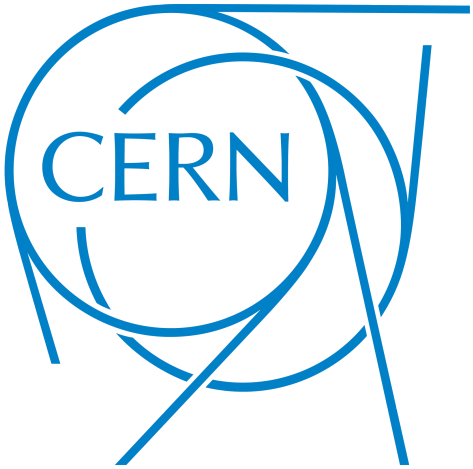
Improvements on the DCS and monitoring system for the ITk Pixel Loaded Local supports

CERN Summer Student Project



Author: **Radek Jirásek**[1]
Supervisor: Hans Ludwig Joos[2,4],
Benedikt Vormwald[2,3]

[1] Institute of Particle and Nuclear Physics, Faculty of Mathematics and Physics, Charles University, Prague
[2] CERN, Physics Department
[3] Institut für Experimentalphysik, University of Hamburg, Germany
[4] Faculty of Physics, Georg-August-Universität, Göttingen

Department: EP-ADE-TK
September 24, 2024

Acknowledgements

I would like to thank my supervisors Hans Ludwig Joos and Benedikt Vormwald for supervising me over the course of the Summer Student program. Especially thanks to Hans for his kind and helpful approach during every day of our work. I would like to thank CERN and the Czech particle physics community for funding this program and giving me an opportunity to participate.

Abstract

This report summarizes the main accomplishments of my CERN Summer student program. One of the tasks was to write documentation for systems that I created. Parts of these documentations are contained in the report.

Work was done on the Loaded local support (LLS) of the Inner tracker (ITk) of the ATLAS detector which will be part of the High Luminosity upgrade of LHC (HL-LHC) in three main separate subprojects. The first chapter examines a database and developed system in the Grafana web application to display data from detector tests and also a complex system for alerts. Further emphasis is placed on the provisioning of this system to any other institutes in the collaboration. In the second part, the report shows work on the detector control system (DCS) for Monitoring if Pixel system (MOPS). A newly created GUI for control over this part of DCS is shown in the second chapter. In the last chapter, the report provides a description of the way how to integrate the interlock logic of FPGA to DCS GUI where we wanted to display the Interlock matrix.

Keywords

ATLAS, ITk, Grafana, DCS, MOPS, WinCC OA, Interlock

Contents

Introduction

ATLAS is one of four detectors on LHC, the largest current particle accelerator worldwide. We will talk about the upgrade of the Inner tracker (ITk) of the ATLAS detector for the future phase of HL-LHC (see Fig. 1). Detectors would need to be more radiation-hardened to withstand at least 5 times higher intensity of particle collisions. The inner tracker would consist entirely of silicon sensors. Strip sensors in the outer part and the pixel sensors in the inner part. We are working with the pixel part of the ITk, denoted by the green and blue-red parts on the right side of Fig. 1.

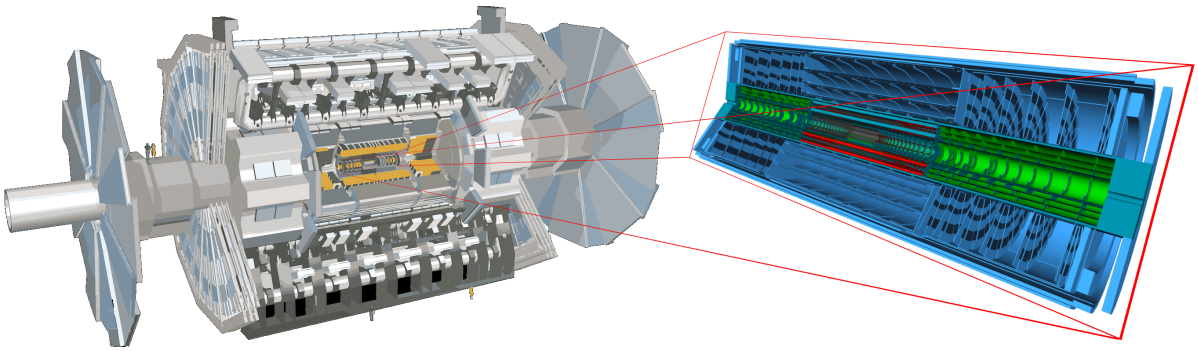


Figure 1: Schematic of current ATLAS detector on the left with a schematic plan of new Inner tracker on the right side. [1]

The individual part of the pixel system of ITk is shown in Fig. 2.

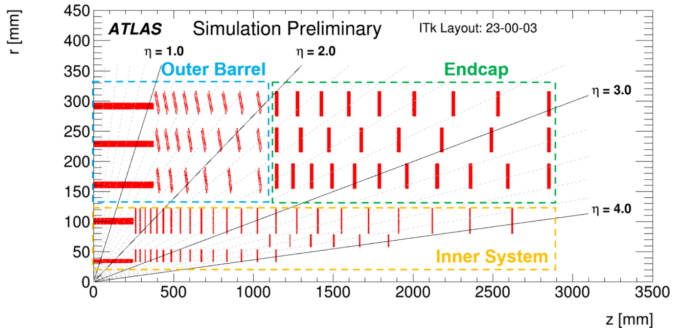


Figure 2: Schematic of the layout of the future ATLAS ITk pixel detector. [2]

Our group is working with the Loaded Local Support (LLS) or "Outer Barrel" part denoted by blue color. LLS holds detectors in their precisely defined places and provides support for cooling, power supply, and readout cables. Outer Barrel LLS consists of two types of structures holding pixel detectors. One is "Longeron" and it is located in the part closer to the interaction point (IP). The second one "Inclined half ring" is placed in further from the IP. We can see their design in Fig. A.1 and A.2 in the appendix A.

Chapter 1

Grafana visualization of data from the DCS project for OB LLSs

1.1 Quick introduction to Grafana

Grafana is an analytics and interactive visualization web application. It can produce charts, graphs, and alerts. Grafana can query data from a database, and then display them on the chart or in a table, etc. Documentation can be found in [3].

We are using Grafana 11.1.1, if you are using a different version, things could be different.

The main part of Grafana are dashboards. We can move between dashboards through the menu on the left-hand side or by clicking on the folder (in our case: "Operator dashboards") of dashboards (if we are in one of the dashboards) and choose the required one. It can be seen in Fig. 1.1. The dashboard can be divided into more sections called rows. They are horizontal separation lines, which can be packed with panels below it. It is convenient to use it to have a clearer view of the dashboard. With rows, we can hide all sections that we are not currently interested in.

Key elements of any dashboard are panels. Panels represent various possibilities of visualization. One can choose from many types of graphs. The most often used is "Time series" which allows us to monitor specific values over time. Grafana offers many more types of graphs, such as gauge monitors, histograms, pie charts, etc... Besides that, panels with text array or canvas can be used to create graphical objects for our dashboard. More about how to create and edit the panels in chapter 1.6.1.

Grafana can use data from various types of sources. The most common is to query data from a database. These data can be used not only as points in our graphs but also for example as parameters changing the colors of our objects in panels or as an input to alerts.

One of the very useful features of Grafana are "variables". These variables are defined for specific dashboards and are displayed at the top of the dashboard. Users can change the values of these variables and then it could be used by dashboard panels anywhere. They are used as parameters for specific types of Longeron (or Inclined half ring) and then the query path is derived according to these variables. That means you can change which variables are plotted in the panels with just one click at the top of the dashboard. One can see an example with variables in the "Inclined half ring" dashboard in the Fig. 1.1.

You can change the time range in which all panels are plotted. This could be done in the top right part of the dashboard and choose the range that we like (see Fig. 1.1). Another way is to drag over the time region in any of the graphs depending on time. If we want to update data in that defined time range (for example "Last 2 hours"), we will click on the button with the refresh icon on the right top part of the dashboard next to the time range.

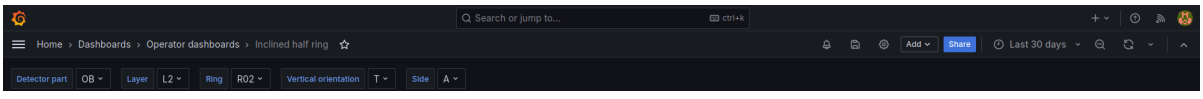


Figure 1.1: Example of top sheet in dashboard "Inclined half ring" with settings buttons and template variables.

1.2 Influx Database

The database is used to store data from the Detector control system (DCS). The DCS specifically uses InfluxDB which is an open-source time series database. It is used for storage and retrieval of time series data in fields, for which it is also more optimized. Documentation can be found here.

1.2.1 Structure of data in our database

Data in specific "measurement" is stored in the database in rows. Every row contains a time stamp and then a set of fields and tags. Fields contain values which are not indexed and should be used for storing changing data. Tags are indexed values meant for discrete sets of values. That means that commands such as WHERE are much faster at looking for specific tag than for specific field values.

In our case, we have two main tags: "hardware name" and "alias". Hardware name (data point element in WinCC OA) is the name reflecting the sensor that measured that value. Alias is a logical name showing what that value represents. For querying we use aliases. Additionally, the same sensor could actually have more aliases due to more logical roles[1] in the FSM (Finite state machine).

Then, we have a separate field for different data types of values such as "original_value_float", "original_value_int", etc[2]. That means that we have measurements of temperature and pressure in the same "column". These variables differ in the tag[3] which specifies what kind of data it is. Besides the actual data in the database, we also store values telling us comments, and other metadata.

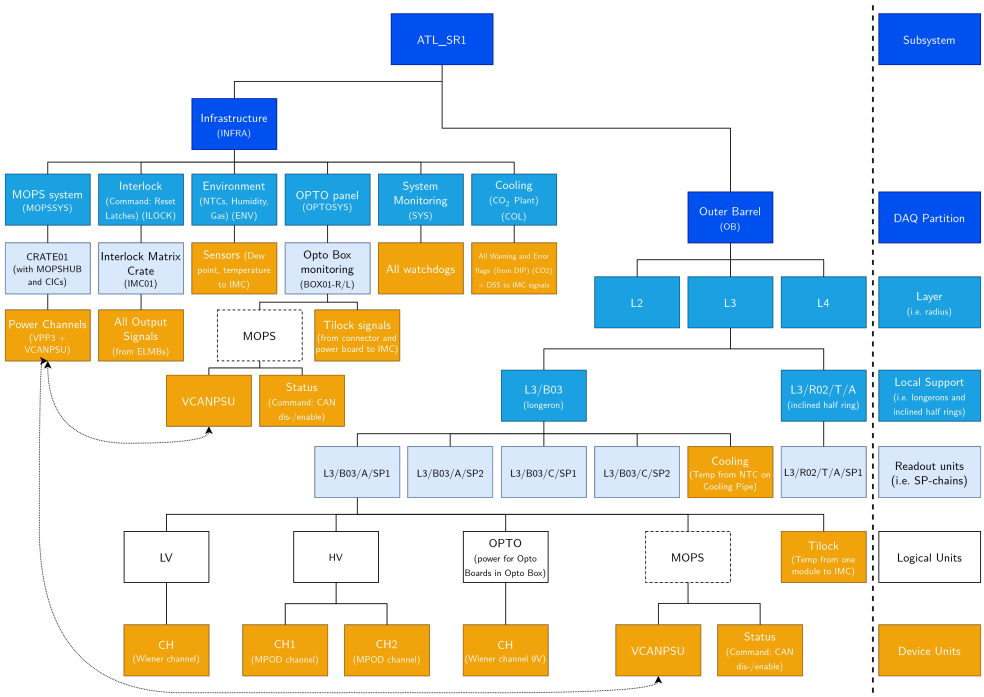


Figure 1.2: Visualization of FSM (Finite state machine) hierarchy for the Detector control system of ATLAS ITk Outer barrel.

[1]The values are then saved in the database as "{first_alias|second_alias|...}". That is also a reason to use regular expressions to query data!

[2]This is due to the impossibility in InfuxDB to have different data types in the same "column".

[3]It could be tag::name for example "`ATLITPLLSDCS:Wiener/LLSQC_MPOD01/Board1/Channel00.MeasurementCurrent`" or tag::alias, for example "`!OB/L3/B01/A/SP1/HV/CH1.Imon`"

As we can see from "`/$Part/$Layer/$Longeron/A/SP1/HV/CH1.Imon/`" in the code example, aliases are built in a logical way to represent a position in the hierarchy tree shown in Fig. 1.2.

1.2.2 Quick overview and usage in Grafana

There is a lot to cover about InfluxDB, but it is not important to understand how Grafana works. The main emphasis is placed on examples of how database commands are used in Grafana directly to query the data. I will describe the command structure using the following example of a query:

```
SELECT mean("original_value_float") FROM "INFLUXDB_EVENT"."INFLUXDB_EVENT" WHERE
    ("alias"::tag =~ /INFRA\/OPTOSYS\/$Box\/PWRBOARD\/Tmon/ AND "alias"::tag !~
    /Tmon2/) AND $timeFilter GROUP BY time($__interval) fill(null)
```

The SELECT clause specifies which values we want to query, here it is a mean value of the field "original_value_float". The mean is evaluated by a time interval specified by the command GROUP BY which is given by the Grafana argument "__interval" which is a variable automatically calculated according to your choice of time range plotted in the graph.

The FROM clause specifies from which part of the database you want to query. The selection is in the form "Retention_policy"."measurement_name". The first argument describes how long InfluxDB keeps data, it could have a default value. The second specifies a so-called "measurement". We can have separate measurements in one database with different variables in it.

The WHERE clause allows us to give conditions on the data that is being queried. Here we demand that individual data points have a tag with the name "alias" which contains (that is "=~", strict equality would be just "=" and without the usage of "/" before and after the string[4]) the specific string in "/string/". The example command also has a second condition to not contain in its alias the string "Tmon2".

The last condition is to query just data in our time range specified by Grafana in the variable "timeFilter".

The last command "fill" could change the value reported for time intervals that have no data. Here we report a null value in this case.

Another useful command (probably more for just exploring the database) is
"`LIMIT <N1> SLIMIT <N2>`". Number "N1" specifies the number of points to return per measurement and "N2" specifies the number of series to return from the specified measurement.

Here is a second example from the Longeron dashboard:

```
SELECT mean("original_value_float") * 1000000000 FROM
    "INFLUXDB_EVENT"."INFLUXDB_EVENT" WHERE ("alias"::tag =~
    /$Part\/$Layer\/$Longeron\/A\/SP1\/HV\/CH1.Imon/) AND $timeFilter GROUP BY
    time($__interval) fill(null)
```

There are two main differences. We can use basic mathematical operations to change the data. Here we change the unit from A to nA by multiplying the value by 10^9. The second point is how we assign variables to the aliases. For example "$Layer" uses the value of the "Layer" variable in the string that is wanted.

In the code attached below, we are showing the last example of a query command, where we plot all module temperature data regardless of their specific module number, but stressing that data is not coming from the MOPSHUB4B project by using a condition for the data point "name" tag (this will only return the interlock module temperature).

```
SELECT mean("original_value_float") FROM "INFLUXDB_EVENT"."INFLUXDB_EVENT" WHERE
    ("alias"::tag =~ /$Part\/$Layer\/$Longeron\/A\/SP1/ AND "alias"::tag =~ /Tmon/
    AND "name"::tag !~ /ATLITPMOPSHUB4B/) AND $timeFilter GROUP BY
    time($__interval) fill(null)
```

[4]InfluxDB support regular expressions. It must be surrounded by / characters and use Golang's regular expression syntax [4].

1.3 Setup tutorial of Grafana

Setting up a Grafana is provided by a docker container. The process of installation is described here and contains a few basic steps (detailed description in the repository above):

1. Prerequisites: We need to install docker itself and set it up. Installation steps for the operation system of type "ALMALinux 9" are described in the repository. If you use a different type of operating system, google how to install docker for your system.
2. Git: download the repository from git to your local server/pc via standard ways (for example git clone).
3. Define variables in the .env file by following the description:
   (a) The environment variables in the .env file are used to create an InfluxDB user named llsqc1user, an InfluxDB database named llsqc1db, and a Grafana user named admin. Change the "INFLUXDB_ADMIN_PASSWORD" and "GF_SECURITY_ADMIN_PASSWORD" environment variables in the .env file to values of your choice. These are the passwords for the InlfuxDB and Grafana users.
   (b) Then it is important to change the other two variables in the .env file for the proper working of the alerting system of Grafana. It will send a notification to the Mattermost channel if any of the specified thresholds would be breached. The first of them is "GF_SERVER_DOMAIN", we need to insert here our public Grafana URL[5]. This would be later used in the alert messages. The second variable that we need to specify is "GF_ALERT_URL", which sets the path to the Mattermost channel.
      i. In the Mattermost: For that we first need to create a webhook in our channel. It is very convenient to create a new channel in Mattermost for these notifications from Grafana.
      ii. In this channel we choose settings in the top left corner (nine dots in the shape of a square) and then select "Integrations" ⇒ "Incoming Webhooks" ⇒ "New incoming webhook" and choose name, channel and other settings which we want. Mattermost then generates a special URL, which we insert into the .env variable "GF_ALERT_URL".
      iii. After restarting docker, Grafana should have configured the Contact point to your Mattermost channel. You can test the function of that in the main menu of Grafana ⇒ Alerting ⇒ Contact Point ⇒ then choose "edit" (little pencil) on of the Contact points called Mattermost (all of them should be mounted to your channel), and press "Test".
   (c) Then you could also configure other variables in the .env file that are connected to alerts (their names start with "ALERTS_"), more about them in chapter 1.5. These variables serve to define which exact variables would be used in the queries and descriptions in alerts. The reason for such a solution is that template variables in dashboards can not be used in the alert rules.

- Usage of docker: We can start the containers by running the following command from the directory that contains the git repository:

```
$ docker compose up -d
```

More information is described in the README of the mentioned repository.

1.4 Structure of dashboards

We split the panels into three logically separated dashboards. First, Interlock & environment monitoring is the most important dashboard for monitoring current values and checking if everything is right. The next two dashboards show data directly connected with modules in Longerons or in Inclined Half Rings. One can switch between these dashboards using the hamburger menu on the left and choosing the corresponding dashboard.

The system is prepared ahead of time when there is data in the database even if just for a few variables. Dashboards and all queries are configured for future aliases which are well known now even if there is no data yet. At the moment when new data will be stored in the database, Grafana will automatically display them in the proper panels.

[5]Ours is for example: 'itkpix-llsqc1-dcs-01.cern.ch', but you need to use your own.

1.4.1 Interlock & environment monitoring

In the dashboard with Interlock & environment monitoring there are two rows. The first one shows panels directly connected to the interlock system. We can see the layout in Fig. 1.3. The panel on the top left-hand side shows all current alerts that are in the state of triggering (condition was breached, sending alerts), pending (condition was breached, but still waiting to send alert), or error state. It could be useful as a quick orientation between all things that happened after a potential accident or any other event. The first two panels in the bottom left part are dedicated to the Dew Point measurement, one of the most important variables. The left one shows the time dependence of the Dew point. On the right side, we show the current value in the gauge monitor. This type of layout is very convenient, especially for interlock variables where we are interested in the last values and the current value. In the middle of the section are four panels. The first two show temperatures in the Optoboards, specifically Powerboard (PWRBoard) and Connectorboard (CONBoard) temperatures.

The second row shows the amount of light in the box. This is very important to be sure that the box is properly closed and that no light is coming inside.

On the right-hand side, there are logic values (0 or 1) for emergency status and two statuses of MARTA (cooling apparatus) at the top. Below that are the states of each switch of the testing box. If the switch is closed, it has a value of 1 (green), if it is open then 0 (red). There is also a time history of states below these logic color-coded indicators.

Figure 1.3: Screenshot of Interlock section in the "Interlock and environment monitoring dashboard". Most of the panels are missing data because at the time of writing this documentation the sensors are not yet implemented in the DCS setup (With data in DB, the purple rectangle would look like separate squares with red or green color depending on the state from queries).

The second row is dedicated to environment monitoring. Every environmental parameter that is monitored but not evaluated for an interlock decision falls into that category. The whole row is logically divided into several parts. The first of them contains two panels dedicated to the Optoboards. The first panel shows time dependence, and the second shows actual values in the gauge monitors. In addition to the values for the temperatures of the PWRBoard and CONBoard which were also displayed before in the interlock section, there are temperatures of all individual, connected Optoboards.

The next part contains panels measuring the properties of air. The first one on the left-hand side is again a dew point. On the top right-hand side, there is a Dry air flow measurement plotting flow in [L/min] of dry air into the test box. Below is a relative humidity measurement of two separate sensors.

The third part is dedicated to other temperature measurements. On the left-hand side, there are two temperatures of the test box (one at the top of the box, the second at the bottom). The right-hand side shows NTC temperatures which are arbitrary temperature sensors that could be placed anywhere according to the current need.

The last part shows variables connected to the CO_2 cooling. On the left side, there are temperatures at the top and pressure at the bottom. A panel on the right-hand side is showing the pressure of CO_2 normalized to its temperature. This is very handy if we want to examine if there is a leakage in the system. The following statement is valid only in a system with closed pipes. In the approximation of an ideal gas in the system and constant volume, $\frac{p}{T} = \text{const.}$. This immediately follows from the equation of an ideal gas: $pV = nRT$. Where p is the pressure of the gas, V is the volume of the system, T is the absolute temperature of the gas, n is the amount of substance of the gas (in moles) and R is the gas constant.

1.4.2 Longeron & Inclined half ring

In this section, we will describe the general structure of rows in the "Longeron" dashboard. The dashboard for the "Inclined half ring" is almost identical. In both dashboards, there is the first-row showing interlock temperatures of SP chains on Longerons or Inclined half rings. The row contains two panels both with temperatures of one module per SP chain. The left panel shows time dependence and the right panel shows current values.

Figure 1.4: Screenshot of one SP chain section of the "Longeron" dashboard. Displayed is some fake random data to show what it could look like.

Every following row in the dashboard is dedicated to the separate SP chains of the LLS. In each of these rows are the following panels: On the top left side is a panel showing the time dependence of temperatures of all modules in that SP chain. On the right-hand side is a panel showing the voltage drop of each module connected to the SP chain. SP1 has 6 modules and SP2 has 12 modules in the case of a Longeron. Inclined half rings could have up to 13 modules per SP chain. Below these two panels are three sections separated by titles. There are two panels with electric current and two with voltage. On the left side is always voltage, and on the right side is electric current. The first of two panels is a time-dependent plot, the second one shows the current value in the gauge monitor. The first section is dedicated to the Low-voltage channel. The second and third are the two High-voltage channels.

1.5 Alerts

The alert system in Grafana is a very convenient way how to get quick notifications if something is not going how you want. It is very important that the operator of the setup can receive a message and take action before

the interlock of the system activates. The message that the interlock has been activated is also very helpful to fix the setup and start new measurements as soon as possible.

We create a sophisticated system of alert rules to get information about all important changes in DCS. These alert rules are listed in the table 1.1 below. Apart from alert rules, there are other things in the alert system. Contact points are ways how and where to send messages. We create additional contact points to send messages to a Mattermost channel (tutorial in the section 1.3) which differ in which templates are being used. Templates are scripts for formatting the text of messages. Examples with descriptions of these messages can be found in section 1.6.2. Besides templates, we define a notification policy which is a structure of labels characterizing how, where, and when alert rules and their notification alert messages should be sent. An example of these labels is "timing" also shown in table 1.1 which specifies the time when an alert would be sent to the contact point. With these labels, we could quickly edit the properties of a whole group of alerts. Labels also provide options to group alerts together. We are using this grouping with the label "severity" = "informative" or "ILOCK". If alert rules are breached in a defined time from each other for alerts of the same severity, they would be grouped together and sent to the contact point as one alert. If we program our templates in a proper way, we can show all of them in one message.

Name	**Variable**	**ILOCK**	**Timing**	**Informative Threshold**	**ILOCK Threshold**
PWRBoard1 Temperature	temp	Yes	always	**[-54, 27] °C**	**>30 °C**
CONBoard Temperature	temp	Yes	always	**[-54, 27] °C**	**>30 °C**
Optoboard E/D2 Temperature	temp	No	work	**[-54, 35] °C**	
Optoboard D/D3 Temperature	temp	No	work	**[-54, 35] °C**	
Optoboard D/D4 Temperature	temp	No	work	**[-54, 35] °C**	
Optoboard C/D5 Temperature	temp	No	work	**[-54, 35] °C**	
Optoboard C/D6 Temperature	temp	No	work	**[-54, 35] °C**	
Optoboard B/D7 Temperature	temp	No	work	**[-54, 35] °C**	
Dew Point	dewpoint	Yes	always	**>-58 °C**	**>-55 °C**
Longeron A/SP1 Temperature	temp	Yes	always	**[-54, 35] °C**	**>40 °C**
Longeron A/SP2 Temperature	temp	Yes	always	**[-54, 35] °C**	**>40 °C**
Longeron C/SP1 Temperature	temp	Yes	always	**[-54, 35] °C**	**>40 °C**
Longeron C/SP2 Temperature	temp	Yes	always	**[-54, 35] °C**	**>40 °C**
IHR SP1 Temperature	temp	Yes	always	**[-54, 35] °C**	**>40 °C**
IHR SP2 Temperature	temp	Yes	always	**[-54, 35] °C**	**>40 °C**
Light sensor TOP	boolean	Yes	always		**false**
Light sensor DOWN	boolean	Yes	always		**false**
Box switch 1	boolean	Yes	always		**false**
Box switch 2	boolean	Yes	always		**false**
Box switch 3	boolean	Yes	always		**false**
Box switch 4	boolean	Yes	always		**false**
Emergency status	boolean	Yes	always		**false**
CO2 plant stopped status	boolean	No	always	**false**	
CO2 plant warm status	boolean	No	always	**false**	
Box temperature TOP	temp	No	work	**>30 °C**	
Box temperature BOTTOM	temp	No	work	**>30 °C**	

Table 1.1: Table of all alert rules. ILOCK parameter indicates if that variable goes to interlock (in notification policy same as label severity: ILOCK = Yes, informative = No). The timing parameter specifies when a user can get notifications, work = from 8 am to 6 pm except weekends and holidays, always=always. If the threshold value is in the "[ ]", that means the system will send a notification if a value is outside of this range.

The logic structure of the Notification policy and its labels can be seen in Fig. 1.5. It might seem that a better structure would be to have a severity or timing as a first and then ILOCK_Active in the end. The problem is that labels specifying grouping must be last. If we want to group notifications together by severity, we need such a logical structure.

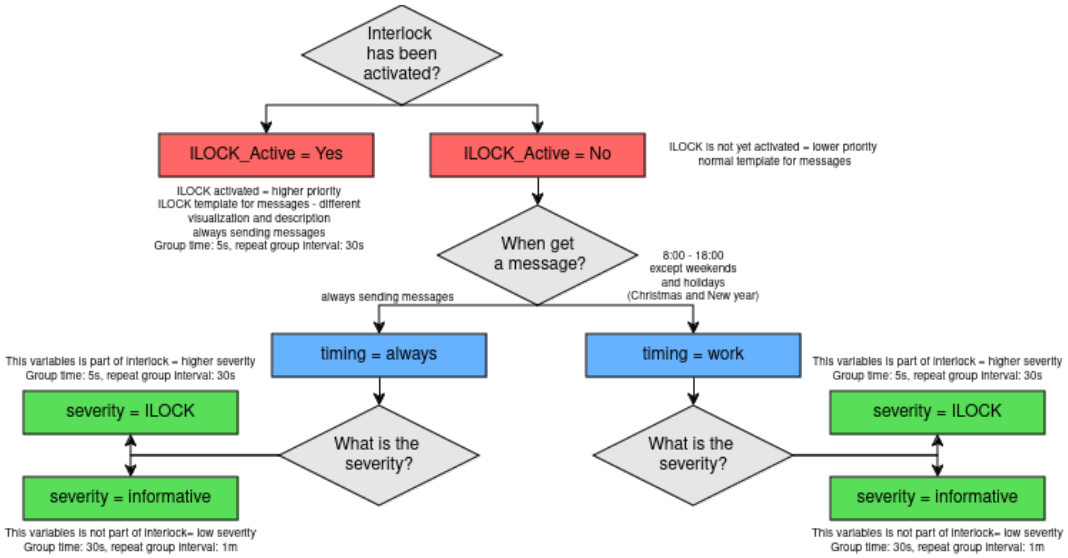


Figure 1.5: Shematic diagram of the logic structure of the notification policy in Grafana alert system.

We are using custom templates creating messages directly designed to our needs. The description of the code is in section 1.6.2. We are showing examples of how messages in the Mattermost channel look like in Fig 1.6 and 1.7 below. We also send notifications when the problem was resolved. The message contains a list of labels which is helpful to identify the exact source of the problem. Below the labels there is a unique summary and description for quick understanding of what is happening and to get an idea of what the operator should do next. We can copy this summary and description (for logging purposes for example) just by clicking to icon on the right (will appear if we hover with the mouse over it). Then there is a link to the panel that shows that variable and in the case of "Firing alert(s)" there is also a URL to silence this alert. In the other screenshot in Fig 1.7, we can see what it looks like when several alerts are grouped and sent together to the contact point.

If you see links with "change_var_in_.env_file_GF_SERVER_DOMAIN" in the messages, it means that you have not configured the server domain in the .env file.

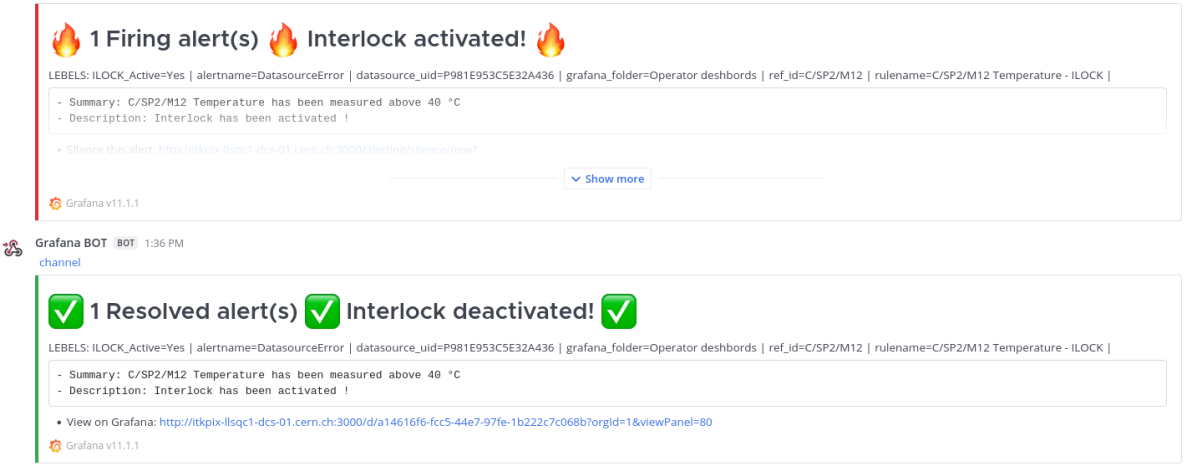


Figure 1.6: Example of alert messages in the Mattermost channel showing design with ILOCK_Active = Yes.

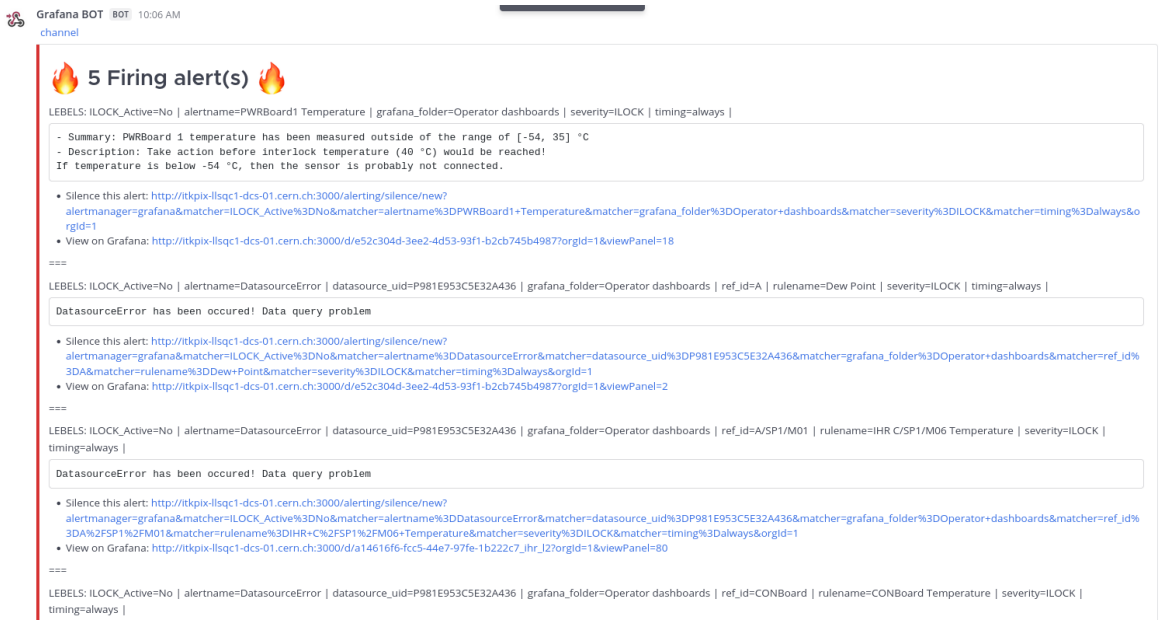


Figure 1.7: Example of alert messages in the Mattermost channel showing several alerts grouped in one message.

1.5.1 Update of alerts

The last part of provisioning is the `alert_update.py` script. This script changes files in the grafana_provisioning folder to update queries, labels, and other settings to reflect newly changed variables in the .env file. There is only one set of alerts to one exact component of the detector. For example longeron in layer 3, number 7 in the outer barrel. When we change template variables in the dashboard to see other ones, alerts are not updated according to it, they have no access to these variables. This script does the following:

1. Update queries in the alert rules for new variables (alert_rules.yaml file)
2. Change properly summary and description labels if it is necessary (alert_rules.yaml file)
3. Change default values of template variables in all three dashboard .json files in the dashboard folder

Let us say, that we changed our setup to test longeron configure as “OB”, “L3” and “B04” from previously used “OB”, “L2” and “B01”. We need to open the .env file in the provided docker folder. Then change ALERTS_LONGERON_LAYER=L3 and ALERTS_LONGERON_LONGERON=B04 and save it. Then you execute the script by command:

```
$ python3 alert_update.py
```

The program also stops the currently running docker container and after applying the changes starts the containers again in the background. The prerequisite for this procedure is to install python3 in your system. You can check if you have installed it with the command:

```
$ python3 --version
```

which should return the version of your installed Python software.

1.6 Project editing

There is a difference in the way how to edit panels or any other things in Grafana when we have our own project or if we get it as a provided project.

1. In the case of own project and permissions to edit (admin): We can change anything directly in the Grafana GUI in the web application. Changes could be saved and propagated to the system.
2. In the case of the provided project: We can sometimes change things (panels) but can't save them. Other things can not even be edited like alerts. There is a way how to do it, but it is a little bit more complicated.

We will discuss both cases in the next sections. First, we will talk about panels, and how to edit them and we also provide a few useful tips.

1.6.1 Editing of panels

In the first case of our own project, we need to be logged in as admin in Grafana to have permissions to change panels or alerts. To edit a panel, we need to click on the three dots in the top right corner of the panel to open a menu and then choose the option "edit". Then we can see a preview of the panel in the top right corner. Below that are the settings for querying data. Here we can change the URL to the database or completely change the query command. If we want to change the data structure we can use the option to "Transform" the data in the top part of this section with the query details. With this option, we can for example merge two columns of the database into one. If we want to use just basic mathematical operations between queries, we can use the "Expression" option at the bottom of the query segment. We can choose between math operations, conditions, threshold evaluations, or other options. We can also directly edit or add alerts connected to this panel. For that, we need to choose "Alert" next to "Query" and "Transform". To save changes we need to click the "Save" button in the top right corner (or "Apply" and then save the whole dashboard).

If we edit panels imported by provisioning as described in chapter 1.3, we cannot save changes directly into the dashboard. Instead, Grafana will offer an option to save a JSON file. This JSON file we can exchange with the one in the following subfolder:

```
influxdb-and-grafana-for-lls-qc-dcs/
/grafana_provisioning/dashboards/Operator dashboards
```

This subfolder is mounted into the docker container and when the container is started, panels in the form of JSON files from this folder are automatically created.

However, when we run the alert_update.py script, the program will overwrite our changes back to the default ones. If we want to have permanent changes (even when we have used the Python script) in the dashboard, we need to make this change also in the backup files. There are ".<dashboard_name>.json" files in the backup folder (Example: ".Longeron.json"). These files have the same structure as the original dashboards but have parameters inserted in them[6]. You should NOT rewrite these parts of the code, otherwise the script will not work. To change things, for example, a panel, you need to find the corresponding part of the code in the JSON file and change only that part without touching the rest of the code. It is a good idea to make some backups before intervention. An example of what we could easily change is the default time range. It is one of the last parameters in the JSON file:

```
},
"time": {
  "from": "now-30d",
  "to": "now"
},
```

We can for example change the range to the last two days just by replacing "now-30d" with "now-2d".

In the following lines, we focus more on a few useful **tips and tricks** than a general description of how to change everything in a panel.

- We want to write a description of units to the axis label. A better way than writing the label of the axis in the panel directly is to set units in the "Standard options". Grafana will show labels in the panel and also recalculate automatically units to other orders. For example when querying 0.007 A from the database, Grafana would display it as 7 mA.
- You have a panel with more data entries and want to change properties of only one, or change properties of all but always to different values. Then use overwrites. Specify which query you want to edit (by

[6]Like ALERTS_LONGERON_DETPART which are used to insert real values from the .env file

name, field, value, ...) and choose the property you want to adjust. We are using this regularly for changing the labels of queries.

- If you want to make a more sophisticated indicator, you can use a canvas and create objects like rectangles on it. They can even have colors depending on the query output.

1.6.2 Alerts editing

When we want to edit alerts in our own project, it is not complicated and well described in the Grafana documentation [3]. We will focus on the other case in this chapter. In the case of a provided project, alerts cannot even be edited. We need to change the scripts directly. The only partial exception are alert rules.

Alert rules

When you click on "more" on the right-hand side of the alert rule, then hover over the "Export" button, and click on the "with modification" button that appears, it will open an edit window exactly the same as in the case of editing your own project (you can follow the Grafana documentation [3]). Then you can click on the export button and paste the code in the file: "grafana_provisioning\alerting\alert_rules.yaml". Such a change only persists until one runs the alert_update.py script. To make the change permanent, you need to make this change also in the file "backup\.alert_rules.yaml". You must NOT change anything besides the small section of code. If you change the section with $ variables, the program will not work anymore. We recommend making only small changes such as a change of summary, description in the alert rule, or change in the time range, etc...

Contact points

To edit where and how messages are sent, we need to change the file "grafana_provisioning\alerting\contact_points.yaml" directly. We can change the name of the bot in the chat for example, or templates used for messages. Be aware not to change $\$GF\_ALERT\_URL$, otherwise the Mattermost webhook URL from the .env file would not be used anymore.

Templates

To edit how notification messages look, we need to change the file "grafana_provisioning\alerting\templates.yaml" directly. In case you want to change it, be aware that it is very sensitive to any whitespaces or tabs. Therefore, be careful with manipulations and make a backup beforehand.

We are showing one example of the template script "Mattermost.ILOCK_yes.text" below:

```
{{ define "Mattermost.ILOCK_yes.text" -}}
{{ if .Alerts.Firing -}}
{{ len .Alerts.Firing }} Firing alert(s)  Interlock activated!
{{ template "alerts.ilock_yes" .Alerts.Firing }}
{{- end }}
{{- if .Alerts.Resolved -}}
{{ len .Alerts.Resolved }} Resolved alert(s)  Interlock deactivated!
{{ template "alerts.ilock_yes" .Alerts.Resolved }}
{{- end }}
{{- end }}

{{ define "alerts.ilock_yes" -}}
{{ range . -}}

===

LABELS: {{ range $k, $v := .Labels }}{{ $k }}={{ $v }} | {{ end }}
~~~{{ $error := false }}{{ range $k, $v := .Labels -}}{{- if eq $k "alertname"}}{{-
    if eq $v "DatasourceError"}}
Datasource Error has been occured! Data query problem {{ $error = true }}{{- end
    }}{{- if eq $v "DatasourceNoData"}}
No data in the database in the last two hours{{ $error = true }}{{- end }}{{- end
    }}{{ end }}{{- if eq $error false}}
{{- if index .Annotations "summary" }}
- Summary: {{ index .Annotations "summary" }}{{ end }}
```

```
{{- if index .Annotations "description" }}
- Description: {{ index .Annotations "description" }}{{ end }}
{{- end}}
~~~

{{- if eq .Status "firing" }}
- Silence this alert: {{ reReplaceAll "localhost:(.*)"
    "change_var_in_.env_file_GF_SERVER_DOMAIN:$1" .SilenceURL }}{{ end }}
- View on Grafana: {{ reReplaceAll "localhost:(.*)"
    "change_var_in_.env_file_GF_SERVER_DOMAIN:$1" .PanelURL }}

{{ end }}
{{ end }}
```

We will summarize a few comments about the template script code (Go's templating language [5]) above to get an idea of how it works:

Every command is written in double curly brackets "{{ }}". Segments of code, for example in the "if" statement, must start with the command "{{ if ... }}" and then end with "{{ end }}". You can use more commands such as "define" to define for example templates which you can use many times later. First, at the beginning of the file, define the file name of the whole script. This definition is mandatory.

There are a few other Grafana commands such as as "reReplaceAll ⟨str1⟩ ⟨str2⟩ ⟨str3⟩" replacing str1 with str2 in str3. A whole list of commands can be found in the Grafana documentation.

Then we can use "." objects. They inherit all objects in the current scope. In general, we could for example point to ".Alerts.Firing" and choose alerts with the first "." and then the object ".Firing" which returns if the currently selected alert is firing. If we loop over ".Labels", then in this scope (between {{ range }} and {{ end }}) the object "." has already only Label child subobject, for example ".Annotations" for current ".Labels", etc...

Notification policy

To edit where and how messages are sent, we need to change the file "grafana_provisioning\alerting\notification_policy.yaml" directly. You can change the properties of the label tree in the file, but be careful. If you change for example the name of the label or its values, it will break all alert rules using this label to send messages to the contact point. One example of what you could change rather safely is time intervals:

- repeat_interval = time after which Grafana would send the same alert notification again (we don't want that)
- group_interval = time after which Grafana would send the notification again in the same group
- group_wait = waiting time for how long Grafana waits for other alerts in the same group before sending an alert message

Be aware of white spaces and tabs in this file, the system is sensitive to that.

Mute timings

Mute timings are settings that turn off alert notifications at defined times and dates. If one wants to edit a mute timing which is used in the labels "timing", open the file: "grafana_provisioning\alerting\mute_timings.yaml". You can for example add new days in the holiday list like this:

```
    - times:
            - start_time: "00:00"
              end_time: "24:00"
        days_of_month: ["5"]
        months: ["9"]
        years: ["2024"]
```

This could be inserted under "time_intervals" in the "holidays" section and would add a whole day with silenced alerts (with label timing=work) for the 5th of September 2024. One could also use the syntax for more days like "["5:7"]" or "["5", "6", "7"]".

Chapter 2

FSM integration of software and hardware status of MOPShub for beginners

2.1 Introduction to MOPSHUB4B

The MOPS chip is a radiation-hard ASIC for **Mo**nitoring Of **P**ixel **S**ystem. They are monitoring module voltages and temperatures inside of the detector volume. Each SP (Serial powering) chain contains one MOPS chip which is read out using one CAN bus per MOPS chip. The whole logic structure of the LLS testing setup can be seen in Fig. 2.1. The current solution for the readout of the MOPS chip is called MOPSHUB for beginners (MOPSHUB4B) and consists of 3 CIC cards (CAN-interface cards) and a RaspberryPi server (see Fig. 2.2). Each CIC card can operate 2 CAN buses and, therefore, read out two MOPS chips, in total six. Data can be read out up to from 4 SP chains and MOPS chips from Optobox. A Readout script written

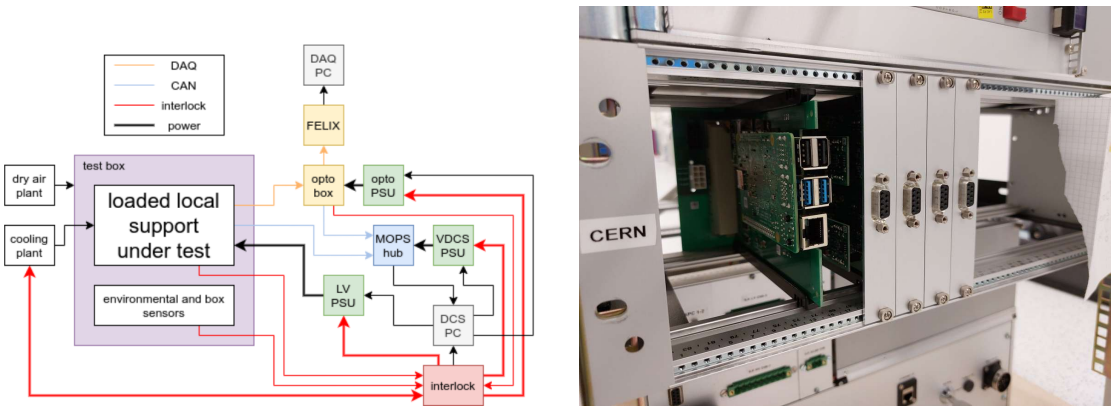


Figure 2.1: Logic structure of test setup for LLS with MOPS hub on the left, on the right side there is RaspberryPi controller in the rack of testing setup. [2]

in Python is running on the RaspberryPi. This software takes care of the readout of data from MOPS chips through CIC cards and populates an OPC-UA server. Initially, the readout software had to be (re-)started manually on the RaspberryPi in a complicated way: connect to the RaspberryPi via SSH, find the running process, kill it, and then start it by executing a Python command. The user also had no information about the state of the software, even if it was running. Our goal was to make a system for controlling the RaspberryPi remotely and also have the possibility of logging the current state. This system has been added to the Finite state machine (FSM) and Detector control system (DCS) of Pixel LLS.

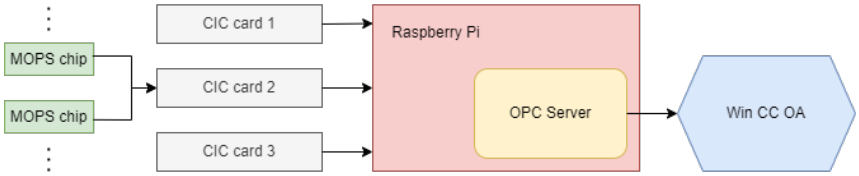


Figure 2.2: Diagram of MOPS hub for beginners.

2.2 MOPSHUB Software in DCS

GUI and scripts were programmed in WinCC OA, the commercial SCADA software chosen for the experiments at the LHC, extended with components from theJCOP Framework. WinCC OA uses a programming language called CTRL based on the C++ syntax. There is a GUI for DCS, where every part of the detector can be controlled from well-defined states. States of the subsystems influence the state of the parent system and the whole detector behaves as a finite state machine. That means the machine has defined all states in which the detector can be and the system knows processes which can take the detector from one state to another.

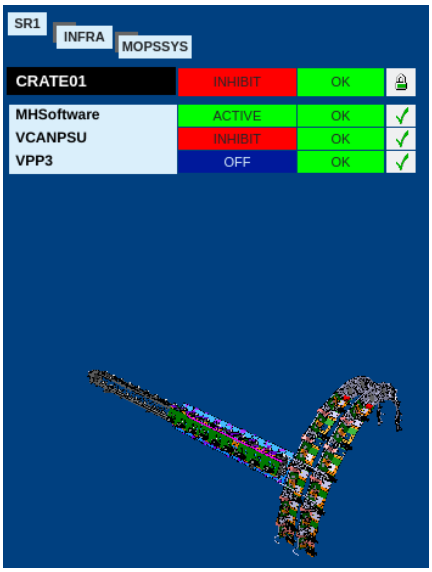


Figure 2.3: GUI of the FSM for a detector control system with the MOPSHUB4B software device unit (highlighted with a red rectangle).

There is an example of the FSM GUI in Fig 2.3, where we can see device units in the CRATE01 control unit which handles all the necessary devices to control a MOPSHUB crate. This CRATE01 unit is part of a bigger system called the MOPSSYS unit, that collects all MOPSHUB crates. MOPSSYS is then part of the infrastructure. There are states as well as alert statuses for each device unit. Readout software described in the previous chapter populates the OPC UA Server and it is accessed by WinCC OA as a client.

Three basic methods for checking the state of the MOPSHUB software have been developed. The first "ping" method sends ping commands to the RaspberryPi server and checks if it is running. If not, the system concludes that the RaspberryPi itself is OFF. If it is running, the system uses a second method to try ssh login to the server using ssh keys configured in the GUI. If the process is unsuccessful, the system takes the state that RaspberryPi is running but it is not connected (not possible to establish connection to it). If the connection is working properly, the system will ask through the SSH connection about the status of the systemd service which takes care of running the Python readout script. It will return one of five possible outputs. The advantage of using a service and not controlling the Python script directly is major. The way how to kill the process is well-defined, there is always only one currently running process and it is not possible to not kill

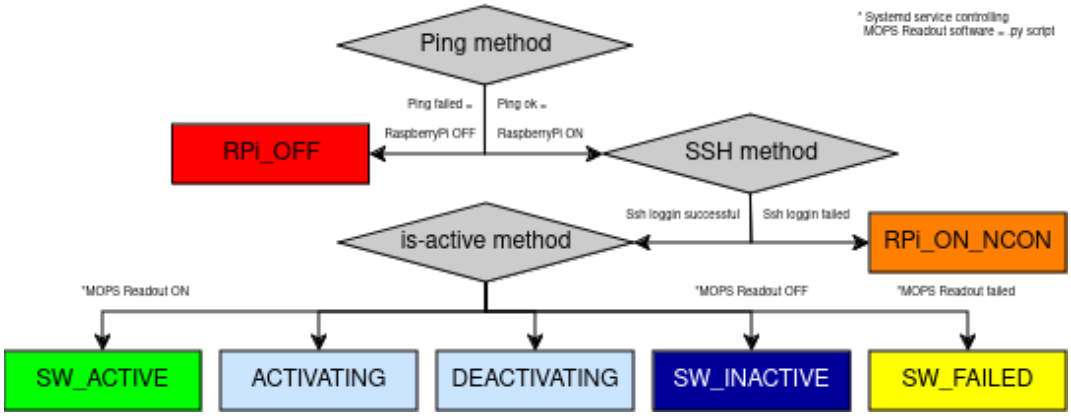


Figure 2.4: Diagram of MOPS Software states in the FSM of DCS.

software and accidentally start another one. We can easily get the status of the software to see if it is running properly. One of the downsides is that if we get the state "active", it just means that this service software successfully starts the Python script, not that the MOPS readout is running properly. For that reason, a readout of logs from the Python script has been implemented.

A diagram of all these mentioned methods and states of the MHSoftware can be seen in Fig. 2.4. There are 7 states in which the system can be. The GUI of the MOPS system, which was created, can be seen in Fig. 2.6.

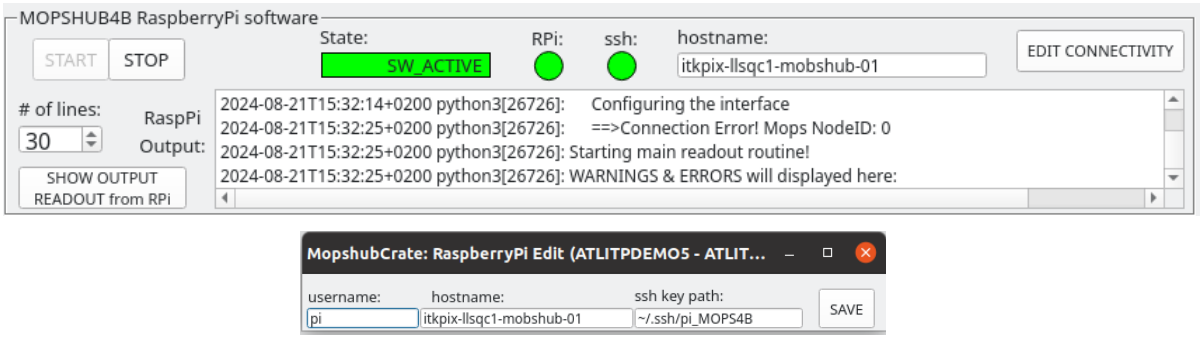


Figure 2.5: GUI of MOPSHUB4B Software in the DCS.

The user has the ability to stop the software by clicking on the button "STOP" (from states "SW_ACTIVE", "SW_FAILED" ⇒ "DEACTIVATING"), or start the software by clicking on the button "START" (from states "SW_INACTIVE", "SW_FAILED" ⇒ "ACTIVATING").

Please, do not stop the software immediately after starting, it could break the Python script. To get information if the software is already running properly (and can be stopped safely) you can use another button "SHOW OUTPUT READOUT from RPi". It will load logs from RaspberryPi python program and display them in the GUI. You can adjust how many lines you want to display above this button. If you see the message "WARNING & ERRORS will be displayed here:", you can safely turn off the software again.

A rectangle-like icon in the middle shows the name of the current state of the MOPSHUB Software and denotes it also in its color (same as in Fig. 2.4). The state is checked regularly every 10 s and every time some parameters changed (like hostname, etc. . .). There are also two icons on the right showing the state of RaspberryPi (RPi) and ssh connection itself. If the "ping" method is successful, "RPi" is green, otherwise red. Same for the SSH connection method with the second icon.

On the right side of the GUI, the chosen hostname of RaspberryPi server can be seen. If settings are wanted to be changed we need to click on the "EDIT CONNECTIVITY" button. Another window will appear (as can be seen below) where we can change the username and hostname of the connection and also a path to the SSH key file for the SSH connection. After clicking on the "SAVE" button, these parameters are saved in so-

called data points and the system automatically rechecks the state of the MHSoftware with updated variables. MHSoftare will move from "ACTIVATING/DEACTIVATING" to a state returned by another call of sets of

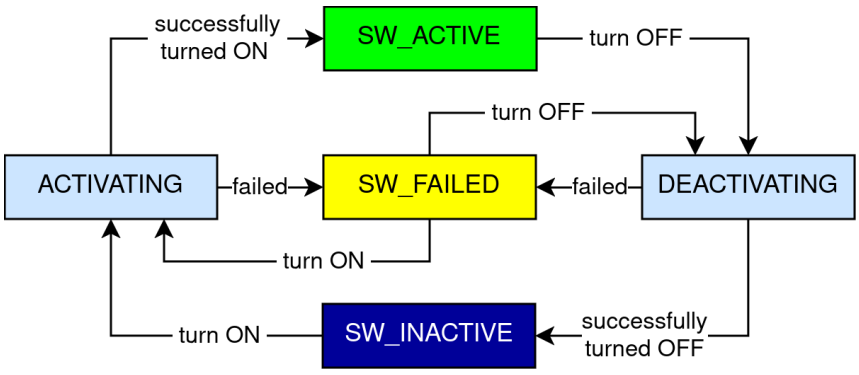


Figure 2.6: FSM state diagram with possible state transitions.

methods checking the state. That means the system could go from these states to any other state depending on the actual behavior of the MHSoftware. The state "SW_FAILED" means that the systemd service could not turn on or off the script. That means probably a problem with the Python script directly or systemd script. That could also happen if you try to send too many commands to the service script at once (for example check state, turn it on, and then turn it off immediately again).

2.3 Setup tutorial

The repository for ATLAS ITk Pixel DCS SR1 setup can be found here: `https://gitlab.cern.ch/atlas-itp-dcs/dcs_itkpixsr1st`. The implemented changes are published in release 2.3.3 or newer. You need to download a release and follow the description in the README. Then there are two steps you need to take described in the sections below.

2.3.1 Systemd service

Systemd (systemctl) is a software suite providing system components for Linux operating systems. This suite of software can manage many aspects of servers, from services to mounted devices and system states. Service units are used to manage software. The unit describes how to manage a service or application on the server. This will include how to start or stop the service, under which circumstances it should be automatically started, and the dependency and ordering information for related software. In the following lines, we will describe how to create a basic service for MOPSHUB Software operating on RaspberryPi. Prerequisites are:

- Working systemd on your Linux server. You can check it by:

```
$ systemctl --version
```

- Working python environment. Check with:

```
$ python3 --version
```

- The python script for readout of MOPS chips.

Open a terminal and go to /etc/systemd/system/ directory. Here we want to create a file with the name MOPS4B.service. Technically we can choose the name as we want but then the software running in the DCS would not find the service unit and fail. Please keep this name the same for that reason. In the file, we will paste the following code:

```
[Unit]
Description=Service for main.py of MOPS4B system
After=multi-user.target

[Service]
Type=simple
User=pi
WorkingDirectory=/home/pi/dev_23
ExecStart=/usr/bin/python3 /home/pi/dev_23/main.py
KillSignal=SIGINT

[Install]
WantedBy=multi-user.target
```

"WorkingDirectory" and "ExecStart" contain path to the Python script and its name. This must be adjusted in the case of a different name of the script or path to it. You may also need to specify the name of the user that you are using on the RaspberryPi server in line with "User" (our case "pi"). Then save the file and type the following command to reload systemd units in the system:

```
$ sudo systemctl daemon-reload
```

Then you need to enable this service unit with the following command:

```
$ sudo systemctl enable MOPS4B.service
```

You can test if it is enabled by the command:

```
$ sudo systemctl is-enabled MOPS4B.service
```

Then if you want to start or stop the service you can do it with the first two following commands. Be aware that stop of the service means stop of python main.py script which could still be in the starting phase and could not result in good behavior.

```
$ sudo systemctl start MOPS4B.service
$ sudo systemctl stop MOPS4B.service
```

If you want to check if the service unit is running you can use one of the following scripts. The first one will return just the state, the second one will provide more information about the state.

```
$ sudo systemctl is-active MOPS4B.service
$ sudo systemctl status MOPS4B.service
```

2.3.2 SSH keys generation and setup

The prerequisite for a successful remote control is the SSH package on both your server where you run DCS and on the RaspberryPi. You can check it just by writing "ssh" in your command line. In the following, configuration steps for an SSH key-based authentication are described.

To generate ssh key you need to run the following command:

```
$ ssh-keygen -t <key_algorithm> -f <key_store_path>
```

with any valid key algorithm (see web for it) and the path where you want to store the key. We used the following option:

```
$ ssh-keygen -t ed25519 -f ~/.ssh/pi_MOPS4B
```

After you have generated your key, you need to send it to the RaspberryPi server. For that use the following command:

```
$ ssh-copy-id -i <key_store_path> <user>@<host>
```

In our case, it looks like:

```
$ ssh-copy-id -i ~/.ssh/pi_MOPS4B pi@itkpix-llsqc1-mobshub-01
```

You need to log in and then it should be automatically configured. To check if it is working properly you can try to log in by following the command (the second one is an example of our case):

```
$ ssh -i <key_store_path> <user>@<host>
$ ssh -i ~/.ssh/pi_MOPS4B pi@itkpix-llsqc1-mobshub-01
```

You should be logged into the server without the need to write your password.

Then we need to configure SSH settings in DCS GUI (Fig. 2.6). Open the window by clicking on the "EDIT CONNECTIVITY" button and fill in your username, hostname to the RaspberryPi server, and path to the SSH key you just created. After clicking the "SAVE" button, everything should be configured and you will see the status of the MHSoftware in the GUI and FSM.

Chapter 3

Interlock Matrix integration to DCS project for OB LLSs

There are situations in which we don't want to continue testing the detector. One of them is when there is too much light in the test box. Light creates free charge carriers in the depletion zone of the sensors and significantly increases the current flowing through the sensor. If the high voltage supply would not be disconnected, the detector would be damaged.

In DCS there is a hardware Interlock logic implemented in a Field-programmable gate array (FPGA). Our goal is to create a script to translate these programmed settings in FPGA to an interlock matrix, that is displayed in the DCS (for monitoring purposes only). In the current system, the DCS display is done manually and with an scripted way, we can reduce a source of error.

The displayed interlock matrix has inputs to the interlock as columns and outputs from the interlock as rows. The script then needs to make from this logic structure an actual .XML table which could be used to configured the DCS GUI. An example of this GUI is shown in Fig 3.1.

Interlock		IN_S7_C5	NTC_S3_C2	NTC_S3_C1	IN_S7_C6	NTC_S4_C
Input		BoxSwitches-MIC	SP2/Tilock	SP1/Tilock	LightInterlock-MIC	Optobox_Power
Dep 1		FALSE (stale)	0.00 (stale)	0.00 (stale)	FALSE (stale)	0.00 (stale
Dep 2			0.00 (stale)	0.00 (stale)		0.00 (stale
OUT_S9_C3	OB/L3/B03/A/SP2/LV/CH/ILK	☑	☑	☐	☐	☑
OUT_S9_C4	OB/L3/B03/A/SP1/LV/CH/ILK	☑	☐	☑	☐	☑
OUT_S12_C1	OB/L3/B03/A/SP2/HV/CH1/ILK	☑	☑	☐	☑	☑
OUT_S12_C2	OB/L3/B03/A/SP2/HV/CH2/ILK	☑	☑	☐	☑	☑
OUT_S12_C3	OB/L3/B03/A/SP1/HV/CH1/ILK	☑	☐	☑	☑	☑
OUT_S12_C4	OB/L3/B03/A/SP1/HV/CH2/ILK	☑	☐	☑	☑	☑
OUT_S15_C12	INFRA/MOPSSYS/CRATE01/VCANPSU/ILK	☑	☐	☐	☐	☐
OUT_S15_C1	OB/L3/B03/A/SP2/OPTO/CH/ILK	☐	☐	☐	☐	☑
OUT_S15_C2	OB/L3/B03/A/SP1/OPTO/CH/ILK	☐	☐	☐	☐	☑

Figure 3.1: Example of an Interlock matrix table in the DCS GUI.

The chosen approach how to do it is to read the LISSY configuration withcode similar to the FPGA compiler to interpret the configuration file. The structure of this code is described in the following picture in Fig 3.2. A lexer creates tokens from the plain text of the source code. Tokens are shorter strings with labels characterizing their meaning. The parser takes the stream of tokens and interprets them to build an abstract object that describes what the source code is doing. In our case, the parser would create a list of outputs that are triggered in the case that a specific interlock input has been triggered. Finally, the tableizer makes from these abstract objects an .XML table.

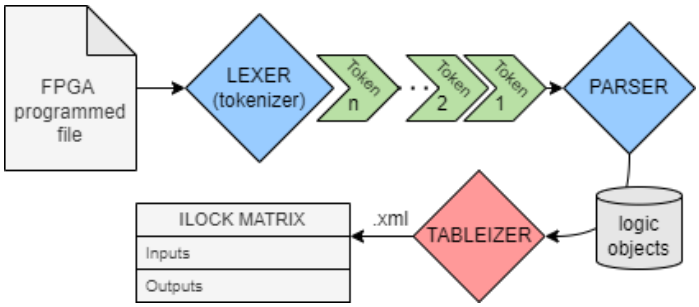


Figure 3.2: Schematic of the program logic of how to translate FPGA programmed settings to an .XML table.

Summary

I created two systems that will be used for the DCS during detector quality control tests. Another part of our work was writing documentation that will help others understand the new features quickly and provide them with easy ways how to implement them in their systems. These documents about Grafana and the MOPS Software are uploaded in the CERN's Engineering Data Management Service (EDMS).

We developed a fully functional system displaying data in a Grafana web application. Part of the work was the creation of the alert system which will notify operators during the tests. These alerts could be very helpful as they allow operators to take action before an interlock stops the measurement. We are deploying this system using docker. Other institutes can download the repository and with only a few commands start a docker container with the exact same software and settings that we are using. This process was already used by another institute and they are currently using part of our software.

In the second part of our work, we created a device unit for the MOPS Software in the detector control system for ATLAS ITk Pixel OB. It provides options to turn off and on the MOPS software, get readout logs, and shows complex status describing the state in which software is. This update is already part of the new release of the DCS for ATLAS ITk.

In the last part of our project, we tried to prepare the system for the integration of the Interlock Matrix into the DCS project from FPGA source code. The plan for the project has been designed. A working Lexer was created and tested. The Parser is in the phase of development and needs to be finished in the future.

List of Figures

List of Tabels

Bibliography

[1] ATLAS Collaboration. *ATLAS Inner Tracker Pixel Detector Technical Design Report*. Tech. rep. 2024-08-21. CERN, 2018. URL: https://cds.cern.ch/record/2285585/files/ATLAS-TDR-030.pdf.

[2] Diego Álvarez, Ana Cueto, Susanne Kuehn, and Benedikt Vormwald. *ATLAS ITk Pixel Outer Barrel Design*. Tech. rep. 2024-08-21. CERN, 2022. URL: https://edms.cern.ch/document/2822664/1.

[3] *Grafana documentation*. URL: https://grafana.com/docs/grafana/latest/ (visited on 08/21/2024).

[4] *GO package, regular expression syntax*. URL: https://pkg.go.dev/regexp/syntax (visited on 09/06/2024).

[5] *Go's templating language*. URL: https://pkg.go.dev/text/template (visited on 09/10/2024).

Appendix - LLS

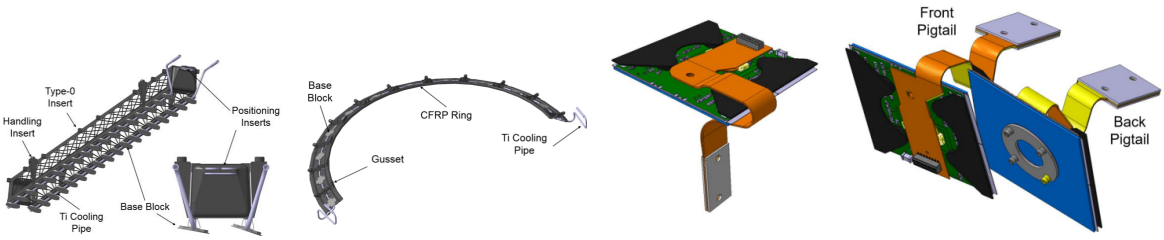


Figure A.1: CAD views of the Longeron and Inclined half ring on the left side and the individual modules support structure on the right side. [2]

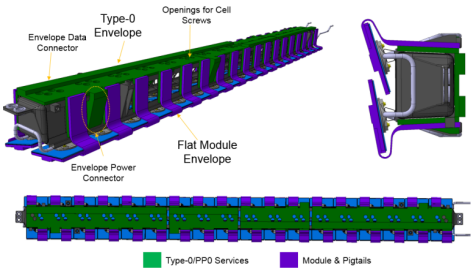


Figure 95. CAD view of the envelopes defined for the PP0s, the module cells and the pigtails of a longeron from layer 4.

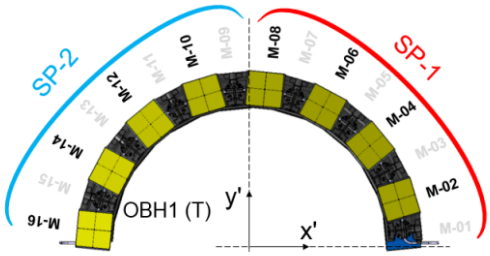


Figure 33. Numbering of the serial powering chains in an Outer Barrel Inclined Half Ring. The number of modules in each SP-chain varies with the layer: eight, eleven and twelve modules are grouped together in the Inclined Half-Rings for layers 2, 3 and 4 respectively. The ring showed corresponds to layer 2.

Figure A.2: Detailed CAD views of the Longeron on the left side and Inclined half ring on the right side with an indication of individual serial powering (SP) chains. [2]