CERN-ACC-NOTE-2021-0011
2021-04-13

Richard.Scrivens@cern.ch,
Maciej.Peryt@cern.ch

The Linac4 Source Autopilot

BE-ABP, BE-CCS: M. Hrabia, D. Noll, M. Peryt, R. Scrivens

Keywords: Linac4, Source, Controls

Summary

The Linac4 Source Autopilot improves the stability, uptime and monitoring of the Linac4 source, by using high level automation to control and monitor the device parameters of the source. It consists of a framework incorporating standard CERN accelerator controls infrastructure, to which users add specific code for specific use cases.

1. Introduction

The Linac4 ion source is a 2MHz RF driven H- ion source, using caesium injection to enhance H- production and lower the electron to H- ratio. The source operates for Linac4 with 800µs long pulses and 1.2 second intervals.

The stability of the H- beam intensity from the source over the period of minutes to days requires adjustment of parameters like the RF power used for plasma heating. Controlling the source in the time domain of minutes to days is the objective of the Autopilot, it is not conceived to work from pulse to pulse, or within a pulse. Both of these would require dedicated systems at the front-end level.

An Inspector Autopilot was developed [1] using Java within the Inspector environment [2] however issues with maintainability called for a new approach which is detailed herein.

A new Autopilot Framework was developed in 2019 and used successfully to automatically tune the source during test runs for Linac4. Within this framework, users can develop algorithmic tasks that are typically run continuously to adapt device settings based on acquisitions received. Typical use cases are slow feedback systems and automated procedures (like resetting and restarting equipment that has stopped due to a fault state).

This document represents the status of the Autopilot at the end of 2020.

2. Framework

The Autopilot Framework, henceforth referred to as the framework, is a set of services and tools (see Figure 1) that allow the users to deploy and execute their algorithmic tasks in a self-service manner, where the framework provides the necessary tooling and infrastructure.

The framework allows the users to subscribe to the Control parameters of their choice, process the received values, and publish the output as Virtual Device Properties. Those Virtual Devices can in turn be used like any other Control Device, in particular they can provide inputs to the user-supplied regulation algorithms (subsequently referred to as actors) that interact with the Linac4 H- ion source.

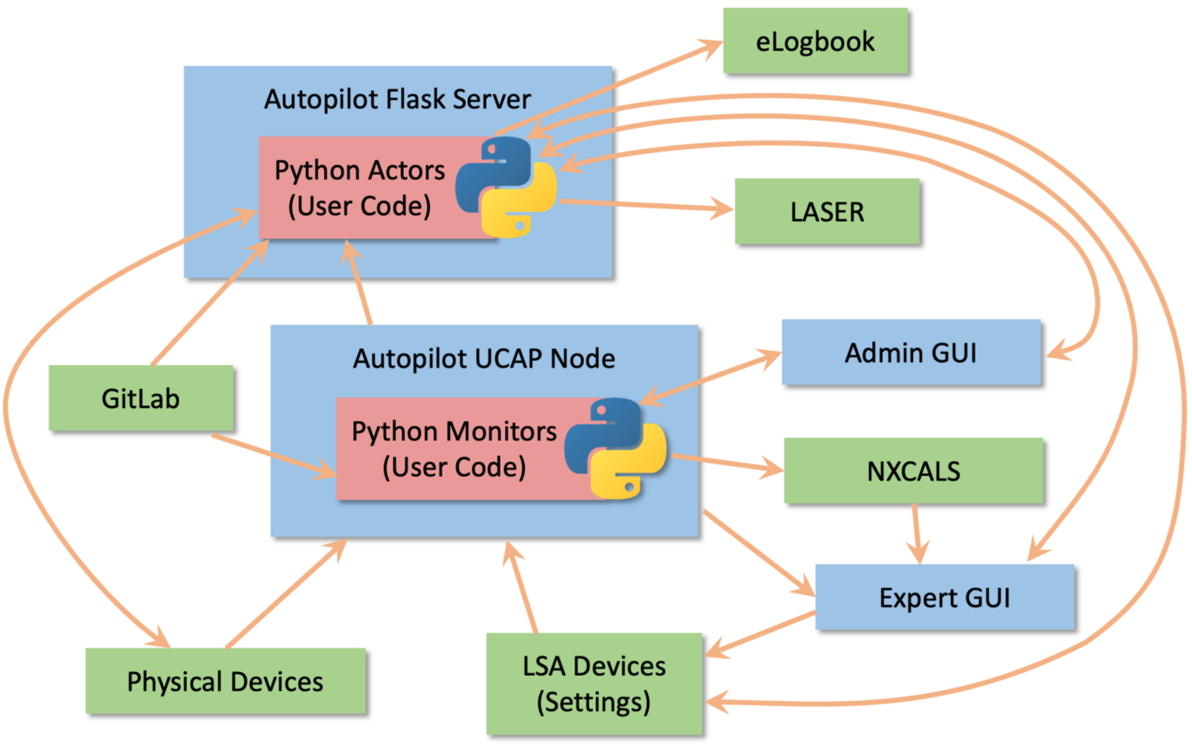


Figure 1. Schematic diagram of the Autopilot Framework.

A particularity of the framework is that although it is based on the Accelerator Control System software stack, that is predominantly developed in Java by BE-CSS (previously BE-CO), it allows the user code to be written in Python. Another characteristic of the framework is that it leverages the well-established services and components that form the CERN Accelerator Control System, in particular CMW, RBAC, the UCAP framework, LSA, LASER, and NXCALS, thus reducing the need for developing custom components.

At the core of the framework lies a UCAP (Unified Controls Acquisition and Processing) node that subscribes to physical and virtual Devices according to the recipes provided by the users. The recipes also contain the triggering conditions. When those conditions occur, the events containing the accumulated data are constructed and forwarded to the user-provided transformation routines, known as “Monitors”. The Monitors calculate the output values and give them back to UCAP, for publishing as virtual Device Properties through a standard RDA3 mechanism.

Along with the UCAP recipes, and the Monitors, the Flask server also manages the user-provided control tasks called “Actors”. These are basically Python scripts whose job is to act upon the updates received from the Monitors and perform the necessary actions to keep the parameters of the Linac4 source within the requested limits. The benefit of the Actors using Monitors, is that the UCAP system takes care of synchronizing multiple data inputs, allowing the Actor to receive all required data in a single subscription.

2.1 Server

The server is a REST service powered by the Flask micro-framework combined with Gunicorn as HTTP Server. The REST communication is hidden behind a simple web interface, which serves as a bridge between the user and the framework itself. The service was built with the use of Acc-Py [3], a Python distribution provided by the controls group.

The web interface (see Figure 2) consists of three subpages, each dedicated to the separate part of the Autopilot Tasks: Monitors, Actors and UCAP configuration. Each of them provides a status view of uploaded user scripts, as well as interactive tools to manage the content of the service, such as adding and removing script files or starting and stopping Task execution.

To make sure that the communication is secure, an HTTPS connection to the server has been set up, with the help of tools provided by CERN Certification Authority, giving an encrypted communication channel between the service and the user.

To restrict the access for unauthorised users, RBAC [4] is used for user authentication. On top of that, to distinguish between the different types of users and restrict access even further, three internal roles have been defined:

- Guest – assigned by default to a successfully authenticated user (meaning someone with a CERN account and with trusted network access). This type of user can only see the list of tasks and their statuses, but cannot interact with them in any way.

- Operator – assigned manually to a specific account – can also start and stop Tasks.

- Super user – assigned manually to a specific account name – can also add new, modify and remove existing Tasks. Reserved for people with expert knowledge on the source control.

The web interface also provides a set of features to help the users self-service their Tasks in case they do not behave as expected, such as:

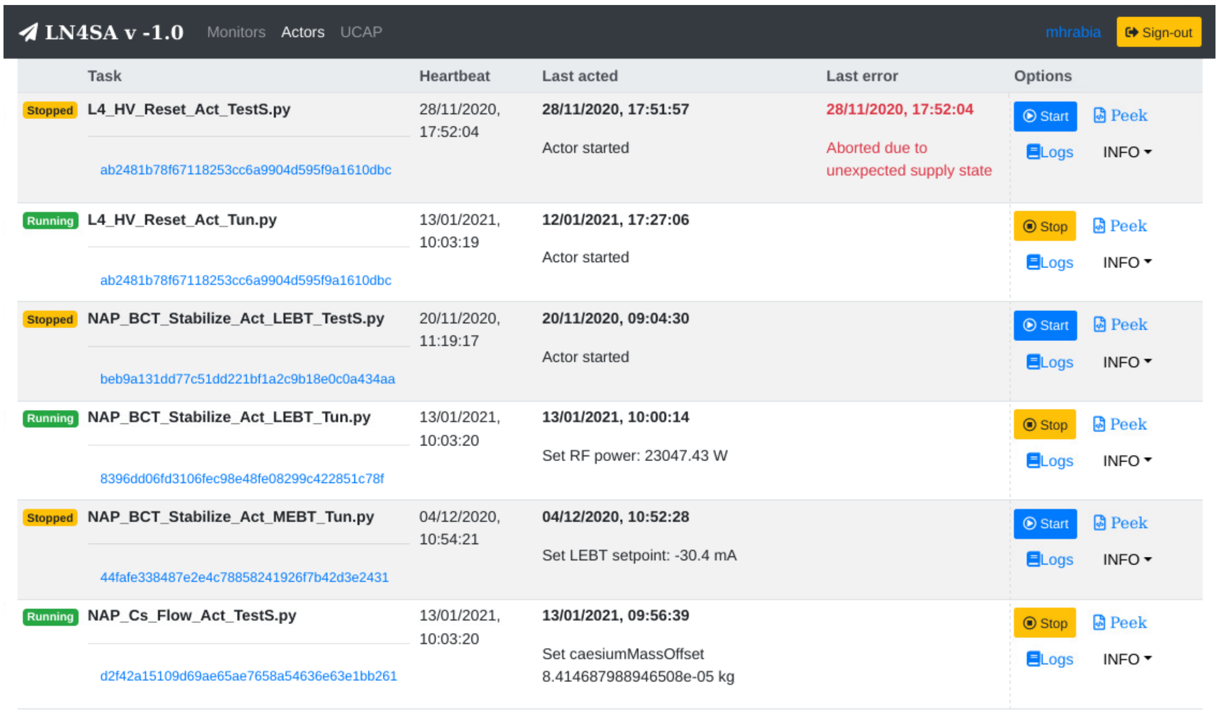
- Checking python script correctness at the startup time – when the user tries to start a new python script, the check for basic code errors, like wrong indentation level, is performed. Since obvious mistakes like this would prevent the script from running correctly, an error message specifying the reason of the failure is shown to the user, and the new process is never started.

- Logging mechanisms – the user is provided with predefined logger, which can be used inside user's script to log any useful information during code execution. User logs are stored on the server and are easily accessible through the web-interface. The logs are being rolled every day at midnight (or after exceeding a specified size) to help trackback potential issues be specifying the exact time range of events stored in a particular log file.

- Automatically catching runtime exceptions – for more subtle errors resulting in exceptions that can happen at runtime, the framework is able to catch them and put them in the user logs, freeing the user from having to worry about handling such scenarios.

- Subscription status preview – for each Monitor Task, the list of all its Data Sources and their connection statuses is provided. This allows the user to see in real time if the data from each Data Source is available. For quick debugging of broken subscriptions, a remote ‘test get’ feature is provided, however it currently works only with RDA3 devices.

- File peeking – the user can visualise the task code directly in the web interface.
- Test data generation – before the Monitor script will be uploaded to the server it has to be written offline. Python at its core is a language where it is very easy to make a mistake, so we give the possibility to record a specified amount of input data from UCAP in the form of JSON files, that can be downloaded by the users and fed to their algorithms offline, making it easier to find some obvious errors, before the final version will be uploaded to the server.



	Task	Heartbeat	Last acted	Last error	Options
Stopped	L4_HV_Reset_Act_TestS.py ab2481b78f67118253cc6a9904d595f9a1610dbc	28/11/2020, 17:52:04	**28/11/2020, 17:51:57** Actor started	**28/11/2020, 17:52:04** Aborted due to unexpected supply state	Start Peek Logs INFO
Running	L4_HV_Reset_Act_Tun.py ab2481b78f67118253cc6a9904d595f9a1610dbc	13/01/2021, 10:03:19	**12/01/2021, 17:27:06** Actor started		Stop Peek Logs INFO
Stopped	NAP_BCT_Stabilize_Act_LEBT_TestS.py beb9a131dd77c51dd221bf1a2c9b18e0c0a434aa	20/11/2020, 11:19:17	**20/11/2020, 09:04:30** Actor started		Start Peek Logs INFO
Running	NAP_BCT_Stabilize_Act_LEBT_Tun.py 8396dd06fd3106fec98e48fe08299c422851c78f	13/01/2021, 10:03:20	**13/01/2021, 10:00:14** Set RF power: 23047.43 W		Stop Peek Logs INFO
Stopped	NAP_BCT_Stabilize_Act_MEBT_Tun.py 44fafe338487e2e4c78858241926f7b42d3e2431	04/12/2020, 10:54:21	**04/12/2020, 10:52:28** Set LEBT setpoint: -30.4 mA		Start Peek Logs INFO
Running	NAP_Cs_Flow_Act_TestS.py d2f42a15109d69ae65ae7658a54636e63e1bb261	13/01/2021, 10:03:20	**13/01/2021, 09:56:39** Set caesiumMassOffset 8.414687988946508e-05 kg		Stop Peek Logs INFO

Figure 2. Screen shot of the Autopilot Tasks web interface, showing Actors and their statuses.

2.2 Source code management - GitLab

To manage the state and content of user script files, the server is connected to a dedicated git repository hosted on CERN’s GitLab [5]. The repository contains a configuration file listing explicitly all Actors, Monitors and UCAP configuration files. This allows the server to easily recognize the expected role of the particular file and handle it accordingly.

By using GitLab, out of the box we get the history of changes on each and every file, having a clear trace path of how the file content was changed, by whom and when. It also gives a simple way to roll back to a previous version of the file in case the new version is not behaving as expected.

The server uses the GitLab REST API to compare files from the repository with those currently hosted on the server as well to download them if necessary.

From the server perspective, updating any script file is as simple as two clicks on the web panel. First click to 'check' for any changes, where the server compares commit IDs of the files, and the second click to 'synchronize' the file, meaning downloading the latest version of the file from the GitLab repository if it is newer than the version currently on the server.

For safety reasons the script files can only be synchronized if the corresponding Autopilot Task is not in a 'running' state. However, the version check can be done at any time, as it does not impact the content of the server.

When any script file has been recognized as outdated – a notification label is shown next to the file name on the web panel, making it easily noticeable. As an added value, the label itself is a hyperlink to the GitLab webpage commit details, which highlights changes in the script file content between versions, thus allowing for a quick verification before deciding to synchronize script files to the server.

The server is also capable of being notified by a GitLab web hook automatically, after any file has been changed, sparing the user the necessity of invoking the 'check' operation manually. However, the current network configuration prevents GitLab to send those web hook calls to a server hosted in CERN's Technical Network (discussions have been launched to try and address this).

No automation is foreseen for the synchronization of script files. Since those files are actually running on the server, it is critical for stable operations to only update runtime content when there is a safe operational window for doing that. This update can only be performed by super-users, as defined in section 2.1 .

2.3 UCAP

The Unified Controls Acquisition and Processing (UCAP) project aims to provide "A generic, self-service, controls data processing platform" [6].

In simpler terms, UCAP is a platform that allows to create an RDA3 based Virtual Device, which receives data from a predefined list of signals, gives that data to the user written or configured code (known as a Transformation), which can in turn perform any kind of calculations and processing, and publishes the result back to the Controls system using a standard middleware protocol. State is retained between calls to the user transformation, so that it is possible to apply transformations over multiple beam cycles (for example averaging a value over some time window).

The UCAP-based Virtual Devices can be registered in CCDE, allowing anyone to subscribe to the data they publish, and for that data to be recorded in the NXCALS data logging system.

At the core of the Linac4 Source AutoPilot, Monitors are in fact UCAP Virtual Devices, publishing transformed data that is in-turn used by the Actors to interact with the Source. Corresponding data is stored in NXCALS and can be used later for any purpose such as diagnostics.

2.3 Monitors

As explained above, Monitors are Virtual Devices running on a UCAP node. The Autopilot server simply serves as a man-in-the-middle between the user and the UCAP framework, meaning the user uploads UCAP configuration and transformation code files through the use of the aforementioned web panel, and the Autopilot servers takes care of turning that into a request understandable by the UCAP framework.

It is the responsibility of the UCAP framework to create a new process to receive, transform and publish data. The Autopilot server does nothing more than ask it to add/remove or start/stop the Virtual Device in question.

The data transformation code is written in Python using a `ucap-python` [6] package, which contains all interfaces representing the data provided by UCAP. The usage of this package is mandatory for every Autopilot Monitor script.

To facilitate code reuse and maintenance, the common code used in all Monitor scripts was extracted into another Python package called `nap-utils` [8]. This package (unlike `ucap-python`) is not mandatory, but it can help keeping Monitor code cleaner.

By design, Monitors can run continuously, as they only read the data published by other Devices and do not interact themselves with the Control system.

Monitors can also receive settings for their execution by including LSA virtual settings values in their data input. For example, a minimum and maximum needed to perform a value validation can be created as a LSA virtual setting, and input to an Autopilot Monitor along with the real Device acquisition for comparison. In this way the LSA parameters allow many different ways to incorporate settings into Autopilot applications, at the same time keeping a history of changes to these settings inside the LSA system.

2.4 Actors

Actors are Tasks that continuously receive data produced by Monitors, and possibly some other sources such as LSA settings (both from real Devices, and Virtual Devices). Based on these inputs they can decide to invoke a particular action if necessary. Each Actor Task runs as a separate process.

A typical Actor Task uses the `pyJAPC` library to subscribe to the data produced by its related Monitor, or any other data source.

Actor Tasks can use a special API to notify the Autopilot server about certain actions, which can be later shown on the web panel (and other services like the eLogbook and LASER systems). This gives useful real-time feedback of a running Actor's behaviour as follows:

- `heartbeat` – an Actor script can call this method to announce that it is alive. It is usually invoked by the user whenever new data is received by the Actor. The web panel shows the timestamp of the last heartbeat of every Actor.

- `set_last_acted` – an Actor script can notify that it is performing an action it considers as an ‘act’ method (like applying a new hardware setting) and provide a short message describing it. The message and corresponding timestamp are shown on the Actor Tasks web panel.

- `set_last_error`/`clear_last_error` – an Actor script can notify the server that it recognized a situation considered as an error, for example that it received corrupted data. The error message and corresponding timestamp are shown on the Actor Tasks web panel.

- `abort_task` – an Actor script can ask the server to cleanly abort the Task execution for example where a set of conditions have no path defined, and the situation is considered unsafe for the actor. The Actor Tasks web panel will show the task in an error state, explicitly stating that it has been aborted.

- `write_to_elogbook` – an Actor script can write a short message to the eLogbook service.

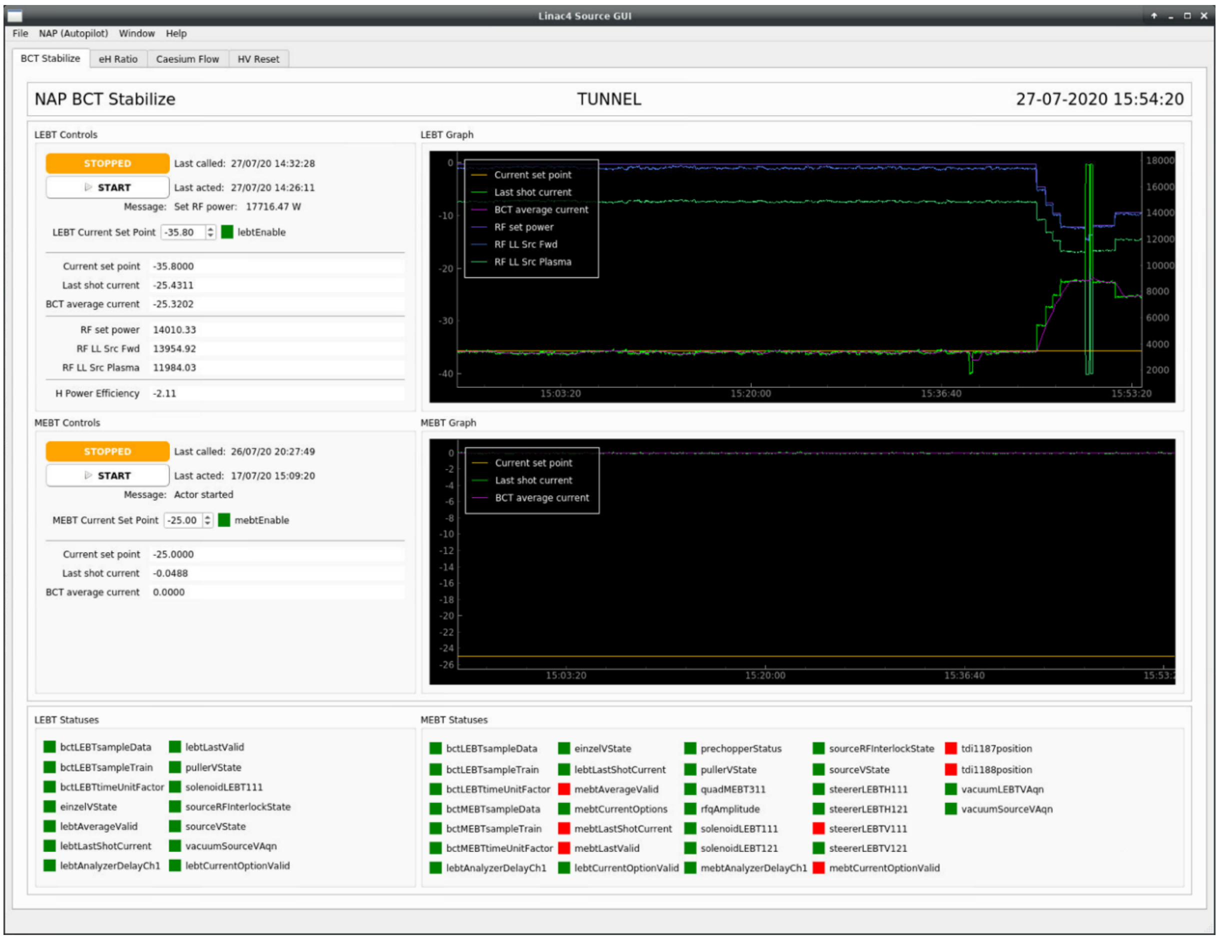
- `set_alarm / clear_alarm` – an Actor script can create a new alarm that will be shown on the LASER console expanding further possibilities of notifying operators that something on the Autopilot requires attention.

Unlike Monitors whose lifecycle is managed within UCAP, the lifecycle of Actor Tasks, is managed directly by the Autopilot server. This means the Autopilot server is responsible for creating a new process and shutting it down later when the user requests the Task to be started or stopped, respectively.

2.5 GUI

The Autopilot can be Monitored and, to a certain extent, controlled through an application (GUI) that can be started from the Common Console Manager (CCM). The screenshot in Figure 3 shows the BCT Stabilize panel of the GUI. The panel provides a graphical overview of the current and historic state of the BCT stabilization Task, discussed in more detail further in this document. It also allows to start and stop the regulation Tasks, and to control the current set-points.

Along with the BCT Stabilize panel, the GUI provides a number of additional panels with a similar functionality, namely the eH Ratio panel, the Caesium Flow panel, and the HV Reset panel.



NAP BCT Stabilize TUNNEL 27-07-2020 15:54:20

LEBT Controls

STOPPED Last called: 27/07/20 14:32:28

START Last acted: 27/07/20 14:26:11

Message: Set RF power: 17716.47 W

LEBT Current Set Point -35.80 🟩 lebtEnable

Current set point	-35.8000
Last shot current	-25.4311
BCT average current	-25.3202
RF set power	14010.33
RF LL Src Fwd	13954.92
RF LL Src Plasma	11984.03
H Power Efficiency	-2.11

MEBT Controls

STOPPED Last called: 26/07/20 20:27:49

START Last acted: 17/07/20 15:09:20

Message: Actor started

MEBT Current Set Point -25.00 🟩 mebtEnable

Current set point	-25.0000
Last shot current	-0.0488
BCT average current	0.0000

LEBT Statuses

🟩 bctLEBTsampleData	🟩 lebtLastValid
🟩 bctLEBTsampleTrain	🟩 pullerVState
🟩 bctLEBTtimeUnitFactor	🟩 solenoidLEBT111
🟩 einzelVState	🟩 sourceRFInterlockState
🟩 lebtAverageValid	🟩 sourceVState
🟩 lebtLastShotCurrent	🟩 vacuumSourceVAqn
🟩 lebtAnalyzerDelayCh1	🟩 lebtCurrentOptionValid

MEBT Statuses

🟩 bctLEBTsampleData	🟩 einzelVState	🟩 prechopperStatus	🟩 sourceRFInterlockState	🟥 tdi1187position
🟩 bctLEBTsampleTrain	🟩 lebtLastShotCurrent	🟩 pullerVState	🟩 sourceVState	🟥 tdi1188position
🟩 bctLEBTtimeUnitFactor	🟥 mebtAverageValid	🟩 quadMEBT311	🟩 steererLEBTH111	🟩 vacuumLEBTVAqn
🟩 bctMEBTsampleData	🟩 mebtCurrentOptions	🟩 rfqAmplitude	🟩 steererLEBTH121	🟩 vacuumSourceVAqn
🟩 bctMEBTsampleTrain	🟥 mebtLastShotCurrent	🟩 solenoidLEBT111	🟥 steererLEBTV111	
🟩 bctMEBTtimeUnitFactor	🟥 mebtLastValid	🟩 solenoidLEBT121	🟩 steererLEBTV121	
🟩 lebtAnalyzerDelayCh1	🟩 lebtCurrentOptionValid	🟩 mebtAnalyzerDelayCh1	🟥 mebtCurrentOptionValid	

Figure 3. The Autopilot GUI, BCT Stabilize panel.

2.5 ELogbook

To allow the Autopilot server to communicate its important actions to the outside world when necessary, Actor Tasks have the possibility to write messages to the new eLogbook [9] provided by OP. Behind the scenes the server uses the REST API of the new eLogbook to write messages from the Actor Task to the Linac4 activity of the eLogbook.

This functionality is limited solely to Actor Tasks, as they are the only processes whose actions can have implications for the operation of the accelerator, and thus should be announced to operators when necessary.

Messages can be written at any time by the Actor, but in principle it should be used only to write important information that should be visible by the operators on shift.

2.5 LASER

The Autopilot server is integrated with the LASER service, giving Actor Tasks the possibility to create a new alarm that will be visible by operators on the LASER console. The functionality is limited to Actor Tasks for the same reason as explained above for the eLogbook integration.

Behind the scenes the communication with LASER server is done using python bindings built on top of the existing C++ libraries, provided by the LASER team.

At any point of its execution an Actor Task can put itself in an alarm state, resulting in a notification, with new Actor alarm details being sent to the LASER service. However, the LASER console will only show one alarm for the Autopilot server.

The content of that alarm contains a list of key-value pairs of Actor Tasks currently in an alarm state together with their corresponding alarm messages. This means that if no Task is currently in the alarm state the LASER console will show no alarm from the Autopilot server, however, if there is one or more Actor Tasks in an alarm state, the LASER console will only show one alarm active on the Autopilot server, and the operator will have to see its details to know exactly how many Actor alarms the server is reporting.

This implementation decision was made to overcome a limitation of LASER, which requires every alarm to be registered in the LASER database upfront – the approach which clashes with the dynamic nature of the Autopilot, where new Actor Tasks can be added to the system at any time.

Having a separate alarm for every Actor would require either:

- Sending a support request to the LASER team every time a new Actor Task is created, which is easy to forget, not pleasant to do, and would have to be tested separately every time to make sure the alarms work as expected.

Or

- Integrate the Autopilot server deeply with the LASER database, meaning any changes on the LASER database could trigger additional work on the Autopilot server side to adapt to those changes, making it very hard to maintain in the long run. It would also require testing any newly added alarm to make sure it works as expected.

The current implementation deals with all of those issues, freeing both sides from having to do any extra work to allow new Actor Tasks to use the alarm mechanism, and does not require any additional testing to make sure the LASER console will show it correctly.

2.6 LSA

LSA is used by the Autopilot to manage Actor settings. This allows to use the standard tools such as working sets / knobs, and LSA App Suite, to influence the behaviour of the Autopilot. It is also possible to adjust a limited number of those settings using the Autopilot GUI, namely the set-points for the LEBT/MEBT BCT current, and the requested eH ratio.

2.7 NXCALS

NXCALS is used by the Autopilot to log the values published by the Monitors. Data is also extracted from NXCALS when starting the Autopilot GUI, to prefill charts with a certain amount of historical data.

2.8 Deployment

The two core parts of the framework are: the Flask Autopilot Server, and its dedicated UCAP Node (see Figure 4). The Flask Autopilot Server, as an ACC-PY based service, uses ACC-PY tools for the releases of new versions. However, for the deployment on its dedicated server, a custom deploy script had to be developed, as currently ACC-PY lacks tools for services deployment. The UCAP Node, being Java application at its core, uses CBNG for both releasing and deployment.

Both services run on the same server running CentOS 7. Although ordinarily UCAP Team takes care of managing UCAP Nodes for their users, it was decided that the Autopilot Framework's UCAP Node will be maintained and hosted separately by the Autopilot Team, for two principal reasons:

- custom solution - for example the Gitlab integration of the Autopilot Framework uses UCAP Python support in a very non-standard way. Ordinarily UCAP requires Python scripts to be released using ACC-PY ecosystem, meaning every little script change has to go through a heavy process before it can be picked up by UCAP. The customised solution of Autopilot Framework relieves the user from all the ACC-PY release burden. Although less heavy and more user friendly, the current custom solution requires both the Autopilot Flask Server and its dedicated UCAP Node to be hosted on the same machine. A similar requirement comes from the management of the script logs that are available to the user for debugging purposes.

- historical - UCAP framework was under heavy development at the very same time that the Autopilot Framework. This resulted in frequent, and often backward incompatible, updates to the UCAP framework, that the actively developed Autopilot Framework couldn't keep up with. Having a separate UCAP Node under the sole responsibility of the Autopilot Team allowed the framework to be updated at a more convenient pace and at the same time it freed the UCAP Team of having to worry about one particular client with the needs and the update schedule different than the rest of their clients, which is still the case today.

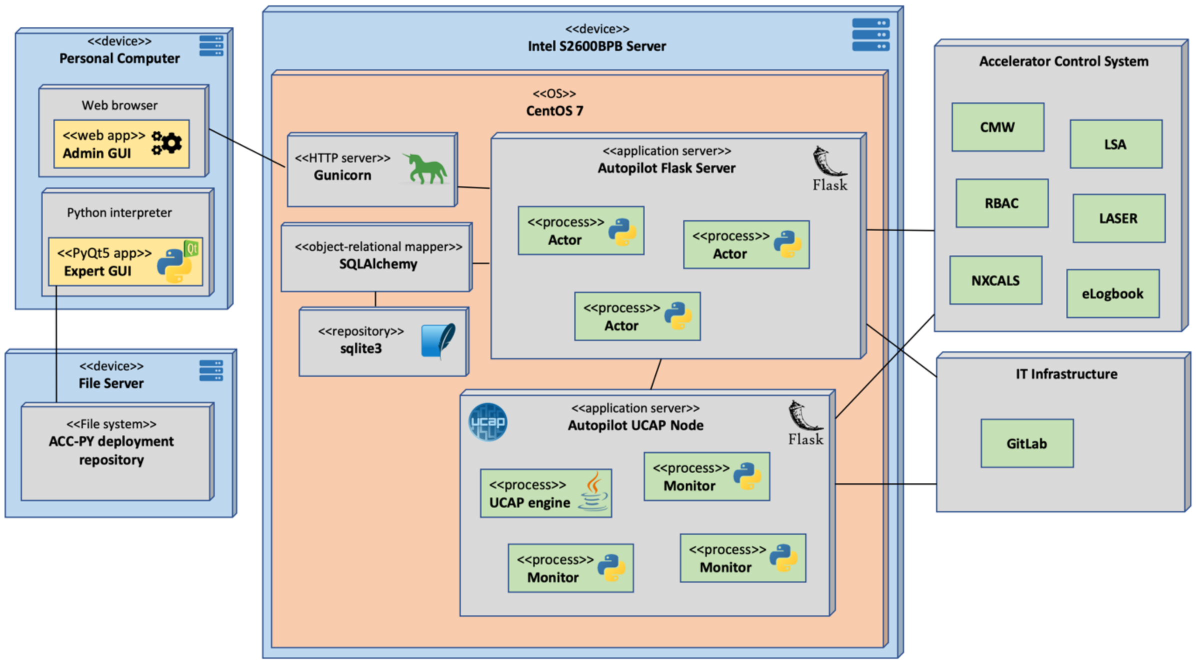


Figure 4. The Autopilot deployment diagram

3. User Tasks for the Source Autopilot

In this section we describe the Tasks available and their present status.

The Tasks are under almost constant development, and therefore the latest descriptions can be found in the wiki site [10].

The Tasks are coded to work both at the Linac4 source and at the Test Stand (which is used for testing sources), with independent settings and validation parameters. Each Monitor and Actor can be started and stopped independently, and when running there is no coupling between the Linac4 source and test stand.

Overall, the Monitors and Actors are kept consistent between the Linac4 source and the Test Stand, with the Linac4 source being the master, and the two copies being identical except for a single line variable that switches any differences in the code (for example, there is no RFQ at the test stand, meaning data validation for its amplitude is not possible, a condition that is handled in the code).

For Monitors, the input data (i.e. the list of real Devices that supply data) is then differentiated through the UCAP json configuration file. The configuration file allows UCAP to supply to the Monitor code data depending on the instance (e.g. the json file configures a power converter to be either one in the Linac4 tunnel, or on the test stand, supplying the data to the Python code under the same alias name).

For Actors the situation is fully handled within the Actor, using a configuration dictionary defining all the different Device names for the Linac4 source and the tunnel. Again, a single variable name change allowing the Actor to select the correct names at run-time.

In both cases for Monitors and Actors, one instance of the configuration, Monitor transformation code and Actor code must be uploaded for each of the Linac4 Source and Test Stand. Nevertheless, the code for a Monitor or Actor can be easily modified only on one instance. Therefore, code modifications can easily be made for the Test Stand, validated, and only then propagated to the Linac4 source during a suitable moment.

In the sections below more details are given on individual Tasks. Italicized names are aliases for full device/property#fields that are given in the table at the end.

3.1 BCT Stabilize

This Task is the main workhorse of the Autopilot and is illustrated in Figure 5. Its objective is to keep the H- beam current constant on either the *LEBT-BCT* or *MEBT-BCT* (BCT=Beam Current Transformer) by adjusting the forward power of the 2MHz RF used for plasma production and heating.

The LEBT BCT Stabilize Task reacts over the order of 1 minute, averaging the beam current during this time window and only then changing the RF power if the average beam current is outside a defined dead band.

The configuration of the source, LEBT and RFQ needs to be confirmed to be valid before the feedback loop acts, hence each shot is validated (using equipment Device fields) and only if it passes the validation test, is the data added to the averaging data stack. The Autopilot will furthermore only vary the RF power if the number of shots that were valid in the last minute exceed a threshold (presently set to 20%).

The MEBT-BCT Stabilize task acquires the *MEBT-BCT* current and if the average is outside the required dead band, it requests a change in the current set-point of the LEBT-BCT Stabilize task. If the MEBT-BCT Stabilize Task is not running, the set-point of the LEBT-BCT Stabilize task will be static, and the feedback will only regulate the source RF power.

Furthermore, the Autopilot has to switch automatically between regulating on the *MEBT-BCT*, and back to the *LEBT-BCT* in the case that the beam is stopped through the RFQ (e.g. RFQ fault or the beam-stoppers are put into the beam). This is achieved through the validation process, if there are insufficient valid current acquisitions in the previous 1 minute, the Actors will not request any change, and therefore they become temporarily inactive. In this way, if the beam is not transferred through the RFQ due to any fault condition, the MEBT BCT Stabilize Actor becomes inactive until the time it receives sufficient new valid data.

The *MEBT-BCT* pulse length is cut by the pre-chopper in a PPM configuration, and therefore the regulation could be dependent on artifacts coming from the different pulse lengths. In order to avoid this situation, the *MEBT-BCT* current is calculated from the time resolved data inside a fixed window (uniquely set up for the Autopilot) and if the beam is cut short in this window, the pulse is not validated. In this way only pulse lengths longer than some defined window (presently 80µs) are counted.

An example of the regulation system in action, over 5 days, is given in Figure 6, where it is shown that this Actor regulates the RF power by the order of 10% over the full-time scale in order to keep the beam intensity constant. Figure 7 shows a comparison periods for the BCT-Stabilize actor task being stopped and started (see caption for details). The increased stability in the dark blue and cyan curves is clearly visible.

This Task in addition to the standard “Start/Stop” possibility, also uses a *EnableLEBT* and *EnableMEBT* specific Boolean setting, that can be used to inhibit the loop while not stopping the Actor code. This is used exclusively by Linac4 operations to inhibit this Actor when the Linac is used in a low intensity mode for RF rephasing.

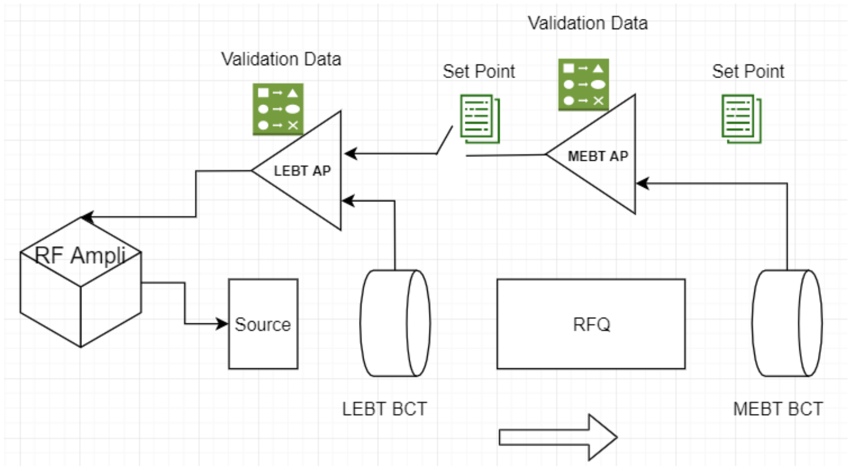


Figure 5. Schematic diagram of the BCT Stabilize Autopilot, with the large arrow showing the beam direction. LEBT AP and MEBT AP refer to the software feedback loops.

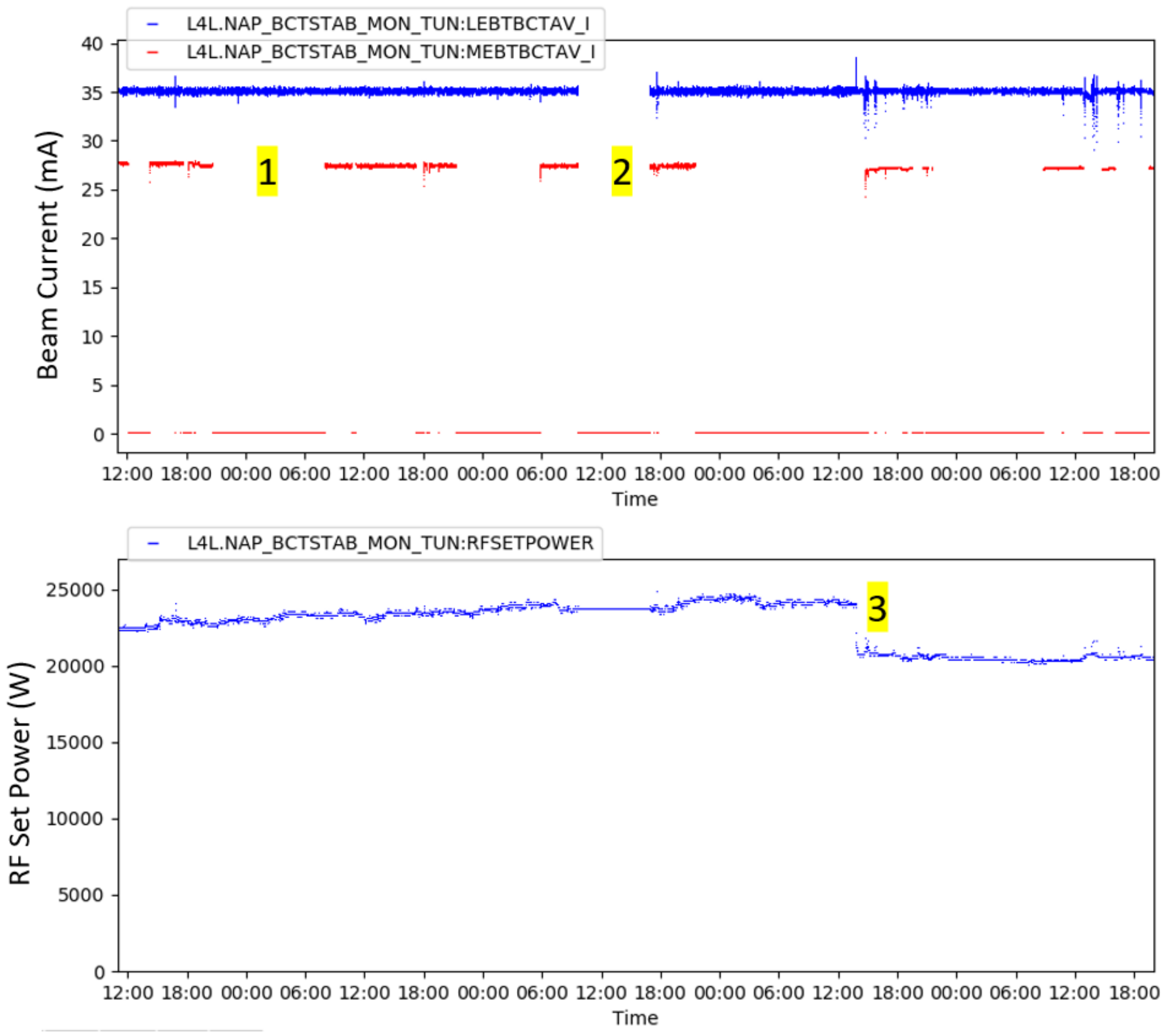


Figure 6. Top: Beam current plot over 5 days (blue: LEBT BCT current, red: MEBT BCT current); Bottom: Source RF Power as set by the Autopilot. Highlighted regions 1: Period without beam through the RFQ, 2: Period with-out beam from the source, the Autopilot stops power regulation, 3: Unstable event with change of source conditions, the Autopilot establishes a new operating power.

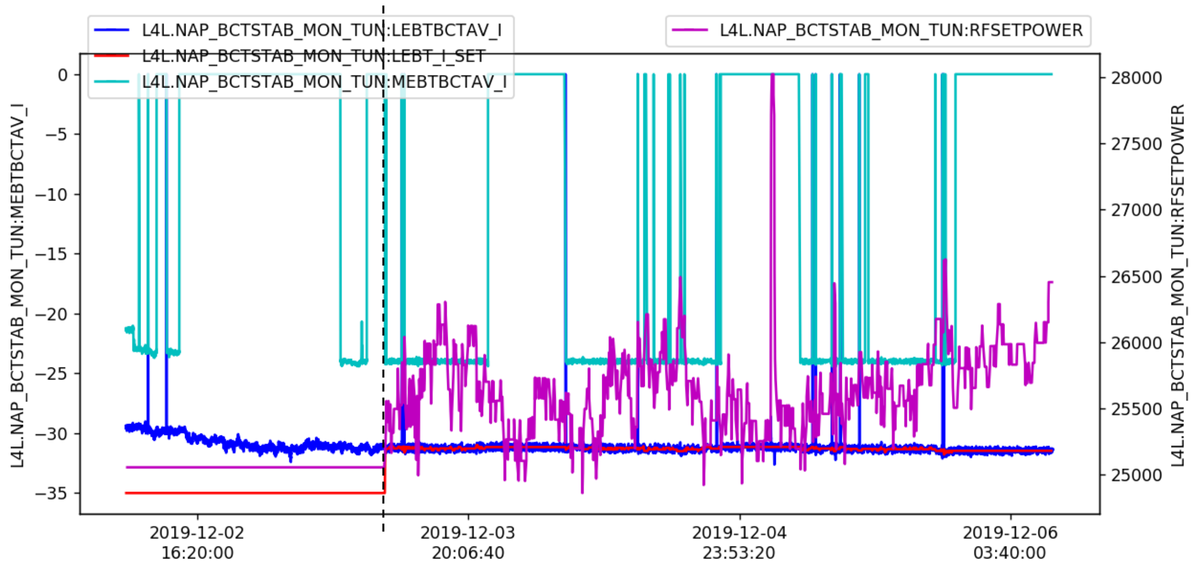


Figure 7. Beam intensity and RF set power from 2 December 2019 to 5 December 2019. Black vertical line shows the time after which the autopilot was started to maintain the MEBT beam intensity constant. Cyan= MEBT measured intensity. Blue=LEBT measured intensity, Orange=LEBT requested intensity, Purple=RF requested power

3.2 Phase Stabilize

As well as the RF power to the ion source, the phase of the RF is also measured. This phase can be used to assess the efficiency of the coupling of the RFQ power to the source plasma. The measured phase is a function of the distance to the resonance frequency for maximum coupling, and can be adjusted with the exact frequency of the 2MHz RF drive.

The Phase Stabilize Task monitors the measured phase over defined windows, and compares them to the set-point. If outside a dead band, the frequency is adjusted along the pulse to return to the phase set-point.

This Task has been tested in 2019 on the Linac4 source while it was not in operation. During that time the source was already operating very stably, and there was negligible enhancement effect from the routine, however it is still valid to continue testing this task.

3.3 HV Reset

Ion sources are typically suffering from sparking events at the order of one or more a day. The source power converters go into a fault state when an over-current is detected, and do not deliver the voltage until reset.

The HV Reset Monitor delivers the present status of the HV converters, as well as counting the number of times that they have gone from ON to FAULT status in the last 24 hours.

The Actor reads the repeated HV converter statuses from the Monitor, and when it moves from ON to FAULT status, it initiates a procedure to restart them.

The following converters are treated by this Task:

- L4L.NFH.014 : EINZEL
- L4L.NFH.015 : SOURCE
- L4L.NFH.016 : PULLER

The three HV converters are physically coupled. Resetting the FAULT state and switching them to ON status requires a relatively complex procedure: commands must be sent to the converters in a specific sequence, and secondary FAULT conditions on restart are common. Therefore, the procedure can still take 1 minute to execute from the FAULT being detected and the converters being returned to their ON state. Nevertheless, this is quicker than the human reaction time to the fault.

The Actor also must be cautious not to restart the converters if they have been deliberately switched off. Therefore, if any of the converters goes directly from ON to OFF, without the Actor making this change, this is considered to be an operator action and the Actor will terminate. After termination, the Actor must be restarted by the operator, only after the converters have been restarted manually.

3.4 eH Ratio

The eH ratio is a Monitor that uses the currents measured on the source high voltage power converters, and the BCT current in the LEBT, in order to calculate an eH ratio (the ratio of electrons to H- from the source).

The eH ratio is calculated as:

e/H = ((*avgDumpCurren*t-DC*offset*) + (*avgSourceCurrent*-DC*offse*t))

/abs(*bctLEBTcurrent*) - 1

The Dump and Source currents are taken from the XenericSampler devices, which in turn come from the OASIS scope signals, averaged from samples in the time window of 275 to 275.6ms. The *bctLEBTcurrent* is calculated from 275 to 275.6ms from the XenericSampler trace of the BCTs.

The standard value that is reported is clipped in the range 0.001 to 200 - which is to allow it to be easily displayed in the Timber web-application using a logarithmic scale (the clipping is mostly to avoid very high e/H ratios that result when there is little beam in the BCT).

A value is also calculated as an average of the e/H over an extended period (currently 1 hour). The average only consists of values that have passed a validation test, similar to the BCT_Stabilize Task, but with an independent list of checks.

The same Monitor calculates a H- Power Efficiency (HPE) (calculated from the Device field *plasmaPowerSamples* in the same time window 275 to 275.6ms).

HPE = *bctLEBTcurrent* / *plasmaPowerSamples*

It would be possible to create an Actor to adjust the caesium oven temperatures to try and maintain a stable e/H ratio, which could be a future evolution.

3.5 Cs Flow

The Linac4 ion source uses caesium to reduce the work function of the plasma electrode, which enhances negative ion production. Caesium is evaporated into the source, under vacuum, from a heated oven.

The amount of caesium that exits the oven can be estimated by scaling of the vapour pressure in the oven to a calibration point. Integrating the flow over time gives a total mass.

A Monitor Task has been added to the Linac4 Source Autopilot to produce this calculation in real-time, and allow the instantaneous flow, and total mass data to be logged over time using the NXCALS system.

The vapour pressure in the oven is calculated from the equation [11]:

P[Pa]= 10.0**(9.171-3830.0/T[K])   : T[K] = *reservoirTemperature* + 273.15

This has been calibrated using quartz microbalance measures as a flow (dm/dt)

dm/dt = c * P[Pa]   : c=1.495 mg.hr^{-1}Pa^{-1} or 4.143*10^{-10} kg.s^{-1}Pa^{-1}

where m is the mass, P the vapour pressure, T the caesium temperature and c the calibration constant. As well as the instantaneous flow, the Monitor will calculate a total mass that has been delivered. When the Task restarts, the Monitor will initiate with the value that is found in the setting field *caesiumMassOffset*. The total mass is only increased if the caesium valve is open, which is read on the Device field *positionValve*. The code assumes that any *reservoirTemperature* above 155C is an error (as this is excluded by the caesium heating system) and if this value is received, it will not add any mass.

To keep the initiation value for the total mass up to date, an Actor updates the value in the setting field *caesiumMassOffse*t every 10 minutes. If the Cs Flow Monitor restarts at any time, it will initiate from this *caesiumMassOffset* value. If it is necessary to zero the mass at any time (for example with a new source) the Monitor Task should be stopped, the setting *caesiumMassOffset* set to 0 via the settings management application, and the Monitor restarted.

4. Summary of Data

The following table links the symbolic names given in the text above to the present device/property#field that is used to acquire the data.

Table 1

Table of symbolic names used in the text. <> signifies an average is taken of the array data from the device/property#field.

Symbolic Name	**Device Property Field**
avgDumpCurrent	L4L.DUMP-I-SA/Samples#samples <>
avgSourceCurrent	L4L.SOURCE-I-SA/Samples#samples <>
DCoffset	L4L.NAP_BCT_Stabilize_Act/EH_Ratio#dumpOffset
SCoffset	L4L.NAP_BCT_Stabilize_Act/EH_Ratio#sourceOffset
bctLEBTcurrent	L4L.BCT.1137/Sampler#sampleData
plasmaPower	L4L.RFLLSRC-PLASMA-SA/Samples#samples <>
caesiumMassOffset	L4L.NAP_BCT_Stabilize_Act/Cs_Flow#caesiumMassOffset
positionValve	L4L.NSRCCSHEAT/Status#csValveStatusDecode
reservoirTemperature	L4L.NSRCCSHEAT/Acquire#reservoirTemperature
plasmaPowerSamples	L4L.RFLLSRC-PLASMA-SA/Samples#samples <>
enableLEBT	L4L.NAP_BCT_Stabilize_Act/BCT_Stabilize#lebtEnable
enableMEBT	L4L.NAP_BCT_Stabilize_Act/BCT_Stabilize#mebtEnable
LEBT-BCT	L4L.BCT.1137/Sampler#sampleData <>
MEBT-BCT	L4L.BCT.31137/Sampler#sampleData <>

5. Summary

Within the Linac4 Source Autopilot project, a framework has been developed to allow users to develop code in Python that runs on a stable server, and interfaces to several accelerator controls components to allow monitoring, settings, diagnostics and testing. The framework is also in use for the Linac3 source, and can be further used for automating other tasks. It can also be envisaged as a stable location where to run common script type tasks extracting data from the NXCALS logging system and using LSA for settings management.

Within this framework, tasks have been developed to continuously adjust the source RF power, and perform automatic resets of the High Voltage converters. The stability of the source has improved remarkably over this period (also when coupled to other improvements like continuous caesiation).

The authors would like to thank the Linac4 ion source team in BE-ABP for providing the motivation for this project and testing time, as well as to the software teams within BE-CCS for their work to make this project possible, in particular the UCAP, ACC-PY and LASER teams.

References

[1] G. Voulgarakis, J. Lettry, S. Mattei, B. Lefort, and V. J. Correia Costa, Autopilot regulation for the Linac4 H- ion source, AIP Conference Proceedings 1869, 030012 (2017), https://doi.org/10.1063/1.4995732

[2] B. Lefort, M. Giordano Ferrari, R. Billen (Ed), CERN Beams Newsletter, December 2013, p4, https://cds.cern.ch/record/2243231/files/BE_Newsletter_009.pdf

[3] https://wikis.cern.ch/display/ACCPY/Accelerating+Python+Home

[4] https://wikis.cern.ch/display/MW/RBAC

[5] https://gitlab.cern.ch/

[6] Lajos Cseppentő et al, L. Jensen (Ed), CERN Beams Newsletter, July 2020 https://cds.cern.ch/record/2724560/files/BE-newsletter_2020_edition33.pdf

[7] https://acc-py.web.cern.ch/gitlab/acc-co/ucap/ucap-python-common/docs/stable/

[8] https://gitlab.cern.ch/acc-co/autopilot/nap-utils

[9] https://logbook.cern.ch/elogbook-server#/

[10] https://wikis.cern.ch/display/LISRC/L4+Source+Autopilot

[11] C.B. Alcock, V.P. Itkin and M.K. Horrigan, Canadian Metallurgical Quarterly, 23, 309, (1984)