Automated RF Conditioning

on CLARA

Now and Forever

**Duncan Scott**

**Astec**

**2-2-2018**

**Abstract**

The aim is to produce a RF conditioning and breakdown monitoring application that will be suitable for the lifetime of CLARA. This will require a modular design that can be extended and is easily configurable to different RF structures.

# Introduction

This programme is an automatic RF conditioning script that is designed to ramp the RF power following a defined ramp curve whilst also staying below a desired breakdown rate. On detecting a breakdown it is designed to switch off the RF power before the next RF pulse and record various datasets for further analysis. The programme is designed to be agnostic with respect to which of VELA/CLARA’s RF structures is being conditioned, but particular effort has been applied to the high –repetition rate photo-injectors cavity. Here, in general terms, are the main experimental goals: [[1]](#footnote-1)

1. To condition the cavities to accept the highest gradient RF field. This is done by ramping up the RF gradient maintaining the breakdown rate below some level (for example ~ 5 breakdowns per 105 RF pulses). On detection of a breakdown (or more generally an event) the power is momentarily disabled while the cavity recovers. Generally, events should be detected in the RF power traces and the power switched off before the next pulse. If the breakdown rate is too high the RF power is reduced until the breakdown rate falls below the required value.
2. A steady vacuum level should be maintained in the cavity. Excessive breakdowns can cause power to be concentrated in local hotspots causing matter to be ejected into the system, ‘*a vacuum spike*’. This matter can redistribute itself in an undesirable manner and/or damage surfaces such as the photo-cathode and cavity effecting future performance and dark current levels. It is hoped that disabling the RF power before the next RF pulse will help mitigate these vacuum events.
3. Good quality coherent data should also be saved. This includes a running log of the state of the main parameters and a ‘data dump’ that is event driven. The data dump includes the RF power traces and other ancillary data. These traces from the basis of the experimental data.

## RF Power / phase traces

The RF amplitude and phase is monitored in up to five places:

Klystron forward and reverse

Cavity forward and reverse

Cavity probe (not always available)

## What is a breakdown? What is a Mask?

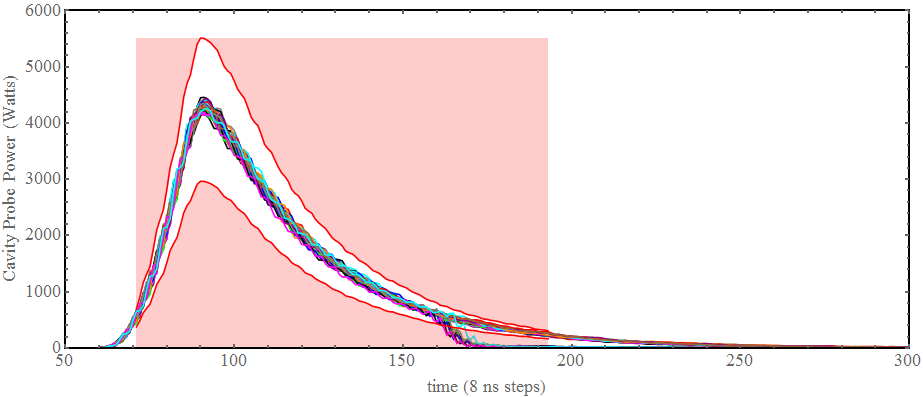
For the purposes of this programme a breakdown occurs when a trace falls outside a defined mask. The mask is defined based on previous good traces at that power and pulse length. [ref]. typically masks have high and low values derived from an absolute or percentage difference of previous values of the trace. Figure 1 shows example traces and mask. (more detail about the mask is given later)

Figure 1: example cavity probe traces with mask(red area). Outside mask traces can clearly be seen at ~170 time steps.

## Active Pulses

Breakdown rates are defined in terms of the number of pulses, this requires a counter of the RF pulses that have power in them. An active power level, typically 1000 W, is used along with the klystron forward power trace to define an active pulse. If at least one value in the trace is above the active power level the RF pulse is defined as *active* and the pulse count is increased and other checks (such as checking masks) are carried out.

## Event Pulses

Each point in the RF power ramp must achieve a specified number of active pulses before the power can be increased. Futher more there must be the required number of active pulses with no events. If an event occurs the number of event pulses should be reset to zero.

# Key Application Design Concepts

Here we will discuss the key concepts that the application is built on. Essentially, the application uses offline data with online *monitored* data to make decisions on how to control the RF. The three main components are thus Data, Monitoring and Control. These three concepts, at times, naturally blur into one another, but generally the distinction is clear.

## Control

The application control is done through a main controller that owns and updates all other objects. Hardware control and monitoring is done through the c++ hardware controllers[[[2]](#endnote-1)]. After instantiating all objects, connecting signals etc the application enters its main event loop, where decisions are made on how to progress the conditioning.

## DATA: File input/output

A number of files that are read/written

### Configuration File

On starting the application a config file is read. This sets some of the main required parameters option. The config data is put into a series of dictionaries, for example all the vacuum monitor settings are held in the ‘vac\_param’ dictionary. These dictionaries are then available to all other classes

### Pulse - breakdown log file

The programme has a ‘*memory*’ of the conditioning history from previous runs contained in ‘*pulse\_breakdown\_log.txt.*’ This enables the number of active pulses, breakdown rate, position on the rf power ramp curve and the pulse length to be carried over from day-to-day. It can be hacked to manually change the conditioning state.

### Data-log

Periodically (as defined in the config but usually every second) the current state of the data is written to file as binary data (for reasons of speed & file-size). Each file contains a plaintext header so that the type of each byte can be interpreted and an example Mathematica notebook is provided to give hints on how to read the file.

### Log.txt

As the programme runs a plaintext log file is kept that updates with messages from the various monitors and controllers. This log file is to help with debugging issues and to be able to check the input parameters when analysing the other data-logs.

### Pickle-Dumps

When an event occurs a dictionary of data is collected, these include trace history before and after the event, and other parameters, i.e. the trace that caused the event, the masks, vacuum level etc. Two types of event are distinguished: an outside mask event and a spike event. For outside mask events the number of extra traces to save after the event is specified in the config file. On a spike event the entire trace history buffer is dumped, with the number of traces defined in the config. The dictionaries are written to file as a pickle-dump, a binary format that makes them easy to re-read into python but more difficult to export as non-python.

### Ramp file

To maintain a steady increase in terms of Watts / active pulse and to allow for variable step sizes the ramp is pre-defined in a look-up table. This gives the power step in watts and the number of active pulses required to make the next step. The pulse\_breakdown\_log records the current index of the ramp file allowing the programme to easily *ramp-up* or *ramp-down* as required

## DATA: Signals

All the data from the monitored signals is held in a single, static, dictionary, called data.values. This dictionary is available to all classes through inheritance or reference. All classes can then update or read the latest data values. As well as signal values data.values holds derived data, such the state of the vacuum, the state of the dark current, i.e. is the vacuum ‘*good*’ or ‘*bad*’. The signal data is updated through a series of monitors.

## Monitors

The data monitoring is achieved through different monitor classes. All monitor have an update time, which define show often they copy data to data.values. Monitors that check the state of the vacuum, DC and breakdowns also have a cooldown timer that is started when the signal goes bad.

### Spike Monitor

An abstract class that can records a signal and keeps a moving average. If the next signal value ‘*spikes’* (i.e. is a specified level above the current mean) then then the object status goes ‘*bad.*’ The status returns to ‘*good’* via one of two modes: after a specified time in ‘*timed* ‘ mode or when the signal value returns to a certain level expressed as a factor of the current mean. When the status is ‘*bad*’ updating the rolling mean is disabled. To allow for the quickest response time to switch off the RF power a spike monitor also has flag to ‘drop amplitude to zero’ if set this will change the amplitude to the requested value on spike detection.

The ‘status’ and the ‘value’ of the signal are updated in the data.values via keys passed during construction

Example uses, vacuum and dark current spike monitors.

### Value Monitor

A value monitor uses the c++ general\_monitor to monitor a signal and write its current value to the passed data.values key. These are used to monitor temperatures, and solenoid currents.

### Hardware Monitors

Specific hardware, such as the modulators, vacuum valves, RF protection are monitored using the bespoke c++ hardware controllers. These hardware controllers could be used at later date to control as well as monitor the hardware (e.g. reset the modulator after a trip).

### *‘simple’* LLRF parameter Monitor

This monitors ‘*simple*‘ llrf parameters. These are the amplitude set point, the llrf trigger and output state, and, amplitude and phase lock. It also gets the mean power values for the traces being monitored. These are calculated in the c++ llrf controller by specifying a start and stop index over which to calculate the mean, this monitor then access those values

### Outside mask trace monitor

The monitoring of the traces and the setting the mask happens in the c++ LLRF controller. This class mainly monitors whether a new outside mask trace has been detected and if so updates the breakdown count and collects the data as it becomes available. (Remember that future traces are saved after an outside mask trace is detected so that it can be confirmed that the power was disabled in a single tick.)

# Design Schematic

DATA

CONFIG READER

DATA

LOGGERR

BASE

DATA

MONITOR

GUI

MAIN

CONTROLLER

C++ HARDWARE CONTROLLERS

|  |  |
| --- | --- |
| DATA | Holds a dictionary with all the single value data, i.e. current vacuum level, cavity reverse power, DC state. This dictionary is accessed by all other classes to update / read latest values  Has a timer that writes the data\_log.dat file |
| CONFIG READER | Reads the config file and puts the input data into a series of dictionaries, for example all the vacuum monitor settings are held in the ‘vac\_param’ dictionary. These dictionaries are then available to all other classes |
| DATA  LOGGERR | Knows how to read and write all files associated with the application, ***APART from*** the ‘config file’. Has a messaging function. Messages can be written to stdout and also to a txt file |
| C++ HARDWARE  CONTROLLERS | c++ hardware controllers, for: LLRF, vacuum valves, RF modulator, RF protection, .. |
| BASE | A utility class to hold all the lower level classes in one place and acts as a single point of contact for the higher-level classes.  Is also a good place to put general functions that can be used in higher level classes |
| GUI | The GUI classes are derived from a QTDesigner file that is then inherited by the top level class, gui\_conditioning.  To make updating the GUI values easy output widgets are held in a dictionary with the same keys as the data.values dictionary. The gui has an update timer that refreshes the gui widgets |
| DATA  MONITOR | The data monitor class hold a series monitor classes. These include abstract monitors such as Spike Monitors and value monitors, and specific hardware monitors such as vacuum valve, rf protection and modulator monitors. Also has a c++ general monitor object that is used to monitor values that do not require control (water temperatures, etc.)  In principal, (but not yet in practice) the monitors are also able to control the hardware. For example, The modulator monitor could reset the modulator after a trip. |
| MAIN  CONTROLLER | This class holds all the objects, and controls the flow of the programme. After all objects are instantiated, signals connected, monitoring started and data is being updated the main controller enters the main event lop. This decides how to ramp up and down the RF |

# Main Event Loop

The main event loop is effectively a decision tree that uses the data generated from all the other loops and monitors to control the ramp.

It relies on 4 states and a number of flags;

The four states that are monitored in the main event are:

vacuum – is the vacuum level ok, and not in cool-down

Dark current – is the vacuum level ok, and not in cool-down

outside\_mask – are the traces inside the mask and not in cool-down

the value of these states can be: good, bad, new\_good, new\_bad. New good/bad means the state has changed to that value this iteration.

The 4th state is rf\_enabled. This is either good or bad. Where good means everything that is required to provide RF power to the cavity is good, i.e. modulator is on, RF permits are good, etc.

If new\_bad: (i.e. a new vac/DC spike, outside trace

Set 0 RF power

Set 0 event pulses

If new\_good and no bad: (i.e. after returning from an event)

If breakdown rate is high:

ramp\_down()

else

continue\_ramp()

If all good:

If reached required event pulse count:

If breakdown rate is not high:

ramp\_up()

ramp\_down(), ramp\_up() and continue\_ramp() use the current ramp index to move up, down or continue along the ramp curve.

Every time the RF power is changed the masks must be reset. Ideally, the changes in power will not be large enough to make the new trace fail the existing mask.

# References

1. ‘ADVANCES IN HIGH-GRADIENT ACCELERATING STRUCTURES AND IN THE UNDERSTANDING GRADIENT LIMITS’ W. Wuensch, CERN, IPAC 2017 [↑](#footnote-ref-1)
2. <https://trello.com/c/SJCXJTlG/100-software-c-hardware-controllers-1-djs> [↑](#endnote-ref-1)