in which the startup functions appear in the startup lists and the order in which the mini-driver files are read (which should be alphabetical by mini-driver filename).

"temperature_run_startup_list":
The instrument driver may optionally return a list of Python functions to be used to setup instrument channels to appropriate values that may not be their default specifically when test scripts are being run over temperatures (e.g. enable the oven). The startup order will be the order in which the startup

functions appear in the startup lists and the order in which the mini-driver files are read (which should be alphabetical by mini-driver filename).

"cleanup_list":
The instrument driver may optionally return a list of Python functions to be used to *cleanup* each instrument. Usually this is a function that sets an instrument to a default or off state. The cleanup order will be the *reverse* order in which the cleanup functions appear in the cleanup lists and the order in which the mini-driver files are read (which should be alphabetical by mini-driver filename). This way the bench can be "unwound" from the way in which it was configured. Use of the cleanup function is highly recommended so that each subsequent test starts from a known starting condition and the bench is left in a safe state following the last test executed.

"shutdown_list":
The instrument driver may also return a list of Python functions to be used to prepare each instrument for release. This list is only run at the conclusion of a suite of tests, as opposed to the cleanup_list that is run between each test as well as at the conclusion. The order of execution is the same as startup_list and cleanup_list: the order the functions appear in the list and the order in which the mini-driver files are read (which should be alphabetical by mini-driver filename). One such use of the shutdown_list is setting an environment chamber to a human friendly temperature once the full temperature sweep is complete.

"temp_control_channel":
A special temperature-control meta-channel should be declared in the temperature controlling instrument's mini-driver file (environmental chamber, Thermostream™, etc.). This channel will be used as the outermost loop variable during each test run (or suite of tests) placing temperature control on the outermost iterator loop. This approach saves environmental resources such as nitrogen, time, noise, etc. All other adverse condition stimuli looping will be swept within each test script as needed.

The value of this key must be a PyICe channel object. A channel object is usually returned from of a PyICe channel declaration (by PyICe driver convention). There should be either zero or one of these declarations per project. Multiple declarations will result in a descriptive error and program stoppage.

The temperature channel is expected to be *blocking* until the chamber has settled and designated soak time has expired.

"special_channel_actions"
The instrument driver may optionally return a dictionary with key-value pairs of ***special_channel_actions*** channel names or PyICe channels and associated special action functions (Python function handles - uncalled). The functions can be defined within the driver file itself and can be actions such as alerting the bench operator that a limiting value was reached or an instrument is angry, for example. All special

action functions should take as arguments the channel, the latest log readings (which will be in the form of a dictionary with the channels as keys), and the test.

As stated above, the keys of the special_channel_actions dictionary can be either a channel name as a string or the channel object itself, and that is what will be returned to the associated function at each logging of the logger. If a channel name is given, treat the channel argument as a string in the function. If a channel object is used as a key, have the function treat its channel argument as a channel object as well.

```
from PyICe.lab_instruments.Franken_oven import Franken_oven

def populate(self):
    if 'Oven' not in self.interfaces:
        return {'instruments':None, 'cleanup_list':[]}
    oven = Franken_oven(self.interfaces['Oven'])
    temp_channel= oven.add_  channels(channel_name='tdegc')
    return {'instruments':[oven], 'temp_control_channel':temp_channel,
'special_channel_action':{'tdegc_sense':temperature_check}}

def temperature_check(self, channel_name, readings, test):
    if abs(readings[channel_name] - readings['tdegc']) > 5:
        print_banner('Whoa! Hold up! The oven temperature is drifting. Are you out of
nitrogen?')
```

CHAPTER 4 – PLUGIN DEPENDENCIES

The ***plugin_dependencies*** folder requires the presence of a **test_template.py** file. This file must contain at least a **Test_Template()** class. The project specific **Test_Template()** should inherit the generic PyICe **Master_Test_Template()** and may inherit any global data of interest as well. The project-specific **Test_Template()** class must be inherited into each test script within the project for the infrastructure system to work.

```
from global_data import Global_Data
from PyICe.plugins import Master_Test_Template

class Test_Template(Master_Test_Template, Global_Data):
    def __init__(self):
        self.populate_global_data() # method of Class Global_data()
```

If the archive plugin is being incorporated, an archive folder *method* may be specified in the project-specific **Test_Template()** class called **get_archive_folder_name()**. This method may be used to declare an archive folder name *specification* by concatenating various live values associated with the archive

data such as a date/time stamp, device ID, device version, data management version control information, etc.

```
from PyICe.plugins import Master_Test_Template

class Test_Template(Master_Test_Template):
    ...
    def get_archive_folder_name(self):
        return f'{datetime.datetime.utcnow().strftime("%Y_%m_%d_%H_%M")}_{self.operator}'
```

If no **get_archive_folder_name()** method is provided, P.I.E. will use a simple timestamped folder name schema for archives.

Alongside the test_template.py file there must be a file that contains a dictionary called **Project_Settings**. The dictionary is required to provide P.I.E. with critical project specific information that will be implemented throughout the testing process. The basic information that will be passed on is where the project can be found below the PYTHONPATH system environment variable via the key **project_folder_name** and the actual file path to that directory via the key **project_path**. More keys are required if plugins are used that require supplemental information.

```
Project_Settings={
"project_folder_name"  :'dummy_project',
"project_path"         : __file__[:__file__.index('dummy_project')+len('dummy_project')]
}
```

CHAPTER 5 – TESTS

Each parameter of interest will need its own script. It is strongly advised to factor the test scripts by each meaningful parameter to be measured and to have many test scripts rather than one big script or fewer test scripts. This will keep the workspace well organized and manageable. Each test script should be in its own folder as it will generate collateral files such as a database, one or more plots, helper scripts, etc.. Since each script will be in an appropriately named folder, the scripts, themselves, can all just be generically called ***test.py***. The scripts would be disambiguated by their containing folder.

Each **test.py** will be the main workhorse that contains the test-specific code on how to the measure the parameter of interest as well as how to optionally evaluate and plot the data. **test.py** should contain at least one Python class. The name of the class is unimportant, but it should inherit the project specific class **Test_Template()** from the file **test_template.py** in the ***plugin_dependencies*** folder.

Each test script will generally have this form:

```
from my_project.infrastructure.plugin_dependencies.test_template import Test_Template

class test(Test_Template):
    def customize(self):
        ### Optional Logger Customizations Here - See Example ###
    def collect(self):
        ### Collect Code Here - See Example ###
    def plot(self):
        ### Optional Plotting Code Here - See Example ###
```

Customizing the Logger

The **`customize():`** method is a hook to optionally change *this* test's PyICe logger before a data-table is created and the columns of the database are finalized. Recall that the namespace for the project is created from the instrument drivers and passed to each individual test as a PyICe logger. Changes to the logger (namespace) using *customize* are local to this test only. They will not persist past the execution of this script. Customize can be used, for instance, to remove excessive channels from a scanning voltmeter to save on test time or to add additional script-specific state-keeping channels or other Python objects if need be. Customize can also include changes to the bench that would be missed by the startup/cleanup functions in minidrivers, but should nonetheless be reversed between tests. This is accomplished through the test's reconfigure() method. Use the channel name to be altered and the new value of the channel as arguments, and the channel will be set to that value at the beginning of the collect method, and written back to the channel's original value after the collect is completed.

Collecting the Data

The **`collect():`** method is the main workhorse method of data collection. It will contain the code that sets up the environmental conditions for the test of interest and decides when to trigger the data to be logged. Generally, the collect method consists of one or more nested loops or *iterators* of adverse conditions such as voltage, current, frequency, etc. It is called once the instruments have been initialized and the database is ready to receive the data. **`collect()`** will run once for every value of temperature specified in the run command after writing the temperature value to the assigned temperature control channel.

Example **`customize()`** and **`collect()`** methods may look like this:

```
from dummy_project.infrastructure.plugin_dependencies.test_template import Test_Template
from PyICe import LTC_plot


class Test(Test_Template):
    def customize(self):
        channels = self.get_channels()
        channels.add_channel_dummy("dumduma")
        channels.add_channel_dummy("dumdumx")
        channels.add_channel_dummy("dumdumy")


    def collect(self):
        channels = self.get_channels()
        for a in [-1,0,1]:
            for x in [0,1,2,3,4]:
                channels.write("dumduma", a)
                channels.write("dumdumx", x)
                channels.write("dumdumy", x+a+1)
                channels.log()
```

Plotting the Data

The **plot()** method determines how the collected data is to be presented. **plot()** operates on the stored SQLite data from the local database. As such, **plot()** can be called and re-called any time after the data has been collected and stored. The PyICe module ***LTC_plot*** can be utilized in the plot method but any plotting routine could be used. The benefit of using LTC_plot is that plots can be regenerated for each successive data collection run resulting in a consistent look and feel as the product evolves through its developmental stages. Separation of data collection and visualization (plotting) is one of the crucial benefits of the PyICe workflow.

LTC_plot is, largely, a Matplotlib wrapper. There are many detailed examples of using **LTC_plot** in the **PyICe** examples folder.

```
    def plot(self):
        database = self.db
        table_name = self.table_name
        plotlist=[]
        G0 = LTC_plot.plot( plot_title  = "Sample Plot",
                            plot_name   = "Plot Name",
                            xaxis_label = "X-Axis",
                            yaxis_label = "Y-Axis",
                            xlims       = (-1, 5),
                            ylims       = (0,6),
                            xminor      = 0,
                            xdivs       = 6,
                            yminor      = 0,
                            ydivs       = 6,
                            logx        = False,
                            logy        = False)
        G0.add_horizontal_line(value=3, xrange=G0.xlims)
        colors = LTC_plot.color_gen()
        for a in database.get_distinct("dumduma"):
            database.query(f'SELECT dumdumx, dumdumy FROM {table_name}
                             WHERE dumduma is {a}')
            G0.add_trace(   axis        = 1,
                            data        = database.to_list(),
                            color       = colors(),
                            marker      = '.',
                            markersize  = 2,
                            legend      = f'EXAMPLE {a}')
            G0.add_legend(axis=1, location=(1, 0), use_axes_scale=False, fontsize=5)
        plotlist.append(G0)
        Page = LTC_plot.Page(plot_count=len(plotlist))
        [Page.add_plot(plot) for plot in plotlist]
        Page.create_svg(file_basename=self.name, filepath=self.plot_filepath)
        Page.create_pdf(file_basename=self.name, filepath=self.plot_filepath)
```

Chapter 6 – OPTIONAL PLUGINS

Introduction

Several features offered by P.I.E. are considered optional plugins. These include bench configuration management, bench image creation, limit evaluation, result correlation, traceability, notifications, and archiving.

To include one or more of these plugins, it is necessary to add them to a list in the Project_Settings dictionary within the plugin_dependencies directory. The plugin value should contain a list of strings of the plugins included in the project.

For example, the entry in Project_Settings may look like this:

```
"plugins"      :["bench_config_management",   "bench_image_creation",   "evaluate_tests",
"correlate_tests", "notifications", "traceability", "archive"],
```

Evaluating the Data

Ideally, every parameter of interest should have some requirement to which it is beholden. Once physical data is collected, it can be compared to pass/fail limits automatically using the **Evaluate Data** extension. This service will compare the collected results to declared limits and subsequently generate a .json file that logs the test script name, the traceability information of the test run, and the grades of all the named tests in the test script, ultimately giving the test script itself a PASS/FAIL grade.

To utilize this feature, either the project's ***test_template.py*** file will need a method called ***get_test_limits()*** which takes in a *test name* as an argument and must return a dictionary of information relevant to the test name, or a dictionary must be provided in the test file as an argument for the specific evaluation method's ***limits*** parameter. The dictionary must include, but is not limited to, the upper and lower limit of the tested parameter (keyed as "upper_limit" and "lower_limit"). How the ***get_test_limits()*** method acquires the test limit data is project dependent and is left to the project leader to determine. One option, for example, would be to have a dictionary of dictionaries, with the primary dictionary keys being the test names, and the values being the dictionaries with limits – also keyed by "upper_limit" and "lower_limit", for example.

To perform the evaluation, each test script must have a class method of ***evaluate_results()***. Within this method, several data formatting options are available for submitting the data for evaluation.

evaluate

In the first method, ***evaluate()***, the test name, matching the primary key of the dictionary received from ***get_test_limits()*** above, is sent as an argument of the method ***self.evaluate()***. Also sent with the method call would be a SQLite search string of ***values***, the adverse ***conditions*** channels list over which the result should be sorted, a SQLite ***where_clause*** string to filter or limit the data (rows) to be evaluated, and a ***limits*** dictionary if the ***get_test_limits()*** infrastructure call will not be used. The values field can be any SQLite compliant string containing the PyICe channel or can be a columnwise calculation of channels to be evaluated. An example may be a simple calculation of a voltage channel times a current channel for power (e.g.: `"iout1 * vout1"`).

P.I.E. will assemble these parameters into a SQLite query, extract the data from the database and compare all readings to the limits registered in the ***get_test_limits()*** infrastructure call or from the ***limits*** argument.

Note that the ***conditions*** parameter does not limit or filter the data, it is a disambiguation list to help sort the results by adverse condition. The corresponding conditions of each row in the ***values*** field will be listed in the evaluation report and ***conditions*** will help determine under which adverse condition the parameter of interest passed or failed.

The ***where_clause*** on the other hand, is a filter on the data retrieved from the database. It will limit the number of rows returned, thereby leaving out cases of the collected data that may not be wanted for the evaluation or are otherwise for information only.

```
from my_project.infrastructure.plugin_dependencies.test_template import Test_Template

class test(Test_Template):
    ...
    def evaluate_results(self):
        self.evaluate( name           = 'CH0_EFFICIENCY',
                       values         = 'vmeas_vout0 * imeas_vout0 / vmeas_avin / imeas_avin * 100',
                       conditions     = ['tdegc','vmaina_force','clock_source'],
                       where_clause   = 'tdegc < 100 AND vmaina_force is 3.3',
                       limits         = {'upper_limit':98, 'lower_limit':90})
```

evaluate_query

The second method, ***evaluate_query()***, can receive a raw SQLite query with the value to be evaluated in the first column position (after **SELECT**) and the disambiguation conditions enumerated in all successive columns. In this case the query is sent on the SQLite database unmolested and the user is responsible for all SQLite syntax, table reference and ***where*** clauses, etc.

As with the ***evaluate()*** method, an optional ***limits*** argument can be provided in lieu of relying on the ***get_test_limits()*** infrastructure call.

```
from my_project.infrastructure.plugin_dependencies.test_template import Test_Template

class test(Test_Template):
    ...
    def evaluate_results(self):
        self.evaluate_query(   name  = 'CH0_FAULT',
                               query = f'SELECT RST_interrupt_count, tdegc, vmaina_force, clock_source
                               FROM {self.table_name}')
```

evaluate_rawdata

In the third method, ***evaluate_rawdata()***, a list of values can be sent to be evaluated. This method can be used if the pre-processing calculation of the data to be evaluated exceeds the complexity of a single **SELECT** SQLite query, for instance. In this case, the author may use whatever method is needed to extract the data from the database, **self.db**, massage the resultant data into a one-dimensional Python list, and

send the list on to the evaluate module. This method can be called multiple times for the same test, and all submitted values for a test will be grouped with those of matching conditions.

As with the ***evaluate()*** method, an optional ***limits*** argument can be provided in lieu of relying on the ***get_test_limits()*** infrastructure call.

```
from my_project.infrastructure.plugin_dependencies.test_template import Test_Template

class test(Test_Template):
    ...
    def evaluate_results(self):
        # Complex SQLite query upon self.db/self.table_name resulting in Python list variable 'values'.
        self.evaluate_rawdata( name = 'EXAMPLE',
                               values = values,
                               conditions = {'tdegc':25}
```

evaluate_db

Finally, in the ***evaluate_db()*** method, the script can retrieve the database handle, **self.db**, perform a query directly upon it, and submit it with ***evaluate_db()***. Of course, the parameter ***name*** field is still required but the database does not need to be returned, the infrastructure already has its handle. Recall that SQLite databases are stateful, the rows and columns will be filtered by those in the query command.

The column order follows the same rules as ***evaluate_query()***. The first column (single channel or compound channel calculation) is the data to be evaluated, any following columns are the adverse conditions for sorting.

As with the ***evaluate()*** method, an optional ***limits*** argument can be provided in lieu of relying on the ***get_test_limits()*** infrastructure call.

```
from my_project.infrastructure.plugin_dependencies.test_template import Test_Template

class Test(Test_Template):
    ...
    def evaluate_results(self):
        self.db.query(f'SELECT vout0_meas, tdegc, vmaina_force, clock_source FROM {self.table_name}')
        self.evaluate_db(name='CH0_FAULT')
```

register_test_failure

At any time during the ***evaluate()*** script method, a test can be forced to result in a FAIL using the ***self.register_test_failure()*** method, regardless of what other data is submitted via the evaluation methods. Pass in the name of the parameter name that is to FAIL (argument is *name*), along with a string the reason for the forced failure (argument is *reason*). Optionally, conditions and query can be passed along as well for record keeping (by arguments *conditions* and *query*, respectively).

```
from my_project.infrastructure.plugin_dependencies.test_template import Test_Template

class Test(Test_Template):
    ...
    def evaluate_results(self):
        if tdegc>100:
            self.register_test_failure(name='CH0_FAULT',
                                       reason='Condition failed',
                                       conditions={'temperature':tdegc})
```

Correlate Data

For instances where hard limits of a parameter aren't as important as how test values measure up to another set of reference values, there is the correlation plugin. While it shares similarities with the evaluation plugin, this variation requires a little more info from the user to get a PASS/FAIL grade for a parameter.

As with the other plugins, it must be listed under the "plugins" dictionary entry in **Project_Settings** as 'correlate_tests' for it to be incorporated in the testing process. Scripts that have parameters to correlate will need their own method ***self.correlate_results***, and within that method for a given parameter ***self.correlate_data*** shall be called. This method has parameters for the *name* of the test, *reference_values* and *test_values* as lists, *spec* which will receive a string of either a percent symbol or a dash, a dictionary of *conditions* for record keeping, and the optional *limits*. The parameter must still have limits that can be acquired using the ***self.get_test_limits*** method in **test_template.py** or in the test via the *limits* parameter, but those limits will now be the acceptable range from the reference values the test values can be, as opposed to the window of acceptable limits that are submitted. The *spec* parameter determines whether the limits indicate the test values can be within, for example, plus or minus 2% of the reference values or plus or minus 2 units.

Traceability Data

For any given test, there may be two classes of data to be collected, the measurement data, of course, but also traceability data, or meta-data. Traceability data may be information pertinent to the setup of the test being executed, information about the target device itself (i.e. serial number), operator or bench equipment information, etc.

While normal PyICe logging retrieves all known measurement channels for each logger call, utilizing the traceability plugin allows meta-data to be collected and logged one time (one row in its own table) at the beginning of the test run. The meta-data table will be stored within the same database as the table with the repeated measurement data storage but its table name postpended with the term "**_metadata**".

To utilize the traceability plugin, 'traceability' must be in the list of plugins in the **Project_Settings** dictionary found within the **plugin_dependencies** directory, and the dictionary must contain a new entry with a key of 'traceability_items'. The value is a new dictionary comprised of keys representing the traceability item names and values of traceability functions (function pointers – uncalled). The traceability functions will only be called at the beginning of a **collect()** run and not during other activities

such as plotting, evaluating, etc. The keys become the channel names to which the meta-data table columns are labelled. The functions will, later, retrieve the values to be entered into those corresponding data columns. Each traceability function must take an argument of **test**. The **test** argument may not necessarily be needed by all traceability functions, but may be needed by some, for instance to retrieve the test's logger.

```
from dummy_project.infrastructure.plugin_dependencies.metadata_gathering_fns import
get_traceability_items

Project_Settings={
"project_folder_name"  :'dummy_project',
"project_path"         : __file__[:__file__.index('dummy_project')+len('dummy_project')],
"plugins"              :["traceability"],
"traceability_items"    : get_traceability_items()
}
```

As can be seen in the example above, the function returning a **traceability_items** object is abstracted to separate file. If desired, the traceability gathering function could be included within the **Project_Settings** file itself.

In the example shown above, the **traceability_items** object was created in a separate project-specific **traceability.py** file as shown below.

```
def add_bench_operator(test):
    return os.getlogin()

def get_traceability_items(test):
    return {'bench_operator' : add_bench_operator}
```

Documenting Bench Connections

By utilizing a clear programming language like Python, documenting the procedure used to measure the test data is nearly automatic.

To improve recollection of how a test connection physical set up was configured, a system of documenting physical connections within the bench may be exploited. Using the **declare_bench_connection()** method, each test may document all connections on the bench to ensure that *this test* runs correctly. Alternately, if the project leader has set up the default bench connections in the infrastructure folder, the test specific **declare_bench_connections()** method may add or subtract connections needed to describe how this test must be configured.

Ideally all physical connections on an evaluation project would be static throughout the project. Unfortunately, reality dictates that some connections will need to be made and some broken between

tests. To collect data as efficiently as possible, it behooves the project manager to group test scripts together into a collection of tests or a *test suite*. Bench Configuration Management is a tool that can be used to electronically represent all the connections of the bench on a test-by-test basis within a simple data structure. By doing so, it is now possible to determine if all the tests within a suite are connection compatible with each other. If they are not compatible, a warning or error can be thrown to alert the operator thereby preventing wasted time on a bad setup.

If the traceability plugin is used, the bench connections data will automatically be added to the meta-data log table for later retrieval, auditing, technical review, etc. The data can even be converted to a connection diagram using ***graphviz***.

Bench Configuration Management

After adding 'bench_config_management' to the 'plugins' list in **Project_Settings**, two things must be added to the project infrastructure to make Bench Connection Management work: a method in the project's **Test_Template()** class called ***_declare_bench_connections(self, component, connections)*** and a method in each test script called ***declare_bench_connections(self, components, connections)***.

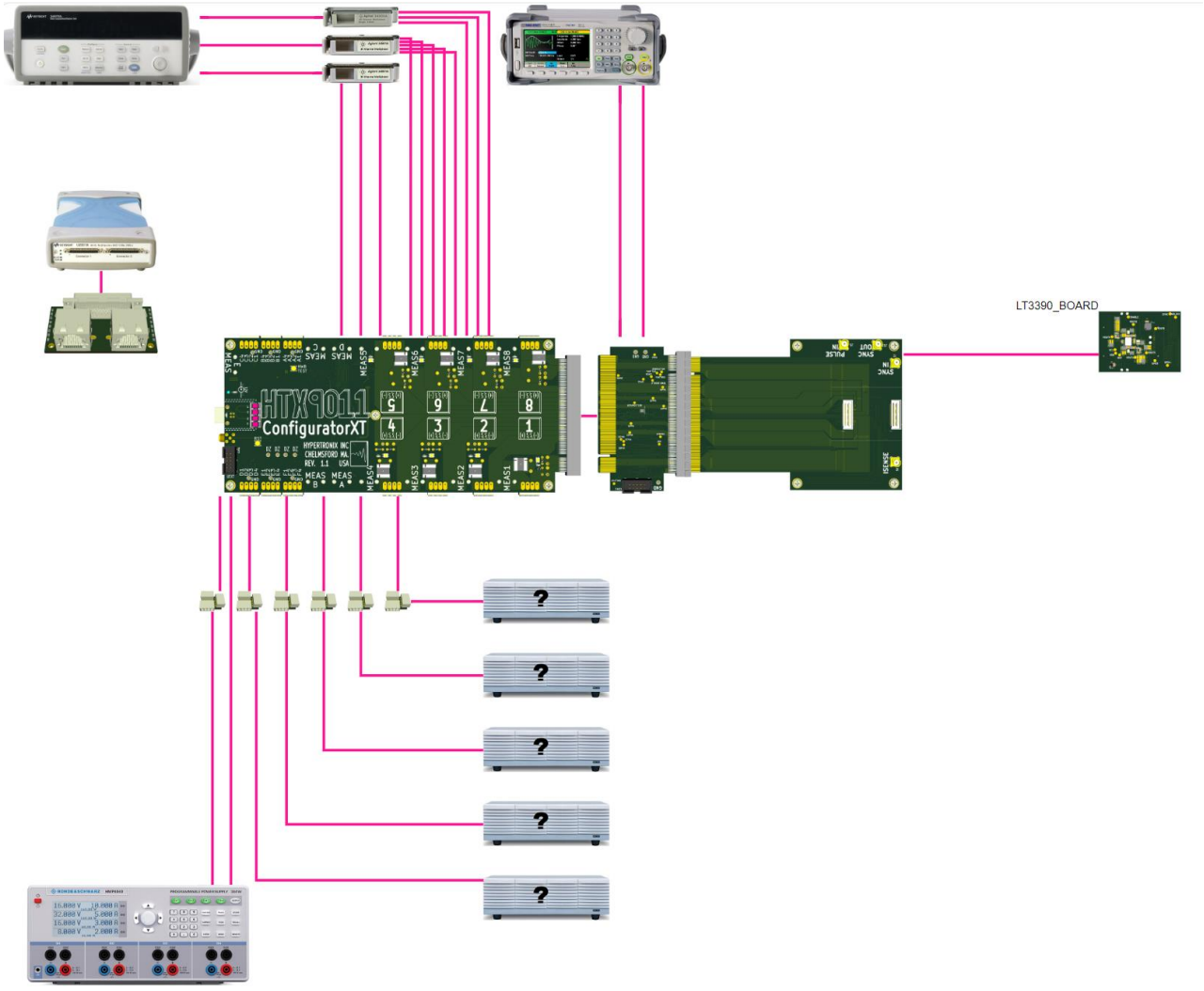
The former will add all the instruments used in the project to the **self.component** attribute and add all the general connections every test of the project will use to **self.connections**. P.I.E. will call the latter, which will add and remove connections between the instruments based on how the specific test is executed.

self.components and **self.connections** are PyICe objects created in a higher level script and passed down into the **Test_Template** object to flesh out the details of *this* test. See tutorial number eight for more info on how to populate component and connection collections.

Bench Image Creation

In addition to electronic logging of the components and connections made for a test, a visual representation can be automatically generated for use in reports. This feature is optional and can only be used if the Bench Configuration Management extension is deployed and 'bench_image_creation' is in the 'plugins' list in **Project_Settings**.

In addition to the requirements for the Bench Configuration Management extension, the user will also have to provide the images used for the instruments and the locations of the images on the page to be created. These will be accessed through a new entry in **Project_Settings,** 'bench_image_locations" and a dictionary of images and their locations. Again, see tutorial number 8 for more information and an example PyICe-ADI/PyICe · GitHub. The output of the image creation feature is an .svg file which can be viewed within a web browser. Hovering over components and wires reveals the name of the object underneath. Below is an example of the automatically generated Bench Image Creation feature.



Data Archival

Once data collection is complete, there are a few options for what to do with it. The data archiving extension will assist in labelling it and safely stowing it away for easy retrieval later. This can prove important for reviewing the data and regression analysis through time. The only requirement to include this extension is to add the string ***"archive"*** to the 'plugins' list in the **Project_Settings** file.

Every complete run of a test script will result in a copy of the database table (and its meta-data table, if traceability is also incorporated) to an archive created in the same directory as the test script. The copied database will be placed in a directory automatically named after the date and time the data was collected for easy sorting. Optional hooks can be used to provide a template function upon which the archive folder name can be embellished if desired.

A **replot_data.py** python script will also automatically be placed beside the archived database. This is a helper script that can run the test's plot method against the archived data and generate plots in the archived directory.

If the evaluate extension is included in the plugins registry, then a **reeval_data.py** helper script will be automatically placed beside the archived database. Executing this helper script will run the test's **evaluate_results()** method using the archived data and generate a .json report in the archived directory.

Notifications

For longer tests, continuous monitoring of the lab bench becomes untenable. The **notifications** extension can send out alerts to a plurality of emails or texts at certain milestones of a test.

The milestones at which alerts are sent out include when a test begins, a new temperature begins, a test crashes, a test successfully resolves, and, finally, when plots are ready. In the last case, whatever plots are added to the test's autogenerated empty list plot_list in the test script's plot method are sent to the recipient.

To deploy this extension, a Python file named after the computer user's username must be present somewhere within the project's infrastructure folder (i.e. ***<login_name>.py***). Uppercase username characters should be converted to lowercase for the file name. This filename disambiguation system customizes the alerts to target the recipient list chosen by the *current* computer user on a multi-person project.

The **login_name.py** file can contain an **add_notifications_to_test_manager ()** function with the user's information added to the passed argument **test_manager** as shown in the example below. The lambda function with its various arguments is only used in higher level functions. The only part the user needs to alter from the example is the email address. The rest can be copied verbatim from here. Alternatively, the file can also contain a **get_notification_targets()** function that returns a dictionary of the intended recipients info. The dictionary shall have 'emails' and 'texts' as keys, and for 'emails' a list of strings of email addresses serve as the value, and for 'texts' a list of tuples of strings consisting of the phone number and the carrier of the user.

```
from PyICe.lab_utils.communications import email

def add_notifications_to_test_manager (test_manager):
    alerted_mail_address = email('your.name@analog.com')
    test_manager.add_notification(lambda msg,
                                  subject=None,
                                  attachment_filenames=[],
                                  attachment_MIMEParts=[]: alerted_mail_address.send(msg,
                                  subject=subject,
                                  attachment_filenames=attachment_filenames,
                                  attachment_MIMEParts=attachment_MIMEParts))
def get_notification_targets():
    targets =   {   'emails':['your.name@analog.com'],
                    'texts' :[('phone_number', 'carrier')]
                }
    return targets
```

Helper Scripts

To operate a **test.py** script, helper scripts such as **run.py**, **debug.py**, **plot.py**, etc. may be made alongside the test script. Examples of helper scripts will be shown later. The **run.py** and **debug.py** scripts will be *live* tests scripts in that they will activate the test equipment and collect data. **debug.py** is simply a version of the run script with the debug option set to **True** when creating the plugin manager. The debug option is simply passed to each test as **self.debug** and can optionally be used to reduce the resolution or range of each test to save time while debugging the test or its tuning its plotting methods.

The **plot.py** helper script only plots the last data collected from a run or debug. **plot.py** may be called repeatedly to fine tune the plots using *LTC_plot* from PyICe. It will only call the **plot()** method of the test script and therefore will not operate any test equipment or collect any new data. Of course, calling **plot.py** only makes sense if a database is present from a previous run or debug call in the folder.

These helper scripts will likely be identical for all tests. The **run.py** script, for example, will look like this:

```
from PyICe.plugins.plugin_manager import Plugin_Manager
from dummy_project.infrastructure.plugin_dependencies.project_settings import Project_Settings
from test import test

pm = Plugin_Manager(settings=Project_Settings)
pm.add_test(test) # Optionally import and add more tests to make a "test suite".
pm.run()
```

Whereas debug.py may look like this:

```
from PyICe.plugins.plugin_manager import Plugin_Manager
from dummy_project.infrastructure.plugin_dependencies.project_settings import Project_Settings
from test import test

pm = Plugin_Manager(settings=Project_Settings)
pm.add_test(test, debug=True) # Optionally import and add more tests to make a "test suite".
pm.run()
```

And plot.py may look like this:

```
from PyICe.plugins.plugin_manager import Plugin_Manager
from dummy_project.infrastructure.plugin_dependencies.project_settings import Project_Settings
from test import test

pm = Plugin_Manager(settings=Project_Settings)
pm.add_test(test) # Optionally import and add more tests to make a "test suite".
pm.plot()
```

APPENDIX 1 – Program Flow