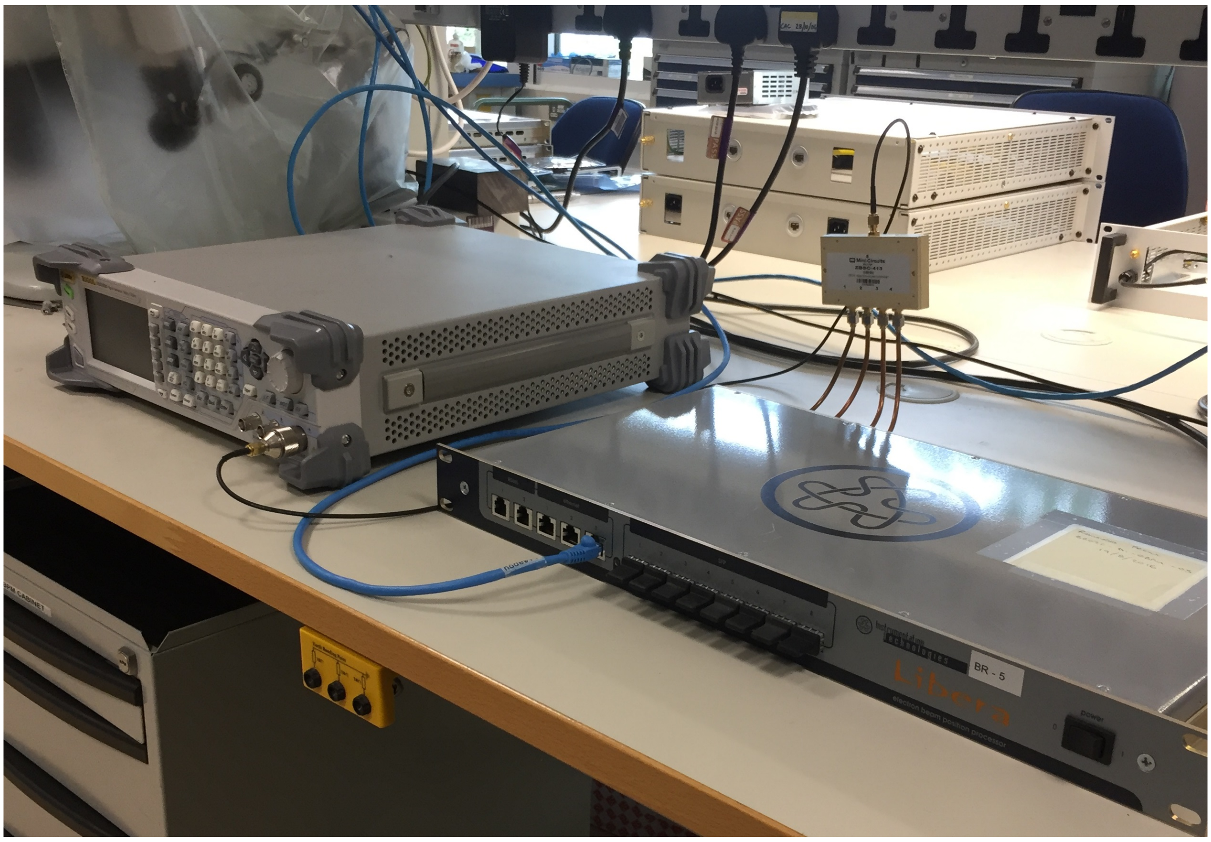
Diamond Light Source Ltd
Beam Diagnostics Group

Date: July 20, 2017
Daniel Harryman, Alun Morgan

Beam Position Monitor (BPM) Test Framework Design Report & Operation Guide



Contents

List of Figures

Preface

This project has been undertaken as part of the Science and Technology Facilities Council (STFC) graduate scheme. Employees on the STFC graduate scheme undertake a three month placement in a different department to gain knowledge of the wider work that STFC does, and to network with other scientists and engineers. Daniel Harryman, who works as an Electronics Engineer for the ISIS Beam Diagnostics group, spent three months in the Diamond Light Source Ltd Beam Diagnostics group as his STFC placement, under the supervision of Dr Alun Morgan.
The project undertaken for the three month placement, was to develop a software framework to test the performance and functionality of several different Beam Position Monitor readout devices. This report contains the development of the final software framework, and an operational guide to use the framework.

The software described in this document can be found here:
`https://github.com/dharryman/BPM_Test_Framework`

1 Introduction

1.1 Background

This document details the objectives, design, and implementation of the Beam Position Monitor test framework developed for use at Diamond Light Source, primarily using scripts, that are written in Python. The tests described in this manual will be for testing the electronic readout devices used to analyse the signals that would typically come from physical pickups such as button pickups. Four of these pickups are used to calculate the position of beam by comparing the relative signals at each pickup. Each pickup is assigned a name of *A*, *B*, *C*, or *D*, depending on their location. Figure 1 shows the typical naming convention of the pickups. To calculate the position, equations 1 and 2 are used the calculate the *X* and *Y* position of the beam respectively. Both of the k_x and k_y values are based on the beam pipe dimensions used in the accelerator. For Diamond, the values of k_x and k_Y are both set to 10 mm.

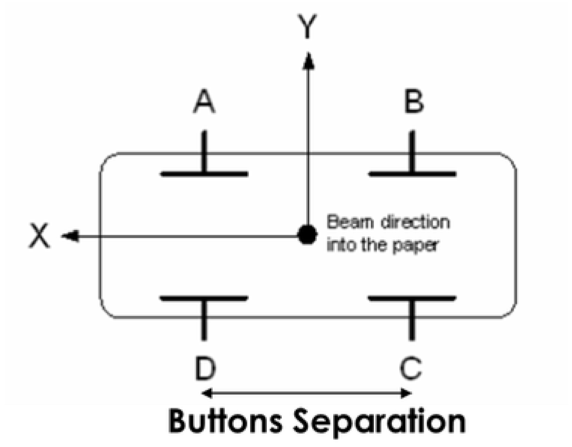


Figure 1: Simplified layout of a four button BPM

$$X = k_x \frac{(V_a + V_d) - (V_b + V_c)}{V_a + V_b + V_c + V_d} \tag{1}$$

$$Y = k_y \frac{(V_a + V_b) - (V_c + V_d)}{V_a + V_b + V_c + V_d} \tag{2}$$

A simple test set-up that will deliver an equal signal into all four BPM pickups can be seen in figure 2. For this type of test there is a single output source that goes through an equal four way splitter, into a BPM's four inputs *A, B, C, D*. It's worth noting that some more advanced tests may require the use of additional hardware that is not shown in figure 2, such as, a programmable attenuator, to deliver unequal signals into each pickup and thus move the position calculated by the readout electronics.

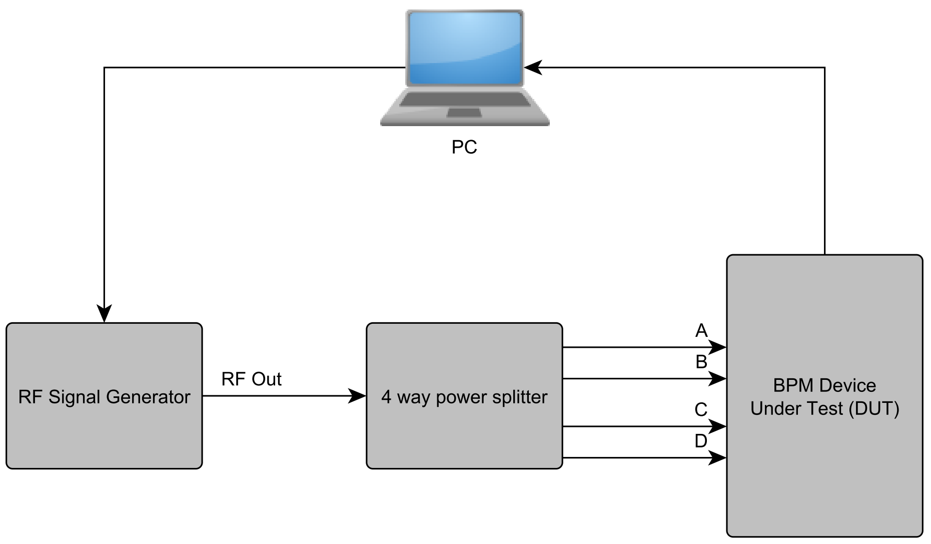


Figure 2: Simple test setup

The goal of this project is to make a software framework that makes it easy to test multiple parameters of different BPM readout devices. Simple tests will test how stable the position is as beam current values or the bunch structure changed, while others will feed different signals into each pickup and compare the position read by the device and the predicted value.

1.2 Objectives

The final product or program produced has a specific set of objectives and goals. The task is not only to create a test application, but to ensure the application created can be added to, and extended over time, allowing for the supported hardware, and the tests that can be run, to grow.

- Create an application or program that will perform tests on the Beam Position Monitor readout instruments, that are used at Diamond.
- Ensure that the hardware and protocols are abstracted from the tests, so that the process of changing the hardware used in the tests will not require any of the tests to be rewritten.
- The final program must be scalable so that new tests can be added and amended easily, as well as making it easy to run a subset of tests or change the order that the tests are run.
- Once a set of tests has run, this should produce a report, that will show or record the results of the tests.
- The software should be designed to good practices and use appropriate commenting styles, unit testing, and simulation where possible.

2 Software Architecture

2.1 Software Design

Given the objectives stated in section 1.2 the software architecture used must be well designed and thought out. This section of the report details the software architecture that has been used to achieve these objectives.
When making a series of tests typically a scripting language will be used. The two most popular ones used in Diamond are, Matlab, and Python. Python has been chosen as it lends itself to object oriented programing slightly more than Matlab, and object oriented programming is used extensively in this project to abstract hardware and protocol layers. Python is also open source so the final program can be easily ported to a different machine without a license needed to run the program.
The main building blocks of the program are the tests, the hardware, and the report. As such a typical test will be a script that has an object for each hardware device, and another for the report that the data will be saved to. Figure 3 shows this, the arguments for each test will be an instance of each of the classes shown in the figure 3, as well as the test parameters themselves. If more hardware devices are needed, then that will also need to be an object that is used in the test's arguments.

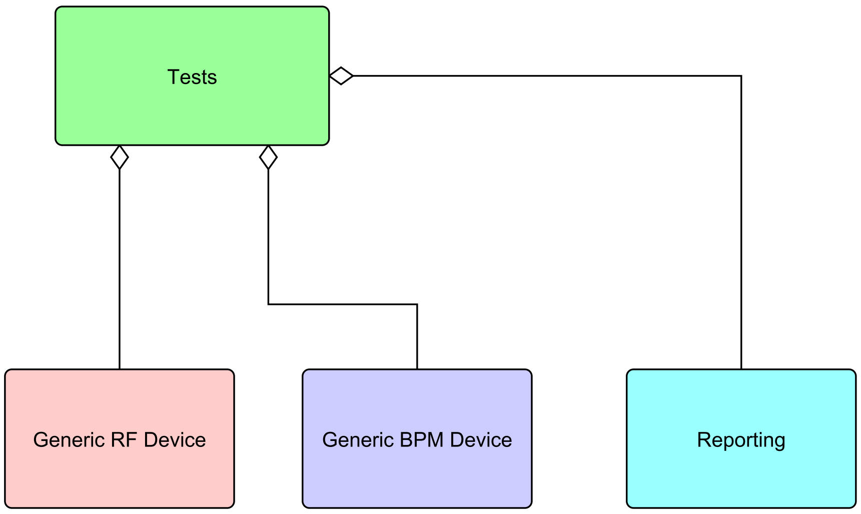


Figure 3: Test Aggregation

A launcher script will instantiate all of the objects that are to be called by the tests, then uses these objects as inputs to the relevant test scripts. Figure 4 shows a very simplified version of the main launcher script. This instantiates an object of each of the type of hardware and a report, before feeding all three into a function from the *test* script where the tests are carried out and their results recorded to the report. If the amount, or type of tests were wanted to be changed, the functions called by this are simply added, removed, or have their arguments changed.

```python
import RF_Source_class  # import the class used to speak to the RF signal generator
import BPM_class  # import the class to speak to a BPM readout device
import test  # import a script that will test a parameter on the BPM using the RF source
import Report  # import a class used for creating reports

rf = RF_Source_class()  # instantiate the RF object
bpm = BPM_class()  # instantiate the BPM object
report = Report()  # instantiate the report object

test(rf, bpm, report)  # run the test using the desired hardware and report
```

Figure 4: Simplified Python Test launcher

2.2 Hardware Abstraction

Figure 4 shows how one calls different tests with specific hardware instances. What would happen if a different piece of hardware was to be used instead? In keeping with the objectives given in section 1.2 the ideal solution would require no changes to the code used in the test scripts. This can easily be achieved using an object oriented approach. Generic classes of each hardware type are setup with specific API methods. The children of the generic objects then overload these methods that are called by the test. Figure 5 shows an expanded view of figure 3, the tests are still given an instance of each piece of hardware, but as long as the children of the generic objects use the same APIs then the tests will still work.

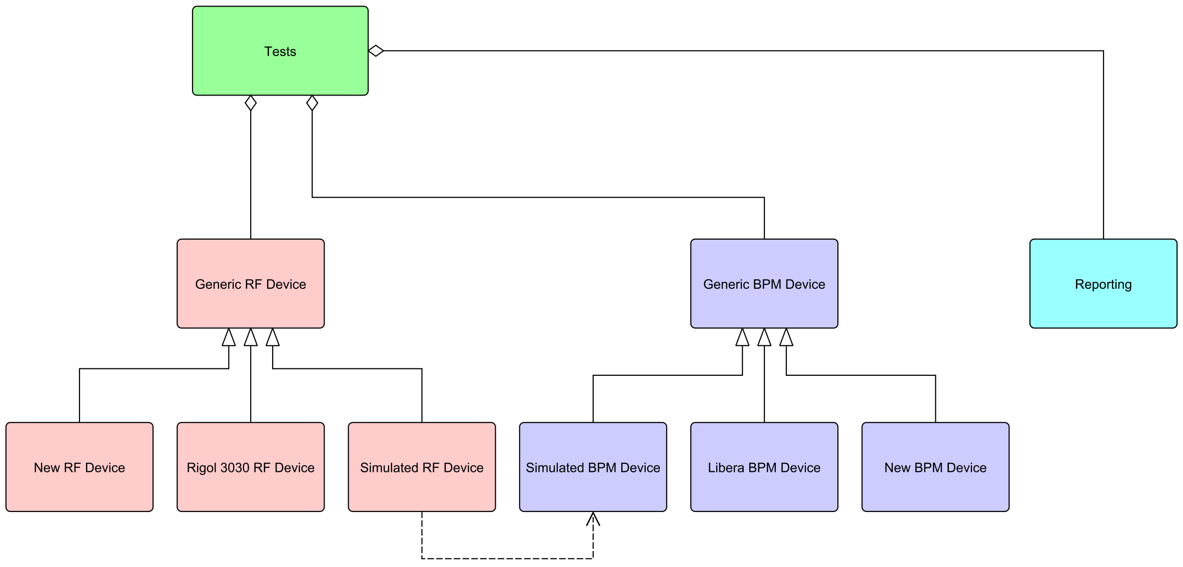


Figure 5: Hardware Abstraction Using Classes

Shown in figure 5 the *Simulated BPM device* has a dependency on the *Simulated RF Device*, this is because there is no physical connection in the real world connecting the output signals from the RF hardware to the BPM hardware, so this is done in software instead with the simulators.
To the test script, is doesn't matter if the exact implementation of communicating with the RF device is done over Telnet, USB, or a different standard, as that is done in the specific API implementation, and abstracted away from the test. Figure 6 shows the implementation of the same function in three different classes, one is the generic parent, one is a simulated version of RF hardware, and the third is the one that communicated with the *Rigol3030* device. Notice how the generic method is decorated with the *@abstractmethod* decorator. This means that instantiating an object with this method is not possible and will result in an error, these methods *must* be over ridden by it's child classes. For the simulator class, it simply feeds out it's device ID that is constructed when the object is. For the *Rigol3030* however a Telnet call is made to a physical device with the **IDN?* command, the reply is then returned as the device ID. Notice that for each of the methods the name, and arguments are the exact same, this is required on all of the over-ridden methods, and it is good practice to do the same for the return type as well. This means using one piece of hardware, or another, or a simulator, will only require a change in the instantiation of the object, all of the code in the tests will remain the same.

```python
# Generic RF Device Class
@abstractmethod
def get_device_ID(self):
        pass
```

```python
# Simulated RF Device Class, inherits from "Generic RF Device Class"
def get_device_ID(self):
        return self.DeviceID
```

```python
# Rigol 3030 Device Class, inherits from "Generic RF Device Class"
def get_device_ID(self):
        self.DeviceID = self._telnet_query("*IDN?")
        return self.DeviceID
```

Figure 6: Python Hardware Abstraction

2.3 Unit Testing & Simulation

Unit testing has been used to test individual class methods, where as system tests are done using simulator classes. For the unit tests the calls to the hardware are mocked out and only the methods functionality is tested. This normally includes checking that the function is called, and that given specific arguments the correct values come back.
Figure 7 shows a very simple unit test. An object of *hardware_class* is instantiated, then it's *get_device_ID* method is replaced with *mock_get_device_ID*. The unit test then checks that the reply is correct, and that the mocked function was called. This technique can be used to remove all calls to hardware and just check the functionality of the code.

```python
def mock_get_device_ID():
    return "device_ID"
```

```python
class ExpectedDataTest(unittest.TestCase):
  def setUpClass(cls):
          # Stuff you only run once
          super(ExpectedDataTest, cls).setUpClass()

  def setUp(self, mock_telnet, mock_telnet_read, mock_telnet_write):
          # Stuff you run before each test
          self.object_under_test = hardware_class()

  def tearDown(self):
          # Stuff you run after each test
          pass

  @patch("hardware_class.get_device_ID", side_effect=mock_get_device_ID)
  def test_device_ID(self, mock_dev_ID):
          self.assertEqual(self.object_under_test.get_device_ID(),"device_ID")
          self.assertTrue(mock_dev_ID.called)
```

Figure 7: Unit test, to test *get_device_id* method of *hardware_class*

This can be used to ensure input arguments are of the correct type and value. For example a frequency can only be set as a positive number, at the start of a function that takes a frequency value as an input argument, the value is checked to ensure it is of a numeric type, and that it is a positive number. If either one of these things is not true either a *TypeError* or *ValueError* can be raised. Unit tests are used to feed the methods arguments that will cause the error and test that the appropriate ones are raised when desired.
Simulation is done to check one of the test scripts quickly and reliably. Simulation can be used to ensure that the plotting and reporting functionality of the tests works as desired without actually needing to connect the hardware. Communication with the hardware is slower than that with a virtual device meaning code iterations can be made much faster, and as long as the simulation class or test parameters are not changed, the results will be fixed, this can help with formatting changes.

3 Instrument Control

3.1 Instrument Control Class Structure

When abstracting hardware and protocols there are a number of ways to do this using class inheritance. The classes used in this framework have been grouped in terms of hardware family, for example; all the classes associated to a specific model of BPM readout device inherit from the class *Generic_BPMDevice*. As detailed before in section 2.2 this is what allows the tests to use multiple types of hardware in a single test.
Many of the hardware classes use the same protocols to communicate with their associated piece of equipment, most notably *Telnet* and *EPICS*. Each of the classes that communicate using these protocols have private methods that perform read/write calls to process variables using *EPICS* or send SCPI commands via *Telnet*. As this is done in multiple devices, device classes for *EPICS* and *Telnet* could have been created with these functions inside them. Multiple inheritance could then use used to ensure the APIs from their hardware type family are picked up as well as these methods to make hardware calls easier. After considering this however, it was decided that a flatter class structure that kept each family fully separated was simpler, and so each class only has a single inheritance. the result is that some classes may have private methods that are repeated in several different classes.
For each class method the Google style of docstrings has been used. this helps specify the types of input arguments and return types given that python is a very loosely typed language. Figure 8 shows a typical method with a full docstring.

```
@abstractmethod
def set_frequency(self,frequency):
    """Abstract method for override that will set the output frequency.
        Args:
            frequency (float): Desired value of the output frequency in MHz
        Returns:
            float: The current frequency value as a float and assumed units.
            str: The current output frequency concatenated with the units.
    """
pass
```

Figure 8: Abstract method with Google style docstrings example

Each API will have a docstring that should be as detailed as figure 8. For simplicity, and to save space, when code is shown in this report many of these docstrings will be reduced or removed from code snippets, but they will still be present in the source code.

3.2 RF Signal Generators

For the tests, an RF signal Generator is used to excite the BPM readout device inputs and simulate the type of signal they would receive when a connected button pickup that is excited by the particle beam. Much like the BPM readout devices, the RF signal generator is also hardware abstracted, this makes it easier to change what signal generator is used should something happen to the *Rigol3030* currently used, or should another facility want to use the program, and not have access to the same model of RF signal Generator.
Figure 9 shows all of the APIs that the class *Generic_RF_SigGen* contains. All of these must be overridden in the child classes to ensure that all classes in the family have a minimum level of functionality. The name of the method quite clearly states what it's functionality is, but if more details on these methods are desired, the docstrings will contain more details about how they're implemented and their functionality. For the methods that set and get parameters, such as *set_frequency* or *get_output_power* there is a return type of a tuple that contains both a double and a string, this is because the double is more useful when plotting information or performing analysis, and the string is there to concatenate units to the number.

RF API List			
RF API Name	Argument Types	Return Types	Units
get_device_ID	self	string	NA
set_frequency	self, double	double, string	MHz
get_frequency	self	double, string	MHz
set_output_power	self, double	double, string	dBm
get_output_power	self	double, string	dBm
turn_on_RF	self	boolean	NA
turn_off_RF	self	boolean	NA
get_output_state	self	boolean	NA
set_output_power_limit	self, double	double, string	dBm
get_output_power_limit	self	double, string	dBm

Figure 9: Abstract RF Signal Generator Class APIs

As seen from figure 9 all of the method APIs have the input argument *self*, this is not actually needed to be entered by the user when calling one of these APIs as python will do this automatically when a method is called from an object.
For this project so far only one RF signal generator has been used, a *Rigol3030*. This device communicates using *Telnet*, and as such specific private functions have been implemented inside the class that are then used in these APIs. Figure 10 shows the initialize method as well as the private *Telnet* methods that simplify the communication to the *Rigol3030*. Using these simple methods SPIC commands such as **IDN?* or *FREQ 500 MHz* can be sent to the device to change the parameters of the instrument. Figure 11 shows the *set_frequency* method, this checks that the input argument is valid, then sends the command to the hardware to change the output frequency, notice how this method calls the *_telnet_write* private method. All of the APIs in this class do something very similar and make it easier to communicate with the instrument without needing to know all of the SCPI commands.
The simulated RF class inherits from the same generic RF class that the *Rigol3030* does, as such, it has the same APIs just implemented differently. Instead of making *Telnet* calls to hardware, it simply has a number of private variables that store a virtual frequency and power level, as well as an output state. These are then used when API calls are made to set and get such values. Were another RF signal generator needed to be added to the framework instead of the simulated class or

the *Rigol3030* this would be only require making a new class that inherited from the same *Generic_RF_SigGen* class and overloading all of the APIs listed in figure 9.

```python
class Rigol3030DSG_RFSigGen(Generic_RFSigGen):

  def __init__(self, ipaddress, port = 5555, timeout = 1, limit=-40):
    # timeout for the telnet comms
    self.timeout = timeout
    # connects to the telnet device
    self.tn = telnetlib.Telnet(ipaddress, port, self.timeout)
    # gets the device of the telnet device, makes sure its the right one
    self.get_device_ID()
    # turn off the RF output
    self.turn_off_RF()
    # set the RF output limit
    self.set_output_power_limit(limit)

  def _telnet_query(self, message):
    self._telnet_write(message)
    return self._telnet_read()

  def _telnet_write(self, message):
    # Checks that the telnet message is a string
    if type(message) != str:
      raise TypeError
    # Writes a telnet message with termination characters
    self.tn.write(message + "\r\n")

  def _telnet_read(self):
    # Telnet reply, with termination chars removed
    return self.tn.read_until("\n", self.timeout).rstrip('\n')

```

Figure 10: Rigol3030 Private Methods to communicate over Telnet

```python
def set_frequency(self,frequency):
  # check the input is a numeric
  if type(frequency) != float and type(frequency) != int:
    raise TypeError
  # check the input is a positive number
  elif frequency < 0:
    raise ValueError
  self.Frequency = frequency
  # write the frequency to the device in MHz
  self._telnet_write("FREQ " + str(self.Frequency) + "MHz")
  return self.get_frequency() # get the frequency after writing
```

Figure 11: Method that changes the output frequency of the Rigol3030

3.3 Gating Devices

A gating device is used to modulate the signal sent from the RF signal generator. Typically, a BPM will not be picking up a signal from a continuous wave beam, as most beams are bunched. Essentially the pickup will see a high frequency signal, followed by a signal of nothing before seeing the same high frequency signal again. This high frequency signal is seen when a bunch goes passed and nothing is seen in the gaps between bunches. To simulate this, the RF signal from the signal generator is modulated with a square wave. To calculate the period of the square wave equation 3 is used.

$$Period = \frac{Number\ of\ Harmonics}{RF\ Frequency} \tag{3}$$

Diamond has an RF frequency of 499.681 768 2 MHz and 936 harmonics, yielding a pulse period of 1.873 19 µs. It's imperative that the gate source used for the tests can be set with a period as low as this. The gating device actually used in this project is the same *Rigol3030* used to output the RF signals. It is however, implemented in a separate class so that a two in one device isn't needed to perform these tests. Figure 12 shows the APIs that the abstract class *Generic_GateSource* have. Like all abstract classes all of these APIs must be overridden so that the device can be used.

Gating Device API List			
Gating Device API Name	Argument Types	Return Types	Units
get_device_ID	self	string	NA
turn_on_modulation	self	boolean	NA
turn_off_modulation	self	boolean	NA
get_modulation_state	self	boolean	NA
get_pulse_period	self	double, string	µs
set_pulse_period	self, double	double, string	µs
get_pulse_dutycycle	self	double	0-1
set_pulse_dutycycle	self, double	double	0-1

Figure 12: Abstract Gate Source Class APIs

As the device used is the same *Rigol3030* used before, the private *Telnet* APIs are copied and pasted over from the other *Rigol3030* class, then the *Generic_GateSource* APIs are overloaded with calls using various SCPI commands. The *Rigol3030* has a number of functions that would not necessarily be implemented on all gating devices, these include the ability to invert the waveform polarity. APIs have been developer for this, but as not all devices in this family are guaranteed to have this functionality, it is left out of the generic class, and is implemented in the *Rigol3030DSG_GateSource* class.
The gate source family also has a simulator, much like the RF source simulator, a number of variables keep a record of the virtual values of pulse period and duty cycle as well as state. These are changed using the *set* methods and then retrieved using the *get* methods. If a user of the framework wanted to add another gating device to the project a new class that inherits from *Generic_GateSource* would need to be implemented, and then all of the APIs listed in figure 12 overloaded.

3.4 Programmable Attenuators

Programmable attenuators are an easy way to change the position calculated by the BPM readout device. As stated in equations 1, and 2, the position calculated by the device is dependent on the relative signal sent to each of the four inputs, *A*, *B*, *C*, and *D*. In order to move the signal read by the device four RF signal generators could be used. However, given the cost and size of a single RF signal generator using four is not advised. Instead, a four channel programmable attenuator can be used. A signal is generated by the RF generator, equally divided using a splitter and then each output fed into a separate channel of the programmable attenuator. The attenuator outputs are then fed into the BPM. Figure 13 shows how this would be setup. The result is that by varying the levels of attenuation on each channel, signals of varying amplitude will be sent to each input and the position calculated by the device will change.

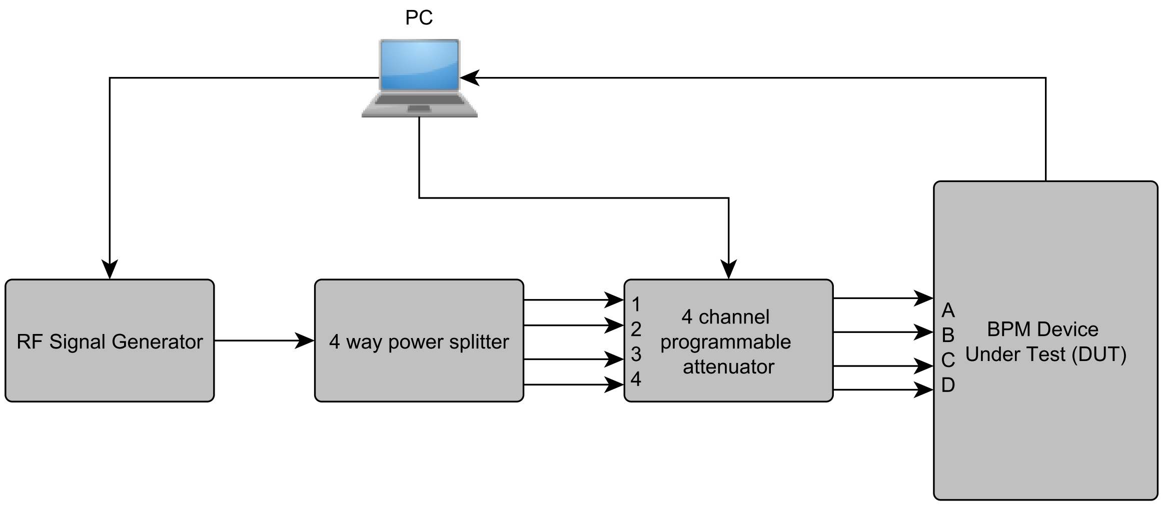

PC
RF Signal Generator
4 way power splitter
1
2
3
4
4 channel programmable attenuator
A
B
C
D
BPM Device Under Test (DUT)

Figure 13: Test Setup with Programmable Attenuator

while many programmable attenuators have advanced features that can sweep and hop from different levels, the APIs must be common amongst all device of this type and thus the APIs are fairly simple. A list of the APIs can be seen in figure 14. All of the values sent and received using these APIs will be in dB. To select a channel, a value of *A*, *B*, *C*, or *D* is entered into the channel argument, whereas a float is entered to set the attenuation value.

Programmable Attenuator API List			
Attenuator API Name	Argument Types	Return Types	Units
get_device_ID	self	string	NA
set_global_attenuation	self, double	double (x4)	dB (x4)
get_global_attenuation	self	double (x4)	dB (x4)
set_channel_attenuation	self, string, double	double	dB
get_channel_attenuation	self, string	double	dB

Figure 14: Abstract Programmable Attenuator Class APIs

As with all device families used for this project a simulator can be used to emulate using an attenuator in the project. The specific programmable attenuator used for this project is the *MiniCircuits RC4DAT6G95*. This allows for a resolution of 0.25 dB with attenuation values and can be communicated with using SCPI commands over *Telnet*. The private *Telnet* commands implemented in both the *Rigol3030* classes were copied into the *MC_RC4DAT6G95* class, however, the *MC_RC4DAT6G95* send termination characters at the start and the end of the replies it sends. As such the private *Telnet* commands were amended to take this into account. The *MC_RC4DAT6G95* can then be programmed using SCPI commands supplied with the device. These are then wrapped inside the APIs listed in figure 14. If a different programmable attenuator was to be used in this project instead, it would simply need to inherit from the *Generic_Prog_Atten* class, and overload all of the APIs listed in figure 14.

3.5 Beam Position Monitors

Of all of the classes used in this project the ones needed to be polymorphic and abstracted the most are the BPM classes. As this is a testing framework for BPMs they are the piece of hardware that will be changed the most. The API calls to the BPMs are seen in figure 15, for all of these functions the units reported back are always assumed to be in a standard format. For example, the *X* position returned by the *get_X_positon* method will always be measured in mm. If a device reports a position value in something that is not mm, such as the *Libera Spark ERXR*, which natively reports values in nm, the number should be coerced into mm so that all devices report the same units from the same API calls.

Beam Position Monitor API List			
Gating Device API Name	Argument Types	Return Types	Units
get_device_ID	self	string	NA
get_X_position	self	double	mm
get_Y_position	self	double	mm
get_beam_current	self	double	mA
get_input_power	self	double	dBm
get_raw_BPM_buttons	self	int (x4)	Counts (x4)
get_normailised_BPM_buttons	self	double (x4)	0-1 (x4)
get_ADC_sum	self	int	Counts
get_input_tolerance	self	double	dBm

Figure 15: Abstract BPM Class APIs

Currently at the time of writing there are three different BPMs setup to work with this framework. They are; the *Libera Spark ER*, *Libera Spark ERXR* and *Libera Electron*. The *Libera Spark ER* communicates using *Telnet* like much of the hardware using in this project, and the other two communicate using *EPICS*. The *Spark ER* uses a variation of the private methods found on the *Rigol3030* to communicate and send SCPI commands. The other devices that use *EPICS* have different private functions that wrap up the *EPICS* calls. Figure 16 shows the private *EPICS* calls that will read and write to *EPICS* process variables that are hosted on the BPM. Like the *Telnet* calls these functions are repeated in all devices that communicate using *EPICS*. While this results in duplicated code it also makes it easier to adapt.

```python
def _read_epics_pv(self, pv):
  # Read selected epics PV
  return caget(self.epicsID + pv)

def _write_epics_pv(self, pv, value):
  # Write to EPICs PV
  caput(self.epicsID+pv, value)
  # Return the PV that was just written to
  return self._read_epics_pv(pv)
```

Figure 16: Private Functions that communicate with *EPICS* Process Variables

The *Spark ERXR* will normally only update it's process variables when using the *camonitor* function, from figure 16 you can see that the private *_read_epics_pv* method uses the single call *caget* function. To get around this issue another private method, shown in figure 17 is used. The *.PROC* database is written to using the *_trigger_epics* method, this then updates all the values so then a *caget* call wrapped in the *_read_epics_pv* method can be used.

```python
def _trigger_epics(self):
  # Write to the .PROC data base to update all of the values
  caput(self.epicsID + ".PROC", 1)
```

Figure 17: Private Function to update *EPICS* database on the *Spark ERXR*

For both of the *Spark* devices there are a couple of APIs that have not been fully implemented. They are *get_beam_current* and *get_input_power*. The reason for this is because this data can be directly extracted from these devices. For the *Libera Electron* the beam current and input power data is a process variable that can simply be read. For the *Spark* devices this data cannot be obtained simply. It can be calculated, by converting the ADC sum into a voltage, and then, if the connection impedance is know (usually $50\,\Omega$) then the signals can be calculated. That would not accurately compare values served up from the BPM but instead ones that are derived from those values, as such this has been left out, so the *get_beam_current* and *get_input_power* methods will just report the ADC sum.
Of all of the simulators used, the BPM simulator is the most intelligent. This is because using the values provided by the other simulators it must calculate a realistic response that a BPM would provide. Also as some tests do not use all of the different hardware classes that BPM simulator must take this into account. For example, for the BPM simulator to report an accurate value for beam current, it must get a value from the simulated RF objects *get_output_power* method. This works sufficiently well, but some other tests may use the gate source as well, where the duty cycle set on the gate source will also impact the beam current value. This has been implemented by having conditional statements that take into account the gate source if one is present. Currently there is no work done on the simulator to take into account the programmable attenuator in the BPM simulator, though this could be implemented by making a programmable attenuator object an input argument in the BPM simulator. The problem with doing things this way with conditional statements is that it does not scale well. If there are a number of tests that all use varying levels of hardware, having a simulator that works for all of them may not but sensible, and making a BPM simulator for each type of test may be a better solution.
There are three different hardware BPMs and one BPM simulator that work with this test framework, these use more than one protocol to communicate with the hardware and the tests see the devices no differently. While the three hardware devices are all made by the manufacturer *Libera* there is no restriction that new devices have to be made by the same manufacturer. As long as the class that communicates with the device inherits from the *Generic_BPMDevice* class and overloads the methods correctly, a new device would slot into the test framework very easily. When overloading functions with the BPM classes it is very important to ensure that the APIs return the same data type in the same units. Otherwise the results from the test cannot be accurately compared.

4 Reporting

4.1 LaTeXReporting

This testing framework needed a mechanism to record the test results it creates. Python has a *PyLaTeX* library, this makes it easy to create LaTeXreports using python code. The method of getting Python to write LaTeXhas been implemented here by creating a class called *Tex_Report*. This class has specific methods suited to this framework, such as *setup_test* and *add_figure_to_test*. In essence, crating an instance of the *Tex_Report* class creates a Tex file, the different methods are then used to add information about the tests to that Tex file. When all of the tests have been run, the *create_report* method is used to compile the LaTeXcode and generate a PDF of the report.
In hindsight, it would have been better to have the same class hierarchy and abstraction levels that exist in the hardware classes, in the report class. That way, more than one type of LaTeXreporting class could be created and used in the framework, or even, a non LaTeXreport style could be used. For the *Tex_Report* class, the APIs constructed will do the job of recoding all of the test data and test parameters as well. If this was wanted to be changed or extended an abstract class with the same public APIs could be implemented.
There are three main APIs that fill the Tex file with details about each test in the *Tex_Report* class. They are:

- *setup_test*
- *add_figure_to_test*
- *add_table_to_test*

The purpose of the *setup_test* method is to start a new section in the report. This section will then have in it the parameters of the test, the devices used in the test, and of course an introduction of the test itself, they are entered into the report using the input arguments found seen in figure 18. To effectively use this class and method the correct inputs must be put into the arguments, for the *section_title*, it's recommended that this is used as the name of the test. The *introduction* argument is used to describe what the test is. This is a string copied to a variable at the start of any test, it will describe what the test does and what information it should give, it should be a comprehensive description. The arguments *device_names* and *parameters* will both be a list, or array, of stings. These are used to fill the report with what devices were used in the test, and what parameters they were tested with.

```python
def setup_test(self, section_title, introduction, device_names, parameters):
```

Figure 18: Input Arguments for *setup_test* of the *Tex_Report* class

The method *add_figure_to_test* is used to add any image to the test, but typically a graph for these tests. The graphs should be saved to a file, normally as a PDF. By feeding the *image_name* argument shown in figure 19 with the same file name and path that a graph is saved to, it will insert it into the report. If desired a caption for the graph can be inserted into the figure as well.

```python
def add_figure_to_test(self, image_name, caption=""):
```

Figure 19: Input Arguments for *add_figure_to_test* of the *Tex_Report* class

The *add_table_to_test* method will place a table into the test report showing values. As each test can produce a large number of samples, the *add_table_to_test* method limits table sizes to twenty rows of data. The input arguments on this method are used to configure what kind of table it will create, all tables are wrapped into a figure in the report that they create. The *format_string* argument is used to select what type of table it will be in LaTeXform, for example *|c|c|c|* is used for a three cell table with lines on each side of a cell. The data is in the form of a 2D array, each row contains one sample from each column and the *headings* is a number of strings used to give headings to each column, and like the *add_figure_to_test* method a caption can be used if desired. Using these arguments correctly will insert a well formatted table into the report to provide supporting data to the graphs.

```python
def add_table_to_test(self, format_string, data, headings, caption=""):
```

Figure 20: Input Arguments for *add_table_to_test* of the *Tex_Report* class

All of these methods are called by each test. Section 5 shows how these methods are used to place information from the tests into the report, before the Tex file is complied by the launcher script.

5 BPM Test Scripts

5.1 Test Script Template

Each test has a similar setup, they all output signals, wait a specified amount of time, then take a reading from the BPM. All of these results are then entered into the same report. As such, a template has been developed as a starting point for any new tests. Figure 21 shows this template. test test input arguments include a *BPMObject* and *RFObject*. These are used to speak with the hardware used in the test, so of course if the test that's being designed requires more hardware the corresponding hardware objects will appear here as well.
For the *ReportObject* to correctly enter data into the report it needs to know what it's entering. This is why the *test_name* is extracted, and why the lists for *device_names* and *parameter_names* are complied. The test is then performed between the # lines, for the template, the values are just simulated, but in a real test this is where the output signal will be changed in someway, before waiting, and then taking a reading. After the *for loop* ends the results should all be appended to a number of lists or arrays. If a *ReportObject* was entered as an input argument, then the data in the form of a graph will be saved to a PDF file and then copied into the report. If there was no report entered, then the plot will simply be displayed.

```python
def Template(RFObject,
             BPMObject,
             argument1=1,
             argument2=2,
             argument3=3,
             settling_time=1,
             ReportObject=None):

    # Readies text that will introduce this test in the report
    intro = r"This is a template test"

    # Formats the name of plot that is saved as
    test_name = __name__
    test_name = test_name.rsplit("Tests.")[1]
    test_name = test_name.replace("_", " ")
    print("Starting test \"" + test_name + "\"")

    # Readies devices that are used in the test for the report
    device_names = []
    device_names.append(RFObject.get_device_ID())
    device_names.append(BPMObject.get_device_ID())

    # Readies parameters that are used in the test for the report
    parameter_names = []
    parameter_names.append("Argument1: " + str(argument1))
    parameter_names.append("Argument2: " + str(argument2))
    parameter_names.append("Argument3: " + str(argument3))

    #######Change these lines with the test you wish to perform#######
    x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    y = []
    for index in x:
        time.sleep(settling_time)
        y.append(2*index)
    ##################################################################

    # plots the test results
    plt.plot(x,y)

    if report == None:
        # If no report is entered as an input to the test, display results
        plt.show()
    else:
        # If there is a report for the data to be copied to, do so.
        plt.savefig(test_name + ".pdf")
        ReportObject.setup_test(test_name, intro, device_names, parameter_names)
        ReportObject.add_figure_to_test(test_name)
```

Figure 21: Template for new Test Scripts

5.2 Beam Power Dependence Test

The *Beam Power Dependence Test* uses the simple setup shown in figure 2. The purpose of the test, is to output a signal from an RF signal generator, equally split the signal into four, and then deliver the equal signals into all four BPM inputs. The signal output from the RF source is then linearly increased in dBm, and the position, power, current, and ADC values, from the BPM are then all recorded with each change in power.
As the signals should all be equal, equations 1, and 2 predict that the position should always read zero. This is not actually the case as there is a noise floor that the signals have to overcome so they can be read by the BPM. The test itself has a number of input arguments that configure how the test is run. Figure 22, shows the arguments used to configure this test.
The objects used to communicate with the hardware are arguments *RFObject* and *BPMObject*, these will use the APIs listed in section 3 to output the RF signals and read values from the BPM. The reporting features of the test similar to the ones described in section 5.1, this uses the *ReportObject* argument to configure a file and save all of the graphs and information into it.
The real test parameters are the non object arguments. They configure the instruments and how the test actually operates. The *frequency* argument is there to set the output frequency of the RF signal generator, this should match the RF frequency of the accelerator the devices are used at, so for Diamond this will be 499.681 768 2 MHz.
The *start_power* and *end_power* arguments are set in dBm. They are respectively, the first and last power values output by the RF signal generator. The *samples* argument states how many steps and thus measurements will go between these two values.
After the RF signal generator outputs a signal, the test waits a number of seconds specified by *settling_time* then takes a reading from the BPM. This is to allows the entire system to settle to a steady state before taking a reading off of the BPM.

```
def Beam_Power_Dependence(
    RFObject,  # Object for communicating with the RF hardware
    BPMObject,  # Object for communicating with the BPM hardware
    frequency,  # Frequency in MHz to set the RF to for the test
    start_power=-100,  # Starting output power for the test in dBm
    end_power=0,  # Final output power for the test in dBm
    samples=10,  # Samples to take from start, to end, power values
    settling_time=1,  # Seconds to wait between changing output and reading input
    ReportObject=None,  # Object for writing to a report file
    sub_directory=""):  # Subdirectory that results should be saved to
```

Figure 22: Input Arguments for *Beam_power_dependence_test*

Figure 23 shows the part of the test that actually takes the measurements. All of the initial values are created or set in the first five lines, this includes setting the frequency and initial power level of the RF source as well as enabling it's output.
The next part of the test is initializing the arrays that the results will be copied to, these arrays are returned at from the function, should post analysis or data recording be required. finally the *for loop* is there to set the RF values, wait for the values to settle, and then take the reading from the BPM. This is repeated for all of the RF values set by the input arguments.

```python
# Set the initial state of the RF device
power = np.linspace(start_power, end_power, samples)  # Creates power values
RFObject.set_frequency(frequency)  # set RF frequency
RFObject.set_output_power(start_power)  # start staring power value
RFObject.turn_on_RF()  # turn the RF output on
time.sleep(settling_time)  # wait for values to settle

# Build up the arrays where the final values will be saved
X_pos = np.array([])  # Init array for X position measurements
Y_pos = np.array([])  # Init array for Y position measurements
beam_current = np.array([])  # Init array for BPM current measurements
output_power = np.array([])  # Init array for RF power measurements
input_power = np.array([])  # Init array for BPM power measurements
ADC_sum = np.array([])  # Init array for ADC measurements

# Perform the test
for index in power:
    # Set next output power value
    RFObject.set_output_power(index)
    # Wait for signal to settle
    time.sleep(settling_time)
    # record beam current
    beam_current = np.append(beam_current, BPMObject.get_beam_current())
    # record X position read from the BPM
    X_pos = np.append(X_pos, BPMObject.get_X_position())
    # record Y position read from the BPM
    Y_pos = np.append(Y_pos, BPMObject.get_Y_position())
    # record RF power output from the RF generator
    output_power = np.append(output_power, RFObject.get_output_power()[0])
    # record RF power input into the BPM
    input_power = np.append(input_power, BPMObject.get_input_power())
    # record the sum of all four ADC channels
    ADC_sum = np.append(ADC_sum, BPMObject.get_ADC_sum())
```

Figure 23: Measurement loop for *Beam_power_dependence_test*

5.3 Fixed Voltage Amplitude Fill Pattern Test

The *Fixed_voltage_amplitude_fill_pattern_test* is a test that will check to see how the BPM will cope with a bunched beam. Essentially, as an accelerator ring is filled up, there are bunches of particles going around the ring, and there are spaces between those particles. To emulate this behavior in the lab the RF signal is modulated with a gating signal that has the same period as the bunch length in the accelerator. Equation 3 shows that the pulse period of Diamond is 1.873 19 μs. This test will output an fixed RF frequency and voltage amplitude, and then modulate the RF signal with the square wave that has a fixed period. The duty cycle of the square wave is then linearly increased from 0.1 (meaning 10%) to 1 (meaning 100%). After the duty cycle is changed the system will wait a short time to allow the signals to settle, and then take a range of readings off of the BPM.

```python
def Fixed_voltage_amplitude_fill_pattern_test(
    RFObject,  # Object for communicating with the RF hardware
    BPMObject,  # Object for communicating with the BPM hardware
    GateSourceObject,  # Object for communicating with the Gate source hardware
    frequency,  # Frequency in MHz to set the RF to for the test
    power=0,  # Output power for the test in dBm
    samples=10,  # Samples to take from 0.1 to 1 on the duty cycle
    pulse_period=1.87319,  # period of the modulation square wave
    settling_time=1,  # Seconds to wait between changing output and reading input
    ReportObject=None,  # Object for writing to a report file
    sub_directory=""):  # Subdirectory that results should be saved to
```

Figure 24: Input Arguments for *Fixed_voltage_amplitude_fill_pattern_test*

Figure 24 shows the arguments used for the test. The Objects are used to communicate with the relevant hardware or report used in the test, and the other arguments are used to set the parameters of the test. Figure 25 shows how the measurements are actually taken. The first part of the script sown in figure 25 shows the initial values being set on the hardware, then the arrays being initialized, and finally a *for loop* that will tell the hardware to output signals and take readings from the BPM. Note that as the duty cycle changes, the RF signal does not. This means that as the duty cycle is reduced, the overall power of the RF signal going to the BPM is reduced.

```python
RFObject.set_frequency(frequency)
RFObject.set_output_power(power)

cycle = np.linspace(0.1, 1, samples)  # Creates samples to test
GateSourceObject.set_pulse_period(pulse_period)  # Set pulse period
GateSourceObject.set_pulse_dutycycle(cycle[0])  # set initial duty cycle
RFObject.turn_on_RF()  # turn on the RF output
GateSourceObject.turn_on_modulation()  # Modulate the RF output with pulse
time.sleep(settling_time)  # wait for signals to settle

dutycycle = np.array([])  # Init array for duty cycle measurements
bpm_power = np.array([])  # Init array for BPM power measurements
bpm_Xpos = np.array([])  # Init array for BPM X position measurements
bpm_Ypos = np.array([])  # Init array for BPM Y position measurements
bpm_current = np.array([])  # Init array for BPM current measurements
ADC_sum = np.array([])  # Init array for duty ADC sum measurements

for index in cycle:
    # set duty cycle output
    dutycycle = np.append(dutycycle, GateSourceObject.set_pulse_dutycycle(index))
    # wait for signals to settle
    time.sleep(settling_time)
    # take BPM power measurement
    bpm_power = np.append(bpm_power, BPMObject.get_input_power())
    # take BPM current measurement
    bpm_current = np.append(bpm_current, BPMObject.get_beam_current())
    # take BPM X position measurement
    bpm_Xpos = np.append(bpm_Xpos, BPMObject.get_X_position())
    # take BPM Y position measurement
    bpm_Ypos = np.append(bpm_Ypos, BPMObject.get_Y_position())
    # take sum of all ADC channel measurement
    ADC_sum = np.append(ADC_sum, BPMObject.get_ADC_sum())
```

Figure 25: Test Procedure for *Fixed_voltage_amplitude_fill_pattern_test*

5.4 Scaled Voltage Amplitude Fill Pattern Dependence Test

The *Scaled_voltage_amplitude_fill_pattern_test*, is very similar in function to the *Fixed_voltage_amplitude_fill_pattern_test*. It is the same, in that the RF signal is modulated with a pulse signal and that it is trying to see what the dependence is on bunch structure for the BPM. The difference with this test though, is that when the duty cycle is changed the overall RF power going into the BPM is kept the same. See how in figure 26 the argument for *power*, has been replaced with *desired_power* as it now takes into account the duty cycle.

```python
def Scaled_voltage_amplitude_fill_pattern_test(
    RFObject,  # Object for communicating with the RF hardware
    BPMObject,  # Object for communicating with the BPM hardware
    GateSourceObject,  # Object for communicating with the Gate source hardware
    frequency,  # Frequency in MHz to set the RF to for the test
    desired_power=0,  # Output power for the test in dBm
    samples=10,  # Samples to take from 0.1 to 1 on the duty cycle
    pulse_period=1.87319,  # period of the modulation square wave
    settling_time=1,  # Seconds to wait between changing output and reading input
    ReportObject=None,  # Object for writing to a report file
    sub_directory=""):  # Subdirectory that results should be saved to
```

Figure 26: Input Arguments for *Scaled_voltage_amplitude_fill_pattern_test*

Figure 25 shows the steps taken to perform the *Scaled_voltage_amplitude_fill_pattern_test*. As well as recoding all of the values from the BPM this will also record the RF output power before it is modulated with the pulse signal. As such, as the duty cycle of the pulse signal goes down, the RF output power will have to increase to compensate. This is done by calculating the loss in dB that the duty cycle change will cause, and then increasing the RF output power by this much.

```python
RFObject.set_frequency(frequency)
RFObject.set_output_power(power)
cycle = np.linspace(0.1, 1, samples)  # Creates samples to test
GateSourceObject.set_pulse_period(pulse_period)  # Set pulse period
GateSourceObject.set_pulse_dutycycle(cycle[0])  # set initial duty cycle
RFObject.turn_on_RF()  # turn on the RF output
GateSourceObject.turn_on_modulation()  # Modulate the RF output with pulse
time.sleep(settling_time)  # wait for signals to settle

dutycycle = np.array([])  # Init array for duty cycle measurements
bpm_power = np.array([])  # Init array for BPM power measurements
bpm_Xpos = np.array([])  # Init array for BPM X position measurements
bpm_Ypos = np.array([])  # Init array for BPM Y position measurements
bpm_current = np.array([])  # Init array for BPM current measurements
rf_output = np.array([])  # Init array for power output set at the RF
ADC_sum = np.array([])  # Init array for duty ADC sum measurements

for index in cycle:
    # set the gate source duty cycle
    current_cycle = GateSourceObject.set_pulse_dutycycle(index)
    # take a reading of the duty cycle
    dutycycle = np.append(dutycycle, current_cycle)
    # calculate the change in duty in loss of dB
    log_cycle = 20*np.log10(current_cycle)
    # increase power in the RF to compensate for duty cycle change
    RFObject.set_output_power(desired_power + np.absolute(log_cycle))
    # wait for signals to settle
    time.sleep(settling_time)
    # take RF output power measurement
    rf_output = np.append(rf_output, RFObject.get_output_power()[0])
    # take BPM power measurement
    bpm_power = np.append(bpm_power, BPMObject.get_input_power())
    # take BPM current measurement
    bpm_current = np.append(bpm_current, BPMObject.get_beam_current())
    # take BPM X position measurement
    bpm_Xpos = np.append(bpm_Xpos, BPMObject.get_X_position())
    # take BPM Y position measurement
    bpm_Ypos = np.append(bpm_Ypos, BPMObject.get_Y_position())
    # take sum of all ADC channel measurement
    ADC_sum = np.append(ADC_sum, BPMObject.get_ADC_sum())
```

Figure 27: Test Procedure for *Scaled_voltage_amplitude_fill_pattern_test*

5.5 Beam Position Equidistant Grid Raster Scan Test

The *Beam_position_equidistant_grid_raster_scan_test* uses the programmable attenuator to take position readings from the BPM. The test creates a group of A, B, C, D, input values that would move a Diamond BPM from −5 mm to 5 mm in both the X and Y planes. The input values used to move the BPM position into these values are converted from ratios into attenuation values to set for each of the four attenuators.
Figure 28 is used to setup the set itself. The number of samples taken will be *x_point* multiplied by *y_points*, for all of these samples the frequency and power output from the RF signal generator will remain constant. Each attenuator is given a starting value of attenuation, this is so there is room to increase the power of the signals as well as reducing them, this starting value is set with the *nominal_attenuation* argument.
As with all of the other tests the objects are used to communicate with the hardware or the report, and the *settling_time* is the time to wait between setting a new value (this time on the attenuator), and taking a reading from the BPM.

```
def Beam_position_equidistant_grid_raster_scan_test(
    RFObject,  # Object for communicating with the RF hardware
    BPMObject,  # Object for communicating with the BPM hardware
    ProgAttenObject,  # Object for communicating with the programmable attenuator hardware
    rf_power,  # Output power for the test in dBm
    rf_frequency,  # Frequency in MHz to set the RF to for the test
    nominal_attenuation,  # Starting attenuation values for all attenuators
    x_points,  # Number of samples to take in the x plane
    y_points,  # Number of samples to take in the y plane
    settling_time=1,  # Seconds to wait between changing output and reading input
    ReportObject=None,  # Object for writing to a report file
    sub_directory=""):  # Subdirectory that results should be saved to
```

Figure 28: Input Arguments for *Beam_position_equidistant_grid_raster_scan_test*

Both equations 1 and 2 are implemented in python to predict where the position of the beam will be. The predicted where the position should be, as the ADCs are not perfectly linear there is a slight discrepancy between the measured and predicted values at the limits measured from the BPM, this is referred to as a *pin cushion* effect. To accurately see this both the predicted and measured values of X and Y are recorded onto the graphs. Figure 29 shows that these values are all returned after the test has been run. As the test is slightly more involved that the others, to see it's functionality please look at the source code to read the docstrings and comments.

```
    return measured_x, measured_y, predicted_x, predicted_y
```

Figure 29: Return values for *Beam_position_equidistant_grid_raster_scan_test*

5.6 Beam Position Attenuation Permutation Test

The *Beam_position_attenuation_permutation_test* is a less intelligent test than the *Beam_position_equidistant_grid_raster_scan_test*. Instead of assuming attenuation values based on position values, a permutation of different attenuation values are created, these are then fed into the attenuator. After each configuration is loaded into the attenuator, the test will wait a small amount of time specified by *settling_time* before taking the position readings from the BPM.

```
def Beam_position_attenuation_permutation_test(
    RFObject,  # Object for communicating with the RF hardware
    BPMObject,  # Object for communicating with the BPM hardware
    ProgAttenObject,  # Object for communicating with the programmable attenuator hardware
    rf_power,  # Output power for the test in dBm
    rf_frequency,  # Frequency in MHz to set the RF to for the test
    attenuator_max,  # Maximum attenuator value
    attenuator_min,  # Minimum attenuator value
    attenuator_steps,  # Steps between min and max attenuator values
    settling_time=1,  # Seconds to wait between changing output and reading input
    ReportObject=None,  # Object for writing to a report file
    sub_directory=""):  # Subdirectory that results should be saved to
```

Figure 30: Input Arguments for *Beam_position_equidistant_grid_raster_scan_test*

While the attenuator values are changing, the RF output frequency and power are kept the same at the RF source, these, values are set wit the *rf_frequency*, and *rf_power* arguments respectively. Figure 30 shows the arguments used to launch the test, as expected all of the object arguments are used to speak with the hardware and write to the report.
The arguments used to configure the attenuator are *attenuator_max*, *attenuator_min* and *attenuator_steps*, *attenuator_steps* creates a linear space between the min and max values in the number of steps provided. This line space between and including the min and max value is used to create all of the different permutations of these values if there were four outputs, as such just having the two values (the min and max only) will create sixteen different permutations. The attenuation assigned to each input is used to calculate the predicted amount of power going into each input, this is then used in to predict what the position value should be given those attenuations, the predicted values are then plotted against the measured values, and both sets are returned from the test.

6 Operation Guide

6.1 Setup

This guide provides details on how to use the software framework developed for testing BPM readout electronics. It is a step by step guide, giving instructions on how to connect hardware, how to launch a test, as well as how to add new tests and hardware to the framework. Figure 31 shows how to setup a simple test that only provides an equal RF signal into each BPM input. As this project uses a *Rigol3030* this is also the setup when performing bunch shape tests, as the *Rigol3030* has the ability to modulate it's own output signals. If these where two different devices that would need to be taken into account.

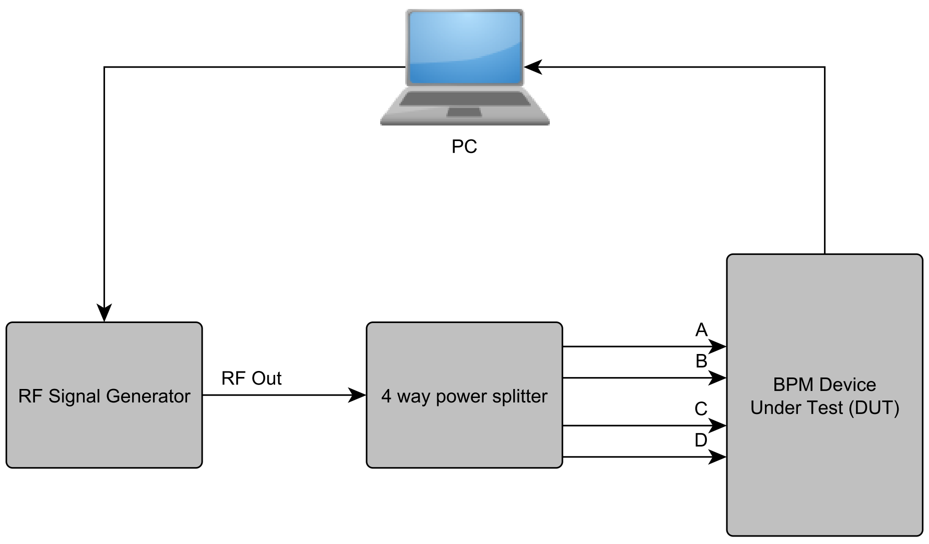


Figure 31: Setup for BPM tests using a *Rigol3030*

Figure 32 shows how to connect the equipment if performing a test that uses the programmable attenuator. Technically this setup could be used for tests that don't need the programmable attenuator, but, it is not believed that the attenuator can truly be set at 0 dB attenuation, meaning it will have some impact on the test results. Also, the tests that do not require the programmable attenuator are not aware of it as a device, as such, they cannot communicate to it to ensure that it has a constant attenuation value or report it's presence in their section of the report. The recommendation is therefore that this should be disconnected when not performing tests that use it.

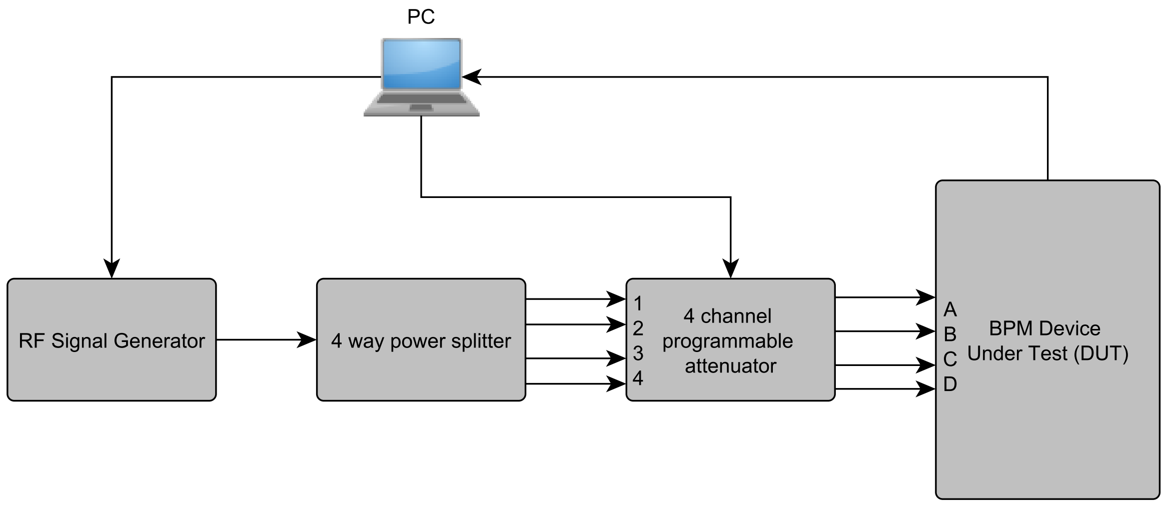

PC
RF Signal Generator
4 way power splitter
1
2
3
4
4 channel programmable attenuator
A
B
C
D
BPM Device Under Test (DUT)

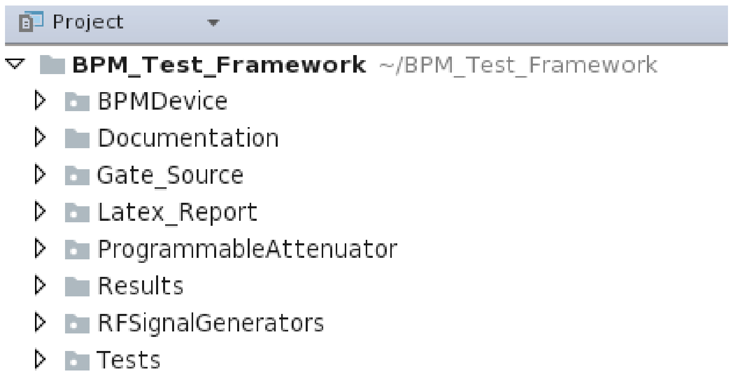
Figure 32: Setup for BPM tests using *Rigol3030* and Programmable Attenuator

6.2 Launching a Test

To successfully launch a test the correct python interpreter must be used. This project has been developed using the in house Diamond interpreter for python 2.73, with *cothread* version 2.14, and as such it's function can only be guaranteed using these versions.
To launch a test a separate script is written, this is referred to as a *Launcher* script, and it has four main components:

- Library Imports
- Object Instantiation
- Initializing Test Arguments
- Launching a Test

The first of these four components, *Library Imports*, is used so that the script has access to the necessary classes and scripts it will need to perform a test. Each class family and test imports the libraries they require inside their own files. For example, a hardware device that communicates using *Telnet*, will import the *telnetlib* library. As such when using a device that happens to communicate using the same protocol, this does not need to be imported in the launcher script. The project folder structure is shown in figure 33, depending on what tests are wanted to be called will change what libraries need to be imported. Each folder that contains python files has an __*init*__ file that will import the specific classes and functions from the scripts. The result of this is that then, only the folder needs to be imported.



```
Project
▽ BPM_Test_Framework ~/BPM_Test_Framework
    ▷ BPMDevice
    ▷ Documentation
    ▷ Gate_Source
    ▷ Latex_Report
    ▷ ProgrammableAttenuator
    ▷ Results
    ▷ RFSignalGenerators
    ▷ Tests
```

Figure 33: BPM Test Framework Folder Structure

To create a *Launcher* script, create a new python file inside the top level of the *BPM_Test_Framework* folder. The file can be called anything, but it's recommended to be something like *Launcher*. Figure 34 shows the lines of code required to import all of the hardware libraries, report libraries, and the test scripts themselves. Once these have been placed at the start of the file. It means that the classes and files contained in these libraries can then be called.

```
import RFSignalGenerators
import BPMDevice
import Gate_Source
import ProgrammableAttenuator
import Tests
import Latex_Report
```

Figure 34: Libraries imported into a launcher file

Once the desired libraries have been imported it's necessary to instantiate the objects for classes used in the test. For example, assume that the *Beam_position_equidistant_grid_raster_scan_test* will be run, the test signals will be applied using the *Rigol3030DSG_RFSigGen* class, and the *MC_RC4DAT6G95_Prog_Atten* class will be used to attenuate the signals going to the *SparkERXR_EPICS_BPMDevice*. All the test data will then be recorded recorded in a LATEXreport using the *Tex_Report* class.
Figure 35 shows these objects instantiated. Each object needs its own input arguments to successfully setup the device. For objects like the report all it needs is a file name that it will save the report as, for the devices that communicate over *Telnet*, an IP address and port number is needed to successfully communicate with the device. Comparatively, *EPICS* devices do not need an IP address, but they do use a unique identifier in their database name. For the *Libera Electron* devices setup by Diamond this is the form of a number that is concatenated to the database name. As the *Spark ERXR* has not been setup for use at Diamond yet, the default *EPICS* database name is used, *libera*. The *Spark ERXR* has several types of acquisition modes, they are, *slow acquisition*, *fast acquisition*, and *turn by turn*. This can be changed by setting the *daq_type* argument to *sa*, *fa*, or *tbt*, respectively.

```
report = Latex_Report.Tex_Report(
    fname="BPM test Report")  # filename the output report will have

BPM = BPMDevice.SparkERXR_EPICS_BPMDevice(
    database="libera",  # libera is the prefix used in the EPICS database
    daq_type="fa")  # set as "fa" for fast acquisition

RF = RFSignalGenerators.Rigol3030DSG_RFSigGen(
    ipaddress="172.23.252.51",  # IP address of the Rigol3030
    port=5555,  # Telnet port used to communicate with the Rigol3030
    timeout=1,  # Timeout in seconds to wait for a reply
    limit=BPM.get_input_tolerance())  # Sets the max power limit of the RF

PA = ProgrammableAttenuator.MC_RC4DAT6G95_Prog_Atten(
    ipaddress="172.23.244.105",  # IP address of the programmable attenuator
    port=23,  # Telnet port used to communicate with the attenuator
    timeout=1)  # Timeout in seconds to wait for a reply
```

Figure 35: Instantiation of Objects used in the Test

Several tests may use the same input argument, for these arguments, it's sensible to initialize them as variables before entering them into the test, as they may be reused. For example, the Diamond storage ring RF frequency of 4.996 817 682 MHz will be the same in several tests, so it makes sense to initialize a variable with this value. As well as this, many of the tests will record a number of different graphs that are saved to a file, these are then inserted into the report when it's compiled. As many of these graphs can be created in a single run, it makes sense to save these into a subdirectory, figure 33 shows a folder called *Results*, the tests have an input argument to select a subdirectory where the results will be saved to, so this also makes sense to be initialized before running the tests. Figure 36 shows these values, the folder subdirectory must be set with the / notation used when saving files, so that *./Results/* will save the files to the *Results* subdirectory.

```
dls_RF_frequency = 499.6817682  # RF frequency of Diamond in MHz
sub_directory = "./Results/"  # Subdirectory the results are saved to
```

Figure 36: Values that may be used across multiple tests

The final part of a launcher script is to launch the test function. As the tests are found in the *Tests* folder the exact test to call will be taken from that library. For this example, the *Beam_position_equidistant_grid_raster_scan_test* is needed. The names of the arguments should be explicitly stated when calling a test, given this it may be obvious what each parameter is, but if more information is required, the tests docstring will contain this. Figure 37 shows the lines of code that will actually perform the test.

```
Tests.Beam_position_equidistant_grid_raster_scan_test(
    RFObject=RF,  # Uses the RF object instantiated to output RF signals
    BPMObject=BPM,  # Performs test on the BPM object instantiated
    ProgAttenObject=PA,  # Changes attenuation values on the attenuator instantiated
    rf_power=-40,  # Sets a constant RF output power of -40 dBm
    rf_frequency=dls_RF_frequency,  # Sets the RF output frequency
    nominal_attenuation=10,  # Given each attenuator a starting value of 10 dB
    x_points=5,  # Number of points to take in the x plane
    y_points=5,  # Number of points to take in the y plane
    settling_time=3,  # time to wait between changing value and taking a reading
    ReportObject=report,  # The report class that the results are saved to
    sub_directory=sub_directory)  # The directory that output graphs are saved to
```

Figure 37: Lines of code that launch a test with corresponding input parameters

Finally, after running all of the desired tests, the *report* class must use it's *create_report* method to compile the LaTeXreport and build a PDF. This can be seen at the bottom of the entire script on line 42, shown in figure 38.

```
 1 import RFSignalGenerators
 2 import BPMDevice
 3 import Gate_Source
 4 import ProgrammableAttenuator
 5 import Tests
 6 import Latex_Report
 7
 8 report = Latex_Report.Tex_Report(
 9     fname="BPM test Report")  # filename the output report will have
10
11 BPM = BPMDevice.SparkERXR_EPICS_BPMDevice(
12     database="libera",  # libera is the prefix used in the EPICS database
13     daq_type="fa")  # set as "fa" for fast acquisition
14
15 RF = RFSignalGenerators.Rigol3030DSG_RFSigGen(
16     ipaddress="172.23.252.51",  # IP address of the Rigol3030
17     port=5555,  # Telnet port used to communicate with the Rigol3030
18     timeout=1,  # Timeout in seconds to wait for a reply
19     limit=BPM.get_input_tolerance())  # Sets the max power limit of the RF
20
21 PA = ProgrammableAttenuator.MC_RC4DAT6G95_Prog_Atten(
22     ipaddress="172.23.244.105",  # IP address of the programmable attenuator
23     port=23,  # Telnet port used to communicate with the attenuator
24     timeout=1)  # Timeout in seconds to wait for a reply
25
26 dls_RF_frequency = 499.6817682  # RF frequency of Diamond in MHz
27 sub_directory = "./Results/"  # Sub directory the results are saved to
28
29 Tests.Beam_position_equidistant_grid_raster_scan_test(
30     RFObject=RF,  # Uses the RF object instantiated to output RF signals
31     BPMObject=BPM,  # Performs test on the BPM object instantiated
32     ProgAttenObject=PA,  # Changes attenuation values on the attenuator instantiated
33     rf_power=-40,  # Sets a constant RF output power of -40 dBm
34     rf_frequency=dls_RF_frequency,  # Sets the RF output frequency
35     nominal_attenuation=10,  # Given each attenuator a starting value of 10 dB
36     x_points=5,  # Number of points to take in the x plane
37     y_points=5,  # Number of points to take in the y plane
38     settling_time=3,  # time to wait between changing value and taking a reading
39     ReportObject=report,  # The report class that the results are saved to
40     sub_directory=sub_directory)  # The directory that output graphs are saved to
41
42 report.create_report()
```

Figure 38: Launcher script to perform Linear Raster Scan Test on *Spark ERXR*

A variation of this script can now be run to a perform any one of the tests on any of the hardware found in the framework. If this same test was to run, but on a different BPM, only lines 11-13 in figure 38 would need to be changed, to instantiate a BPM object of a sibling class also called

BPM, figure 39 shows lines of code that could be used. Instead of a *Spark ERXR* the *Electron_BPMDevice* class corresponds to the *Libera Electron*. By simply instantiating this object instead of the BPM object shown in figure 38 the test will be run on the different hardware device. Obviously, this would also require the *Spark ERXR* under test to be disconnected from RF signal generator, and the *Libera Electron* to be connected instead.

```
BPM = BPM = BPMDevice.Electron_BPMDevice(
        dev_ID="TS-DI-EBPM-04:")  # The ID number assigned to that specific BPM device
```

Figure 39: Instantiation of a different BPM object

If a different test was to be run instead of the one shown in figure 38. The lines required to change would be the ones found between lines 28 and 41. The exact name, arguments, and functionality of each test can be found in section 5, or in the docstrings of the test itself. Figure 40 shows the lines of code required to run an alternative test. This test only uses the an RF source and a BPM, where the output power is linearly ramped in dBm from *start_power* to *end_power*.

```
Tests.Beam_Power_Dependence(
    RF=RF,  # Uses the RF object instantiated to output RF signals
    BPM=BPM,  # Performs test on the BPM object instantiated
    frequency=dls_RF_frequency,  # Sets the RF output frequency
    start_power=-100,  # Sets the starting output power in dBm
    end_power=-20,  # Sets the final output power in dBm
    samples=10,  # Sets the number of samples to take between the two powers
    settling_time=1,  # Time to wait between changing value and taking a reading
    report=report,  # The report class that the results are saved to
    sub_directory=sub_directory)  # The directory that output graphs are saved to
```

Figure 40: Lines of code that launch an alternative test with corresponding input parameters

Multiple tests can be run in a single script, one after another, though the user must take into account that some tests will have different setups, as stated in section 6.1. If the tests do have a similar setup though, they can be launched one after another, and all of the results compiled into a separate report. If the setup is different a user prompt or wait could be inserted in between tests allowing the user to change the hardware setup.

6.3 Adding New Hardware to be used in a Test

One of the main objectives for this framework was that it should be scalable and extensible. As such, the ability to add new hardware to use for in testing is necessary. If the objective is to add new device to an existing family of classes, section 3 details the various APIs that each type of device class needs to override. The process of overriding these methods for a BPM is described here. The same process can be done for all of the existing software classes, by changing the overridden methods, and the class inheritance.
With respect to creating a new BPM class that will work with the tests there are a few steps to take note of:

- Class Inheritance
- Communication Protocol
- Library Imports
- Method Overriding
- Documentation

All of the BPMs should inherit from the *Generic_BPMDevice* class, figure 41 shows a class that does this. The class name here is *NEW_EPICS_BPMDevice*, typically this name should be more hardware specific, like how the class that communicates with the *SparkER* is called *SparkER_SCPI_BPMDevice*, but as this is an example *NEW_EPICS_BPMDevice*, will be used.

```
class NEW_EPICS_BPMDevice(Generic_BPMDevice):
```

Figure 41: BPM Device class that inherits from *Generic_BPMDevice*

All BPMs will communicate using a protocol, the two protocols that have already been used with framework are *Telnet* and *EPICS*. As it's probably more likely that new devices will use *EPICS* that is what is listed in this guide, though, the steps are somewhat transferable to other protocols as well. The *cothread* library used at Diamond contains the calls to and from *EPICS* process variables, typically these are *caget* and *caput*. These will read and write to process variables that are hosted on the device as long as they are given the appropriate variable name. To simplify things, these could be wrapped into a private class method, private methods are methods with a _ at the front meaning they are only supposed to be used internally within the class. This can be seen in the *_read_epics_pv* method, shown in figure 42.
Setting up private methods like the ones seen in figure 42 is optional to getting a new class to work with the framework, but it's good practice, and all of the current classes in the framework do this. The benefit of using private methods is that the *caget* call is now wrapped in a function that could include more functionality. As all of the public methods that will be overridden, probably require a call to an *EPICS* PV, this will save time if extra functionality was to be implemented, such as repeated attempts to read a PV if it fails on the first read. The calculation of getting the mean value that is currently done in the *get_X_position* method, the man calculation could also be moved to the *_read_epics_pv* method if this was wanted on all process variables that are read.

```python
# Name of the class and what it inherits from
class NEW_EPICS_BPMDevice(Generic_BPMDevice):

  # Arguments needed when instantiating the class
  def __init__(self, database):
    # Copies the database name to the internal class data
    self.epicsID = database

  # Private method to read an EPICS PV
  def _read_epics_pv(self, pv):
    # Read selected epics PV
    return caget(self.epicsID + pv)

  # Public method to get the BPMs X position
  def get_X_position(self):
    # Get the X position (in um) from the PV in the database hosted on the BPM
    x = self._read_epics_pv(".X")
    # Gets the mean value for X if it returns more than one sample
    x = np.mean(x)
    # Convert from um to mm
    x = x / 1000.0
    # Returns the mean X position calculated by the BPM
    return x
```

Figure 42: New BPM Device Class with public and private APIs

Figure 42 shows a private method that communicates with the hardware, and a public method that is used in the tests. For a device to successfully work with the framework, all public APIs found in the *Generic_BPMDevice* class must also appear in the child class that's being added. These methods can be seen in figure 43, and most of them will simply call a *caget* to a process variable hosted on the device. If for any reason the device does not have the ability to report the data needed in one of these APIs, its recommended that the API should return a zero.

Beam Position Monitor API List			
Gating Device API Name	Argument Types	Return Types	Units
get_device_ID	self	string	NA
get_X_position	self	double	mm
get_Y_position	self	double	mm
get_beam_current	self	double	mA
get_input_power	self	double	dBm
get_raw_BPM_buttons	self	int (x4)	Counts (x4)
get_normailised_BPM_buttons	self	double (x4)	0-1 (x4)
get_ADC_sum	self	int	Counts
get_input_tolerance	self	double	dBm

Figure 43: Abstract BPM Class APIs

So assume that the new BPM being added to the framework has the following process variables hosted in it's database:

- *.X* for the X position in µm
- *.Y* for the Y position in µm
- *.ADC.A* for the ADC counts on the A pickup
- *.ADC.B* for the ADC counts on the A pickup
- *.ADC.C* for the ADC counts on the A pickup
- *.ADC.D* for the ADC counts on the A pickup
- *.Current* for total current input to all pickups in mA

Using the private function to read an *EPICS* process variable, it's simple to implement all of these methods as a simple call to the appropriate PV. Figure 44 shows the implementation of a few of these methods given the process variables listed above. The *get_beam_current* method simply gets the PV value much like the *get_X_position* method shown in figure 42. This BPM does not have a PV to report the input power, or ADC sum. For the *get_input_power* method this will just return a zero, this is because to accurately calculate the power, the impedance of the link between the source and BPM needs to be known. As this is not known, any result will have to make an assumption, so a null value zero is returned. Conversely, wile the ADC sum can't be acquired directly from the device in a single PV, but it can be simply worked out by summing the number of ADC counts on each pickup. As such, the *get_ADC_sum* method, actually calls the *get_raw_BPM_buttons* method and then sums the results. This is deemed to be OK, as there are no assumptions made.

```
# Public method to get the BPM current
def get_beam_current(self):
  # Get the current (in mA) from the PV in the database hosted on the BPM
  current = self._read_epics_pv(".Current")
  # Gets the mean value for current if it returns more than one sample
  current = np.mean(current)
  # Returns the mean current calculated by the BPM
  return current

# Public Method to get the BPM Power
def get_input_power(self):
  # This BPM does not calculate input power, so 0 is used instead
  return 0.0

# Public Method to get the RAW ADC counts of each pickup
def get_raw_BPM_buttons(self):
  # Returns the number of counts on each ADC, each PV only returns one sample
  a = self._read_epics_pv(".ADC.A")
  b = self._read_epics_pv(".ADC.B")
  c = self._read_epics_pv(".ADC.C")
  d = self._read_epics_pv(".ADC.D")
  # Return all four ADC values
  return a, b, c, d

def get_ADC_sum(self):
  # Gets the raw counts of each pickup
  a, b, c, d = self.get_raw_BPM_buttons()
  # adds all of the counts from the pickups together to obtain the sum
  ADCSum = a + b + c + d
  # Returns the total number of ADC counts
  return ADCSum
```

Figure 44: NEW BPM Class Overridden Methods

All methods must be override in the abstract class for the child class to function in the tests. As the test itself finds it hard to know what hardware it's actually using for the test, the method *get_device_ID* is used. For devices like the RF signal generator or programmable attenuator, the main functionality of this is to ensure that the correct device is connected. For the BPMs though it's important to have a unique identifier for each BPM to know what tests were run on what hardware. The unique identifier used in this framework is the MAC (Media Access Control) address. Getting this data off of a device that is communicated to via *EPICS* is a little convoluted, but can be done. Figure 45 shows how to get the MAC address using ARP and *cainfo*. These two functions will return lots of data about a device on the network, and the rest of the functions are extracting the MAC address from the different bits of data that they return.

```python
def get_device_ID(self):
  # Pick a PV that is hosted on the device
  pv = ".X"
  # Get the IP address of the host
  node = connect(self.epicsID + pv, cainfo=True).host.split(":")[0]
  # Get info about the host using arp
  host_info = Popen(["arp", "-n", node], stdout=PIPE).communicate()[0]
  # Split the info sent back
  host_info = host_info.split("\n")[1]
  # Find the first ":", used in the MAC address
  index = host_info.find(":")
  # Get the MAC address
  host_info = host_info[index - 2:index + 15]
  # Copy the MAC address to a class variable
  self.macaddress = host_info
  # Returns the EPICS database name and the MAC Address
  return "Libera BPM: "\"" + self.epicsID + "\" + \"" + self.macaddress + "\""
```

Figure 45: *get_device_ID* method that will return MAC address of an *EPICS* BPM

With all of the information given in this section it should be possible to add a new BPM device. Adding a new RF source or programmable attenuator follows very similar steps, as long as the inheritance is correct, and the methods overridden with the correct data types then the new hardware should work sufficiently well with the tests. It's worth noting that all of the hardware classes developed have detailed docstrings containing more information on the classes and methods. If new hardware devices are added to the framework, it's recommended that they use the same docstring style of *Google Style Python Docstrings* as these are the ones used at Diamond, and in this framework to date. These have been left out of the code views in this report, as they take up a lot of space.

6.4 Adding a New Test

All of the tests currently implemented are done so, by settling initial starting values on the hardware. Then Entering a loop. The loop sets a value on the hardware, waits a small amount of time, and then takes a number of readings from the BPM or other hardware. The report section 5 details a lot more about the specific tests but also includes a test template. The template is a good starting point to ensure that the reporting is implemented correctly. All of the tests so far use a *for loop* to signals on the hardware, then take readings off of the BPM before repeating again with different values. The graph plotting is then done with the python library *matplotlib*. This is not necessarily the best way to perform tests and more elaborate tests may require more elaborate code. For now, the current tests, and the template, should provide enough information to replicate how a test can be implemented and how the data can be recoded.

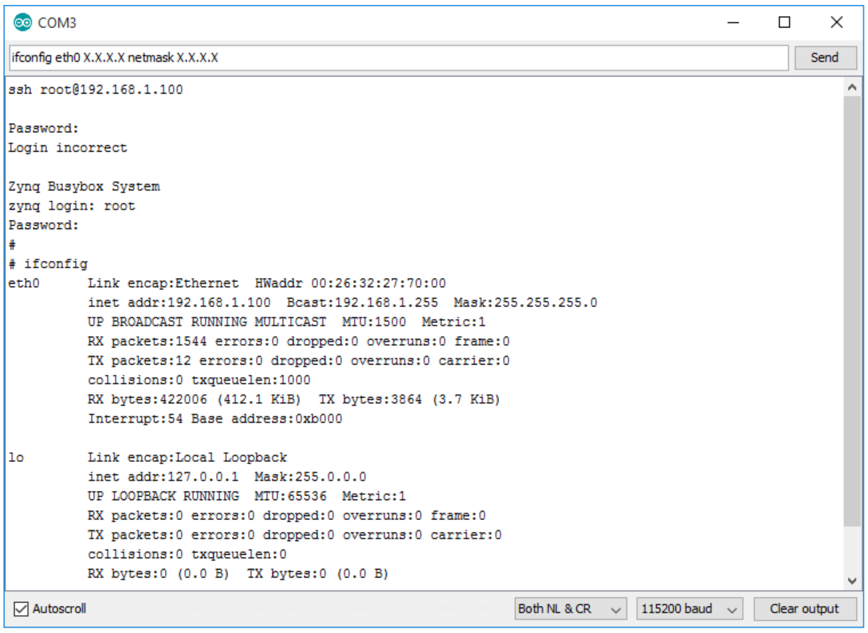
7 Useful information

7.1 IP Addresses

For this test framework a number of hardware devices have been used. To connect to these devices either an IP address or *EPICS* database name has been used. If the hardware has not been changed since development the following IP addresses and port numbers were used for the *Telnet* instrument classes.

- *Rigol3030DSG_RFSigGen*: IP 172.23.252.51, Port 5555
- *Rigol3030DSG_GateSource*: IP 172.23.252.51, Port 5555
- *MC_RC4DAT6G95_Prog_Atten*: IP 172.23.244.105, port 23

The *Rigol3030* IP address can be changed directly on the instrument, though this has been assigned a static IP for a reason, so it would be good to check if this is necessary before doing so. For the *MiniCircuits RC4DAT6G95* the IP address is changed using the software that came with the device, when leaving Diamond, the CD with this software on was inside the box for the device. To find the SCPI list of commands for these devices please see the files *SCPI_LIST.txt* and *DSG3000_ProgrammingGuide_EN.pdf*, both contained in the Tex source file directory for this report.
Another device that communicates over *Telnet* is the *Spark ER* BPM. This device will communicate over port *23* like most *Telnet* devices but as the time of writing the *Spark ER* has not been set a static IP address, partly because when it restarts the IP resets. To change the IP address of the *Spark ER* until it's powered off, one must change this via serial. Connect the micro USB change into a PC and open up a serial interface. the one used to change it here was the Arduino port monitor. Select the desired COMM port, and change the baud rate to 115200. This serial monitor can now be used to ssh into the device. Type in *ssh root@192.168.1.100* it will ask for a password, use, *Jungle*, this may take a couple of attempts. Once logged into the device, the IP address can be change using the command *ifconfig eth0 X.X.X.X netmask X.X.X.X*. Figure 46 shows the window of the program used to write these steps.



```
ifconfig eth0 X.X.X.X netmask X.X.X.X
```

```
ssh root@192.168.1.100

Password:
Login incorrect

Zynq Busybox System
zynq login: root
Password:
#
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:26:32:27:70:00
          inet addr:192.168.1.100  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1544 errors:0 dropped:0 overruns:0 frame:0
          TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:422006 (412.1 KiB)  TX bytes:3864 (3.7 KiB)
          Interrupt:54 Base address:0xb000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Figure 46: Screen shot of window used to change *Spark ER* IP address

8 Future Improvements

8.1 Permanently changing the *Spark ER* IP Address

As listed in section 7.1 the *Spark ER* BPM does not have a static IP address, but instead one that is reset every time the device is power cycled. there are steps in the *Spark ER* guide that mention setting a static IP address, but this was not looked into too hard due to time constraints on the project, and an acceptable work around for the time being. An improvement to this project would be to permanently set the IP address of the *Spark ER*.

8.2 Making the *Simulated_BPMDevice* class work with the *Simulated_Prog_Atten* class

At the time of writing, the *Simulated_BPMDevice* class works sufficiently well with the *Simulated_GateSource* and *Simulated_RFSigGen* classes. It works so much, so that if an output value is changed on the other simulators, the values read from the *Simulated_BPMDevice* object is are changed in accordance. this is not the case for the programmable attenuator. The *Simulated_BPMDevice* class is unaware of what a programmable attenuator is, and has no way of taking its readings into account for its outputs.
One way to teach the *Simulated_BPMDevice* class about programmable attenuators, would be to enter a programmable attenuator object as an input argument into the *Simulated_BPMDevice* class. Have the *Simulated_BPMDevice* class call the programmable attenuator methods so it can take their values into account when giving readings. This is how the *Simulated_BPMDevice* class works with the other simulators, so the source code should give an indication of how this could be done.

8.3 Expanding and checking the *Beam_position_equidistant_grid_raster_scan_test*

The results obtained using the *Beam_position_equidistant_grid_raster_scan_test* do not seem to line up with what is expected. the values seem to drift more than what is expected. This may be to do with the resolution of the programmable attenuator, or it may be to do with the ratios generated to set the attenuator values. Either way this deserves some inquiring to get to the root cause of the problem and fix it.
For both of the position tests, the *Beam_position_equidistant_grid_raster_scan_test* and the *Beam_position_attenuation_permutation_test*, the results are only given in the form of a graph, a table might be useful for these readings and could be included in the report. This could be implemented using the report classes method *add_table_to_test*.

8.4 Creating a safer RF class that checks BPM and attenuator values

Currently, the RF classes have an input argument called *limit* when initializing the classes. This sets a maximum output power the RF can go to so that it does not damage the hardware it's connected to. The problem with this, is that it does not take into account the attenuation value that a programmable attenuator might provide, and if the wrong argument is set into this the output power will not be capped as desired.
A way around this would be to have the RF signal generator aware of the BPM, and programmable attenuator classes, this way it can query them each time a change in power value is request, so it can check the BPM and programmable attenuator levels making sure the hardware is not damaged and the maximum safe output levels can be achieved.
The way to implement this is to make another child of the *Generic_RFSigGen* class, and give it the input arguments of a BPM object programmable attenuator object, and gate source object. Then when a new power is set on the device, it will check all of the other hardware and calculate if the new power level will damage the BPM.
A new child should be used instead of extending a current RF class as the tests will not always need all of these hardware devices, and this would be very bad for class coupling.

8.5 Adding abstraction layers to the *Tex_Report* class

As alluded to in other sections of this report there is only a single report class, *Tex_Report*. The report class could have a family and abstract parent class just like the hardware classes used in this framework, by creating an abstract parent class of the report class different types of reports could be complied from the data sent to the class via it's methods. the other classes don' necessarily need to be LaTeXreport, potentially removing the dependency on LaTeXfor this framework to report successfully.

8.6 Return the standard deviation with multi sample methods

All of the methods that return values on the BPMs return a single sample of each associated value, such that if the *get_X_position* method is called only a single value for the X position is returned. That *Spark ER* APIs request a large number of samples from the hardware, then take the mean value, and return the mean in the method. Similarly, the *Spark ERXR* can be configured to return fast acquisition or turn by turn data that will also return multiple samples where the mean is then taken. The problem with this, is that for noise or high frequency analysis, the mean value may not be sufficient. As such, the BPM could either return the samples themselves, or the standard deviation of the samples.
This can be implemented in several different ways. Either, a new set of APIs could be added, called something like, *get_X_postion_variation*. These would have to be added to the abstract class, and overridden by the child classes, so that they can be called in the same manner other methods are. Another way of implementing this would be to return the standard deviation as well as the value that's desired. For example, the *get_X_postion* method returns a single double, that is the X position. It could be changed to return two doubles, one for the X position like now, and one for the standard deviation. To do it this way would require a change in all of the method call in the current tests.
Figure 47 shows how the method calls will have to be changed in all current tests if these method return values were updated. Normally calling the method will get all of the method return types. As the return types will be updated if implemented this way, the exact return item needs to be specified.

```python
# Current syntax to get all return values
BPM.get_X_position()
# New syntax if functions updated return two values, and first is desired
BPM.get_X_position()[0]
```

Figure 47: Changes to original method calls, if method return types updated

A simple way around this, would be to include a new argument in the method calls where the default is *False*. By entering a *True* for this argument instead the function will then return two values, one for the mean position as before, and another for the standard deviation.

```python
# No arguments entered, return single position value
BPM.get_X_position()
# Optional argument used, now returns two values
BPM.get_X_position(std_dev=True)
```

Figure 48: Use arguments to update method return types

For these features to be implemented on the three BPMs used in this framework the *Electron_BPMDevice* class will need to be updated, as it only returns the slow acquisition values currently. The slow acquisition values are a single sample, to make the standard deviation value mean anything, the fast acquisition and turn by turn acquisition of this monitor will need to be added to this class. This should be implemented, by changing the *dev_ID* argument to include the *SA*, *FA*, or *TBT* values, and removing the *SA* prefix on all of the method calls.