ATLAS DCS

FSM Integration Guideline

Document Version: 3.0
Document ID: ATL-DQ-ON-0010
Document Date: 01.02.2016
Document Status: In Work

Institutes and Authors:
CERN: A. Barriuso Poy, S. Schlenker

Table of Contents

1. Preface

The ATLAS detector control system will be represented by means of a finite state machine (FSM) hierarchy which is operated by a DCS operator through an FSM and alarm screen.

2. FSM Architecture

The DCS Back-End system in ATLAS is organized in three functional horizontal layers and the FSM is the main tool for the implementation of the full control hierarchy (see Figure 1).

The detector is broken down into finite state machine units that are hierarchically controlled by other FSMs. These units can represent device entities, like a pump or a high-voltage crate, or logical groups of such devices, like a sub-detector or a gas system. Each unit will react on changes of the internal status of the individual device or groups of devices it is representing and allow simplified control, error handling and interaction with other detector components in the hierarchy.

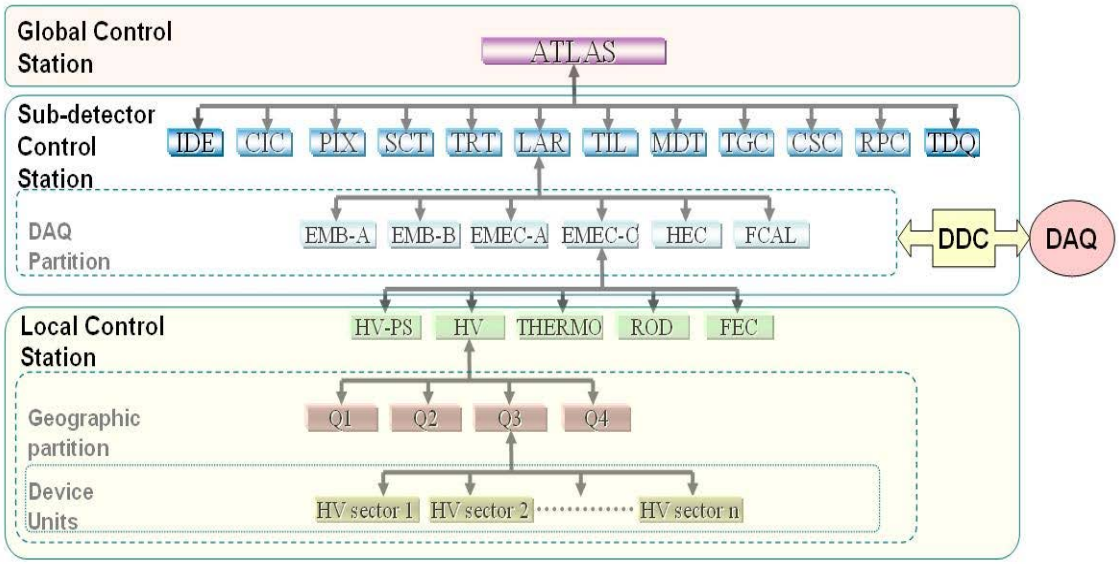


Figure 1: ATLAS FSM Architecture

2.1. *Global Control Station*

On the top, there will be a Global Control Station (GCS) which is in charge of the overall operation of the detector; it provides high level monitoring and control of all sub-detectors, while data processing and command execution are handled at the lower levels. The GCS may trigger actions itself or propose to the operator to do so.

2.2. *Sub-detector Control Station*

The Sub-detector Control Stations (SCSs) form the middle level of the hierarchy. The SCS allows the full local operation of the sub-detector.

At this level in the hierarchy, the connection with the Data AcQuisition (DAQ) system takes place in order to ensure that detector operation and physics data taking are synchronized. At

the SCS, the sub-detector is divided into partitions which are based on the DAQ TTC (Timing, Trigger and Control) zones (see also Figure 1). The synchronization of both systems is accomplished by means of the DAQ-DCS Communication (DDC) software package. The FSM plays an important role during the interaction with DAQ. It reports the DCS state of the TTC zones to DAQ and it executes commands received from DAQ (for more information see Appendix E).

2.3. *Local Control Station*

The bottom level of the hierarchy is made up of the Local Control Stations (LCSs), which handle the low level monitoring and control of instrumentation and services belonging to the sub-detector. The LCSs execute the commands received from the SCS in the layer above, but may also trigger predefined actions autonomously if required.

Each LCS is in charge of a certain system within the sub-detector (i.e. HV, cooling, etc). It is highly recommended to divide all these systems according to common geographic zones (i.e. quadrants, disks, slices, etc). Thus, the sub-detector can be organized in both, a system view and a geographical view.

The very bottom level is formed by Device Units (DUs) which link the FSM with WINCCOA and define the *granularity* of the system. The information located below these boundaries is encapsulated and not accessible from the FSM. At this point, one has to find out the structure of encapsulation which will yield the best system decomposition. In order to choose this granularity several points have to be taken into account:

1. Too fine granularity (channel level) requires a large number of connections between the FSM and WINCCOA, which may overload the processors (see Sec. 3.1).
2. Very coarse granularity would accumulate too much information in a single entity, making it difficult to define its functioning states.
3. The DUs are the smallest entities to which commands can be sent from levels above.
4. Ideally, even in case of evolution of the Front-End, the chosen DUs should be re-usable.

3. FSM Implementation

In this section, the basic concepts of the ATLAS DCS FSM implementation are explained and some design considerations and recommendations are given along with hints for the actual implementation of the FSM hierarchy. All necessary steps which have to be performed to create your own hierarchy are described in detail in Appendix A.

3.1. *Recommendations*

Performance Issues

During the creation of the hierarchy the final performance has to be taken into account. Thus, in order to build a safe/robust hierarchy it is recommended not to exceed a certain number of FSM elements:

- CU - Control Unit (**Up to 50 CUs per WINCCOA PC**)
  - Can be Included, Excluded, etc and Taken in stand-alone mode.
  - Corresponds to one smiSM process.
- LU - Logical Unit (**Up to 500 LUs per WINCCOA PC**)
  - To be used at the bottom levels of a tree (just above the DUs).
  - Can contain children, but not of type CU
  - Can be Enabled/Disabled (can not run in stand-alone).
  - Corresponds to an object within a smiSM.
- DU - Device Unit (**Up to 1000 DUs per WINCCOA PC**)
  - Corresponding to a "real" device in WINCCOA.
  - Can be Enabled/Disabled (can not run in stand-alone).
  - Behaviour defined via WINCCOA scripts (instead of SMI code).

The numbers presented above are recommendations, and, under certain circumstances, they could be exceeded. However, it is foreseen that any system belonging to a “normal” control hierarchy will not need to exceed these quantities.

Alarm Handling

Alarms from the WINCCOA alert configurations at the data point level will be displayed using the framework (FW) Alarm Screen, and are intended to be used for detailed problem tracking and acknowledgement. It is strongly recommended to have an alert handling configuration at least for each Data Point (DP) that corresponds to a certain DU.

In addition, a simplified alarm handling mechanism is introduced at the level of the FSM units – the “STATUS” (see next section) – representing a scaled down version of the WINCCOA alerts. The STATUS allows for context based signalisation of problems and error tracking inside the control hierarchy directly on the FSM operator interface. Note that consequently the alarms of those DPs which are not considered in the hierarchy are thus only visible on the FW Alarm Screen.

Command Execution

For the final production systems it is envisaged that DCS users will operate the systems *only* through the FSM and the FW Alarm Screen.

Thus, the developer must implement the commands to be sent by the operators inside the FSM units rather than from WINCCOA panels or scripts. This will help to maintain and understand the different DCS projects and ensure proper execution of actions by a single process.

FSM Version Consistency

When integrating different FSM trees belonging to different WINCCOA systems check that the FSM versions are the same. The most recent FSM production version to be used will be announced on the central ATLAS DCS web page (https://twiki.cern.ch/twiki/bin/view/Atlas/AtlasDcsPackages) or directly install it from the central DCS repository disk (/winccoa/fwComponents, see https://twiki.cern.ch/twiki/bin/view/Atlas/AtlasDcsTestSystem).

If you are upgrading your FSM please always check the release notes before:

http://lhcb-online.web.cern.ch/lhcb-online/ecs/fw/FW_FSM.HTML

Panel Export

It is important to verify that your final operator panels function correctly in different systems since they will be exported to other distributed projects (i.e. GCS or SCS).

► Most probably the system name will be different in your development project compared to your production project(s). Check your WINCCOA panels before exporting them to the GCS.

Simple DU Scripts

When defining the DU types, keep the *state configuration scripts* as simple as possible. These scripts are called each time there is a change on the value of the DP associated to the DU.

Unique DIM DNS node

During the development process one can run the `dns` binary locally. However, when integrating several FSM running in different computers a unique DNS DIM node or node list must exist. For final ATLAS, there will be a list of redundant DNS DIM nodes to be used by all FSM objects (`pcatlfsmdns1,pcatlfsmdns2`). In the CERN general purpose network, a dedicated test DNS is available: `pcatltestdns`.

3.2. *State & Status*

The “STATE” and “STATUS” are two aspects that work in parallel and provide all the necessary information about the behaviour of any system at any level in the hierarchy. The STATE defines the “operational mode of the system” and the STATUS gives more details about “how well the system is working” (i.e. it warns about the presence of errors). The main reasons that these two information elements are provided at each level of the hierarchy are the following:

- Information about the operational mode of a complex system or a group of systems is not lost when an error occurs. For instance, a HV system is in RAMPING_UP

state and this process may take several minutes to finish. If in the meantime a slight error (that permits to continue with the ramping up) occurs, it can be propagated up by means of the STATUS while keeping the same STATE. In addition, the two aspects define more accurately the behaviour of each level on the hierarchy.

- Complex systems can be supervised “more in detail”, e.g. an error may be treated differently depending on the operational mode of the system. As an example, depending on whether the STATE is ON or OFF, different severities can be attributed or different actions triggered. See Appendix C.
- The STATUS is somehow similar to the alert screen. Having the display of the STATUS within the FSM is useful to find out faster the information located in the WINCCOA panel of the element with an error…..

STATUS

The STATUS names are fixed and all sub-detectors must use these qualifiers. The colours fulfil the Framework Look-and-Feel convention.

STATUS	
OK	System working fine.
WARNING	Low severity. The system can go on working. To fix in the following working hours.
ERROR	High severity. Serious error for the functioning of the system. To be fixed ASAP.
FATAL	Very high severity. The system cannot work. Run away!!!

STATE

For consistency, the common control domains (i.e. SCS, HV, LV, Cooling, etc) should have the same (or similar) states, status, transitions and actions. As a result, we will make life easier to the future shift operator. Thus, two generic state machines are proposed in order to homogenize the different control domains.

The first state machine (see Figure 2) corresponds to those control domains that represent abstract entities, these control domains are:

- Sub-detector Control Station
- DAQ partitions
- Any geographic partition (i.e. a quadrant, a wheel, etc)
- Any environment system

Each sub-detector has 3 mandatory STATES: READY, NOT_READY and SHUTDOWN. These three states must be propagated to the GCS. In between these states, the sub-detector is free of defining their own states depending on the requirements within the lower levels of the hierarchy, i.e. below the TTC partition level. The sequence SHUTDOWN - READY passes normally through all intermediate states. Two additional states out of the normal sequence are UNKNOWN and TRANSITION. These states are reachable from any other state.

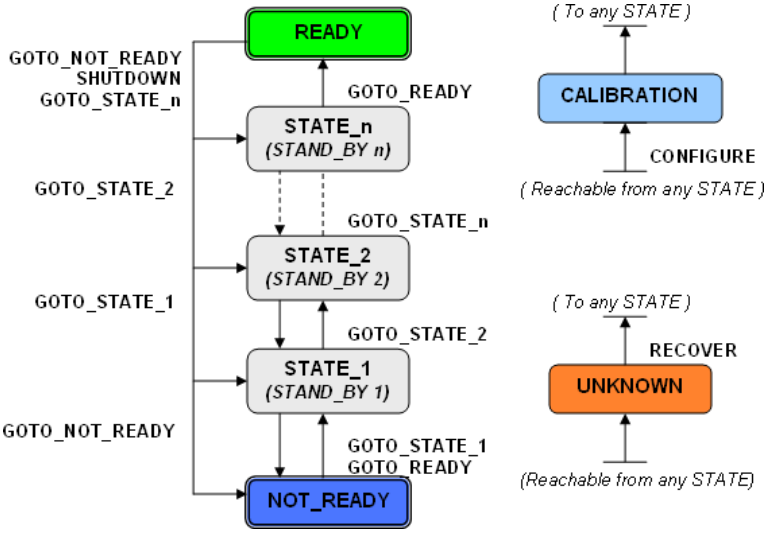


Figure 2: State machine for control units (CU), e.g. SCS, partitions, and environment.

The second generic state machine (See Figure 3) corresponds to those control domains that represent concrete device entities like the HV, LV, gas, cooling, rack, etc. These control domains have two mandatory end-points ON and OFF. In between these states, there could be transient states for those systems with slow response (i.e. RAMPING_UP, RAMPING_DOWN) and as many intermediate states as needed (i.e. ON25, ON50, STANDBY, etc).

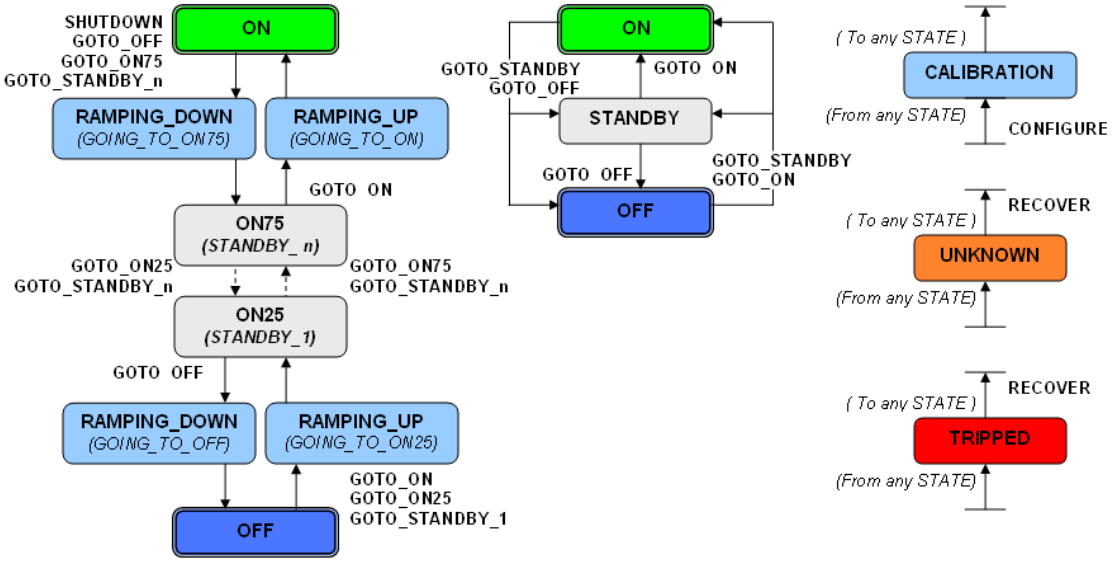


Figure 3: State machine for device units (DUs), e.g. an HV system. *Left*: Example with transient states and multiple intermediate states between ON and OFF. *Middle*: Simple version lacking transient states and only one intermediate state. *Right*: Additional optional states.

The colour coding for the states is the following:

STATE COLOUR CODING	
GREEN	Static state. The system reached its final operational stage
BLUE	Static state. The system did not reached yet its final operational stage
DARK BLUE	Static state. The system is in its lowest operational stage
TURQUOISE	Static state. The system is in the STANDBY state (safe for LHC unstable beams)
LIGHT BLUE	Transient state. Signalizes an ongoing transition between normal states
ORANGE	Error state. Used if state is UNKNOWN (e.g. due to loss of communication)
RED	Severe Error state. For example TRIPPED in case of a power supply trip.

The colors are defined within the *fwFsmAtlas* module.

Finally, below you can find a table with a list of states and associated commands as also defined in the FSM object type ATLAS_CU within *fwFsmAtlas*

STATE	COMMANDS
READY	GOTO_READY
NOT_READY	GOTO_SHUTDOWN, GOTO_READY
SHUTDOWN	GOTO_ READY
UNKNOWN	RECOVER
STANDBY	GOTO_STANDBY
ON, ON25, ON50, ON75…	GOTO_ON, GOTO_ONxx
OFF	GOTO_OFF
TRIPPED	RECOVER
LOCKED	UNLOCK, LOCK
RUNNING	RUN
STARTED	START
STOPPED	STOP
TRANSIENT STATES	**COMMANDS**
TRANSITION	
RAMPING_UP, GOING_TO_ON, …	GOTO_ON, GOTO_ON75, GOTO_STANDBY_n
RAMPING_DOWN, GOING_TO_ON, …	GOTO_OFF, GOTO_ON25, GOTO_STANDBY_n
CALIBRATION	CONFIGURE
GETTING_READY	GOTO_READY
GETTING_NOT_READY	GOTO_NOT_READY
STARTING	START
STOPPING	STOP

- All state, status and command labels should use upper case letters.

► If you are developing the FSM behaviour for a DCS system, please discuss the design (for each CU, LU or DU) on paper first with the central DCS team before starting the technical development process.

3.3. *FSM Hierarchy*

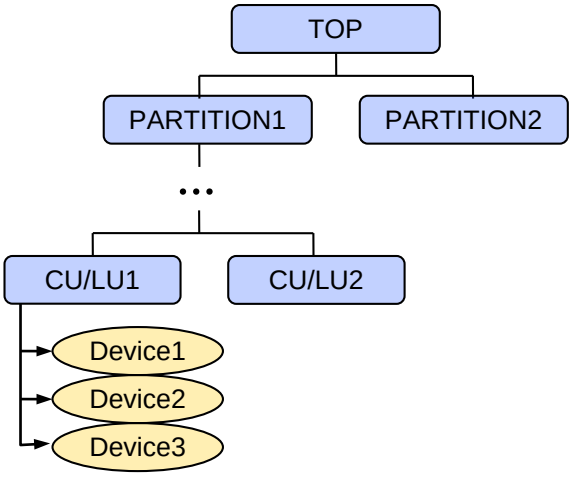


Figure 4: Example FSM hierarchy.

An FSM hierarchy for a particular subdetector should be structured as shown in Figure 4. Several design considerations must be taken into account. The first level below any subdetector top node (corresponding to the SCS) must contain all Atlas TTC partitions defined for this subdetector as control units (CUs). Further below, a number of subsequent layers of control units can be used to divide the subdetector into several logical parts, either into groups of identical subsystems or into geographical entities. This depends on considerations regarding control timing, error handling and configurations aspects. For example, it might be useful to group all HV systems of a partition to be able to switch them on or off altogether using a single CU. On the other hand, electrical power distribution might require controlling all systems of a specific geographical part of the subdetector. Further, the number of CU/LU layers between the partition and any device layer should be kept as low as possible and never exceed five.

On the lowest level, device units should be grouped such that one CU or LU controls devices of the same type. In addition, the number of devices per parent unit should be kept below ~100.

An example hierarchy including all FSM unit types is implemented within the fwFsmAtlas module as a framework component, called *AtlasFsmExample* (see Appendix A).

4. FSM Naming Convention

All FSM names must use upper case.

4.1. *FSM Object Names*

FSM object names should follow the conventions below. Note that a child object should always contain at least the last part of the parent name to allow to recognize parent/child relations from the name. **Attention:** The maximum number of characters for an object name is 32.

- Sub-detector top-most node:
  ATL_<sub-detector name>
  Ex: ATL_LAR
- The children of the sub-detector top node are the sub-detector's TTC partitions.
  <sub-detector name>_<TTC name>
  Ex: LAR_EMECC

- The children of the sub-detector can be either a geographical division or a system division.
  <sub-detector name>_<TTC name>_<geographic name>
  Ex: LAR_EMECC_Q1
  <sub-detector name>_<TTC name>_<system name>
  Ex: LAR_EMECC_HV

- The next level can again be either a geographical division or a system division.
  <sub-detector name>_<TTC name>_<geographic name>_<system name>
  Ex: LAR_EMECC_Q1_HV
  <sub-detector name>_<TTC name>_<system name>_<geographic name>
  Ex: LAR_EMECC_HV_Q1

- The next children will normally be a DU. However, if it is not the case, one could follow with the same convention:
  <sub-detector name>_<TTC name>_<geographic name>_<system name>_<part name>
  Ex: LAR_EMECC_Q1_HV_SECTOR1
  <sub-detector name>_<TTC name>_<system name>_<geographic name>_<part name>
  Ex: LAR_EMECC_HV_Q1_SECTOR1

- The STATUS nodes belonging to a certain node should add the prefix “STATUS_”.
  Ex: STATUS_ATL_LAR, STATUS_LAR_EMECC, STATUS_ LAR_EMECC_Q1

4.2. *FSM Object & Device Type Names*

The fwFsmAtlas component provides different type templates for all FSM types (e.g. *ATLAS_CU*) which should be used for the top-level FSM nodes. Below the partition level, custom FSM types can be used which should follow the following naming scheme:

- Logical Object Types:
  <Sub-detector name>[_<partition type>][_<sub-partition type>]
  Examples: LAR_HEC_LV, CIC_ENV_HUMIDITY

- Device Unit Types
  [*<Device base type>_]<sub-detector name>_<sub-system name>*
  Examples: fwAi_TRT_HVMODULE

For the STATUS FSM object, the predefined types *ATLAS_STATUS* and *ATLAS_DU_STATUS* should be used for CU/LUs and DUs, respectively. Whenever it is necessary to modify these types, follow the same naming scheme as above and append *_STATUS.*

4.3. *WINCCOA Panel Names*

Each Control Unit (CU), Logical Unit (LU) or Device Unit (DU) can have a WINCCOA panel associated to it. The name of this panel should be the same as the FSM object.

► Example: if there is a CU with name “TRT_SCS“, then the panel will be called “TRT_SCS.pnl”.

For each FSM node, it is also foreseen to have an additional secondary panel with the suffix “_info” (see FSM User Interface).

► Example: if the CU panel is called “TRT_SCS.pnl” the secondary panel will be called “TRT_SCS_info.pnl”

To follow the convention for the panel names is important for a later integration of all the ATLAS DCS distributed systems in the shift operator interface.

5. FSM User Interface

This section explains the common FSM Operator Interface (OI) and the basic procedure that needs to be followed in order to integrate all FSM OI in ATLAS. To make the integration easier an “fwFsmAtlas” module has been created.

The basic premise for the design of the final OI is that it needs to be operated from a single window. It is not desired to search information across windows in such a big process control system like the ATLAS DCS. To improve the human-machine performance all the DCS information is available in parallel. The idea is to have a single user interface allowing navigation through all the different levels of the FSM hierarchy (see Figure 1). In order to fit the big amount of DCS data in a single display, a frame with five constituent parts allowing for easy navigation has been designed (see Figure 4).

5.1. *Screen Layout*

The operator screens consist of two individual screens, the FSM and Alarm screen with a resolution of 1280x1024 each. The FSM screen is represented by a WINCCOA panel with fixed dimensions covering the whole screen. The panel contains several modules presenting the behaviour and allowing control of the detector at the different levels of the DCS hierarchy. *FSM Module*: The STATE and STATUS of a FSM node and its children is displayed providing all FSM functionality. This module has a limited size, and, in case of many FSM children a scroll bar appears.

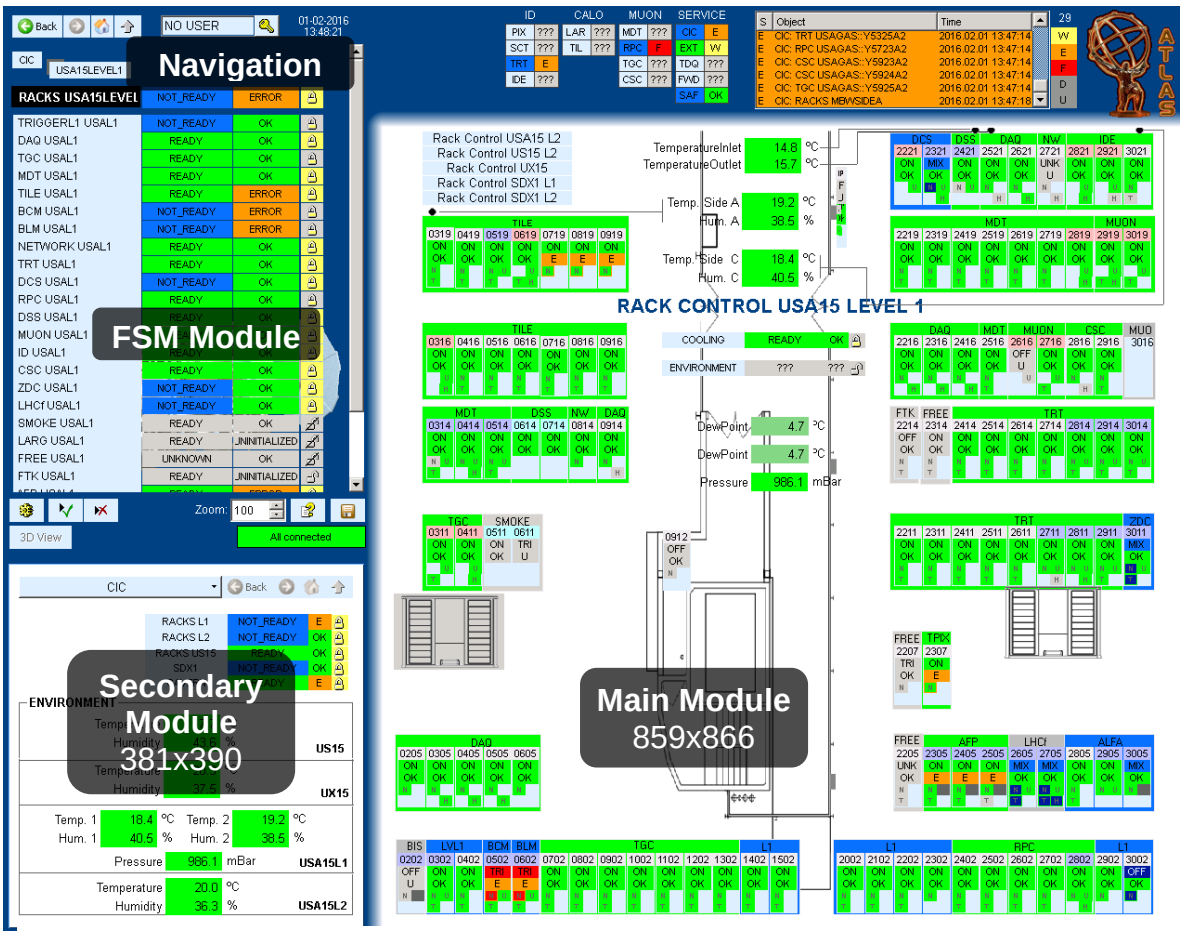


Figure 5: FSM operator screen components.

- *Main Module*: Its dimensions are 859x866. This is the main panel for the selected FSM node, this can be a SCS, a HV system, etc. Two dollar parameters are passed from the FSM Module to Main Panel ($node and $obj).
- *Secondary Module*: Its dimensions are 381x390. The purpose of this is to keep a main view of a certain sub-detector while studying more in detail a problem that triggers deeper in the hierarchy. Thus, it is needed to create an additional panel with a summary of the information presented for each main panel. Two dollar parameters are passed from the Main Panel to the Secondary Panel ($node and $obj). Optionally, the secondary module is used to display a 3-dimensional view of the detector DCS objects.

► A navigator is available, similar to that of web browsers. It has a four buttons:

- **Back**: The operator goes back to the previously used panel.
- **Forward**: The operator goes forward to the previous used panel.
- **Home**: The operator goes to the topmost FSM node.
- **Up**: The operator goes one level up in the FSM hierarchy.

► Additional navigation possibilities exist. See Appendix B.

In both Main and Secondary module, the developer has full FSM functionality (i.e. one can send FSM commands, change the partitioning mode, etc.). This also means that within a certain workspace (i.e. a HV system) information related to any other workspace (i.e. cooling) could be displayed. Using the navigation functionality the operator can jump from one workspace to another using any of both, main and secondary modules.

In order to assist developers in the creation of FSM panels with common functionality, a set of widgets have been created. These widgets permit to display the state and the status, change the partitioning mode, send FSM commands and navigate between different control domains (for more details see Appendix B.3).

5.2. *Operator Interface Design Problem*

As shown in Figure 6, two questions pertinent to interface design arise when creating the displays for the different work domains.

First one needs to distinguish the relevant way of describing the complexity of a work domain (content), and then, define the effective way to communicate this information to the operator.

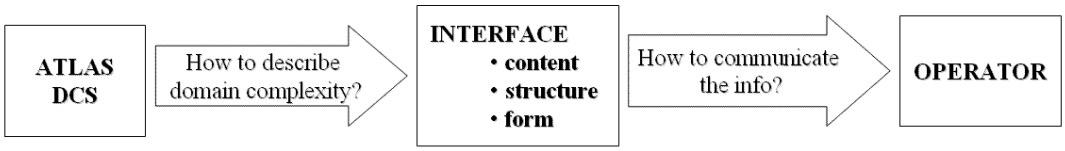


Figure 6: Structure of the interface-design problem

In designing a well-behaved man-machine interface for process control, some guidelines are added in order to solve the design problem of a general operator interface (consistency, intuitive approach, reduce chance for typing errors, etc.). These guidelines are listed here:

- The graphics interface must be designed with the operator's view of the process in mind.
- Functional relationships between sub-systems need to be represented in the display next to each other ("out of sight, out of mind"). Otherwise operators can treat the sub-systems as being independent of each other.

- The operators act directly on the display forcing feedback interaction and selection emphasis.
- The control hierarchy of the displayed information must be isomorphic. As described in section 1, the different sub-detectors will use a similar architecture.
- Each sub-detector is physically composed by many different systems (HV, gas, etc.). Thus, the system model contains both, a geographical and a functional representation of the sub-detectors.
- The demand seems obvious, but the operator must always be able to trust the values displayed. The usage of plots and time values can help.
- The interface should not be “programmable”, it is kept stable and solid.
- To prevent confusion, the operator interface itself should be as simple as possible.
- The displays must follow cultural standards (i.e. language English, Local Time and Standard Units).
- Irrelevant information must not be present. Otherwise, operators will have to determine what information to attend to, and what information to ignore.
- Critical information must be present having constrains between systems explicitly represented.
- The interface should be designed such as the perceptual saliency of its objects is relative to its importance.

5.3. *Access to the fwFsmAtlas Module*

The actual skeletons of the FSM panels are assembled within a framework component module *fwFsmAtlas* which is accessible via SVN.

► Installation and usage of the “fwFsmAtlas” component is described in Appendix D.

6. Outlook

In the next versions of this document the following topics will be included:

- The FSM and Access Control.

Appendix A. FSM Hierarchy Implementation

In the following, a step-by-step guide to create a very simple FSM hierarchy for a set of different devices (see Figure 7) is given. This example hierarchy is contained within the *fwFsmAtlas* module as *AtlasFsmExample* component. Note that this example can serve as a starting point for the actual implementation of your own hierarchy, i.e. you can copy the existing FSM unit types and modify them according to your own needs.

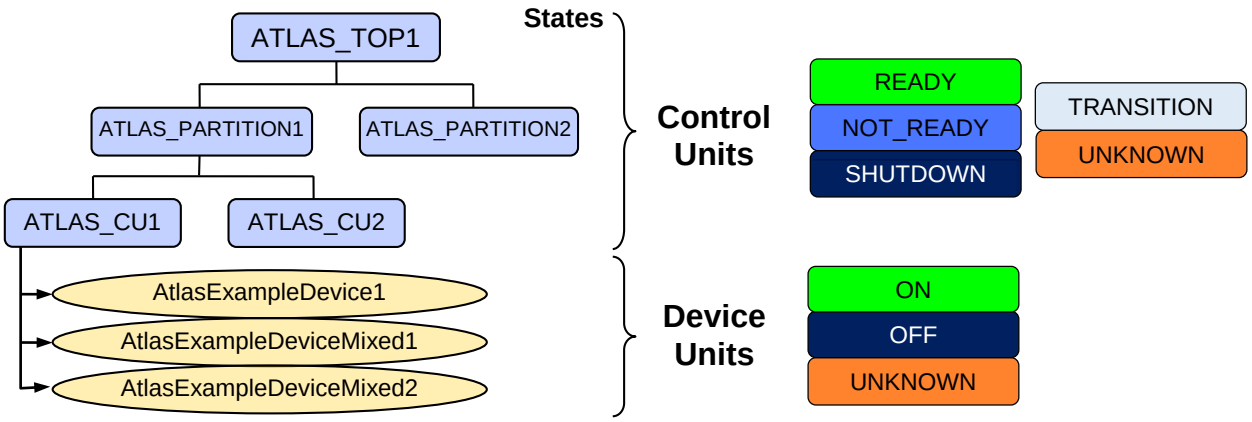


Figure 7: Example Hierarchy of the *AtlasFsmExample* component.

The mechanism adopted for modelling the structure of sub-detectors, sub-systems and hardware components is a hierarchical (tree like) structure. In ATLAS this tree is composed of two parallel paths, one for the STATE (CUs, DUs or LUs) and one for the STATUS (LU). A CU represents its own SMI domain while the DUs and LUs are part of the SMI domain of its parent CU. Each CU and LU in ATLAS must have associated a STATUS defining the severity of a pending problem if any.

A.1. Requirements

The actual FSM implementation depends on the design requirements, namely, the actual device representation in WINCCOA which needs to be controlled, and the interface to the top level Atlas control.

Devices

The example devices are represented by one or several WINCCOA data points (DPs). The framework FSM actually only allows to define FSM device units, which depend on a single DP. However, in many cases this is not sufficient since a FSM DU could represent a device which depends on values of another device, e.g. a HV channel for which the STATE depends on DP values of an OPC server.

The example devices used here are reflected by a DP of WINCCOA type *AtlasExampleDevice* and *ExampleOPC* (see Figure 8). There are three device instances for which the FSM will be implemented in a different way to illustrate the recommended possibilities.

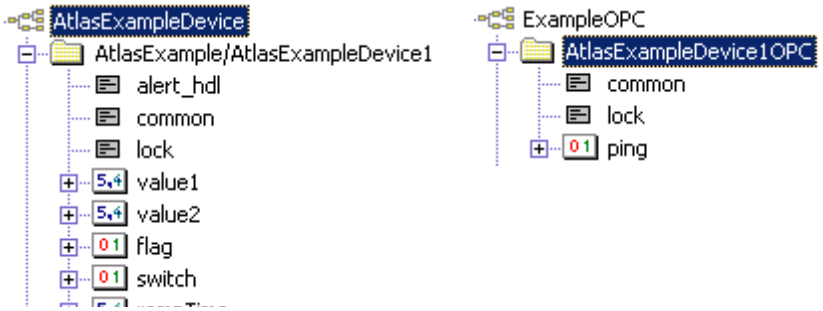


Figure 8: WinCCOA DPs used for the example devices.

Control Interface

The interface to the top-level control is represented by a FSM CU with the possible STATEs and STATUS information defined in Section 3.2.

A.2. Creation of FSM Unit Types

The first step of the actual implementation is the definition of FSM unit types for the devices and the logical control units forming the hierarchy. This can be done using the framework tool *DeviceEditorNavigator* (DEN) and applying the naming conventions described in Section 4.2 for types with different implementation.

Device Unit Types

Property Overview

There are three properties of the DU which have to be defined: initialization, STATE and STATUS definitions, and command actions. The STATEs for the DUs were chosen, according to the conventions described in Section 3.2, to be ON, OFF, UNKNOWN, or TRIPPED. The possible commands are GOTO_ON, GOTO_OFF and RECOVER. Upon creation of the device unit type, the three different functions

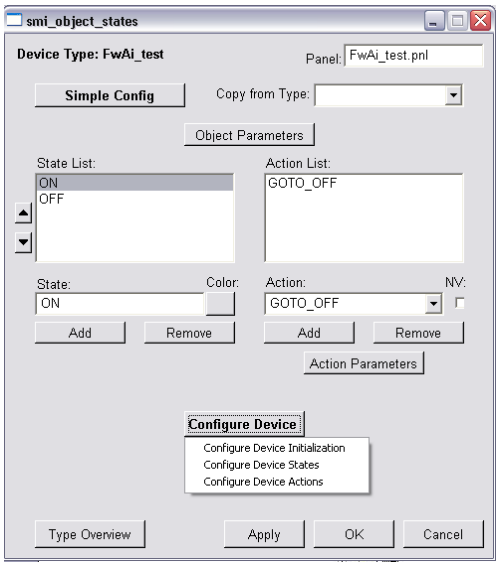


Figure 9: Device type dialog.

1) `<TypeName>_initialize(string domain, string device)`
2) `<TypeName>_valueChanged(string domain, string device, [type dpe, …], string &fwState)`
3) `<TypeName>_doCommand(string domain, string device, string command)`

must be implemented with the DEN. The scripts can be accessed from the button `Configure Device` of the device dialog shown in Figure 9. The initialization 1) is only executed on startup of the FSM. The most critical function in terms of performance is 2) since it is executed every time a DP element of the device changes which the STATE or STATUS depends on. The command execution 3) is executed every time an FSM command was issued to the device unit.

For each of the three functions implement a wrapper function inside a ctrl library. The library is recommended to have the following path and naming for a given sub-detector or system name XYZ:

`ATLAS_DCS_XYZ/scripts/libs/xyzFsm/xyz<TypeName>Fsm.ctl`

Implementation for Single DP Device

For a device which is represented only by one DP with several DP elements (here type *AtlasExampleDevice*), the implementation of the corresponding DU type *AtlasExampleDeviceDu* is straight forward:

```
#uses “xyzFsm/xyzAtlasExampleDeviceFsm.ctl”

AtlasExampleDeviceDu_initialize(string domain, string device)
{
  xyzAtlasExampleDeviceFsm_initialize(domain, device);
}

AtlasExampleDeviceDu_valueChanged(string domain, string device, float value1, float value2,
bool flag, string &fwState)
{
  xyzAtlasExampleDeviceDuFsm_valueChanged(domain, device, value1, value2, flag, fwState);
}

AtlasExampleDeviceDu_doCommand(string domain, string device, string command)
{
  xyzAtlasExampleDeviceDuFsm_doCommand(domain, device, command);
}
```

Within the library `xyzAtlasExampleDeviceDuFsm.ctl` the functions are implemented:

```
xyzAtlasExampleDeviceDuFsm_initialize(string domain, string device)
{
  // do any intialization of device here
}

xyzAtlasExampleDeviceDuFsm_valueChanged( string domain, string device, float value1, float
value2, bool flag, string &fwState)
{
  string targetStatus = "OK";

  if (flag) {
    fwState = "TRIPPED";
  }
  else if (value1 <= -0.5 || value2 <= -0.5) fwState = "UNKNOWN";
  else if (value1 <= 0.5) {
    fwState = "OFF";
  }
  else if (value1 < 10.) fwState = "RAMPING";
  else fwState = "ON";

  if (value2 > 2.) targetStatus = "FATAL";
  else if (value2 > 1.) targetStatus = "ERROR";
  else if (value2 > 0.5) targetStatus = "WARNING";

  fwFsmAtlas_setStatus(domain, device, targetStatus);
}

xyzAtlasExampleDeviceDuFsm_doCommand(string domain, string device, string command)
{
  action(domain+”::”+device+” starting command ”+command);
  if (command == "GOTO_OFF") {
    dpSetWait(device+".switch:_original.._value", false);
    fwDU_startTimeout(30, domain, device, "UNKNOWN", "OFF");
  }
  if (command == "GOTO_ON") {
    dpSetWait(device+".switch:_original.._value", true);
    fwDU_startTimeout(30, domain, device, "UNKNOWN", "ON");
  }
  if (command == "RECOVER") {
    // do any recovery here
```

```
    fwDU_startTimeout(30, domain, device, "UNKNOWN", "OFF");
  }
}
```

Here, the DPE *.value1* and *.flag* determine the STATE and *.value2* influences the STATUS. Note that the full implementation can not be done with the "SIMPLE CONFIG" wizard since it does support neither the STATUS nor several DPEs.

STATE and STATUS Dependence on Several DPs

A complication arises in the case where the STATE and STATUS of the device include another DP, here a DPE *AtlasExampleDevice1OPC.ping*. The usage of the `valueChanged()` function is no longer possible for this DU (here *AtlasExampleDeviceMixedDU*) since it is only executed on changes within the DP *AtlasExampleDevice1*. The implementation must be changed in a way that the function `valueChanged()` has to be left empty, and replaced by another callback function which is then connected in the initialization stage using a `dpConnect()` call:

```
xyzAtlasExampleDeviceMixedDuFsm_initialize(string domain, string device)
{
  // do any initialization of device here
  dyn_string nameComponents = strsplit(device, '/');
  string ping = nameComponents[dynlen(nameComponents)]+"OPC.ping:_online.._value";
  string value1 = device+".value1";
  string value2 = device+".value2";
  string flag = device+".flag";

  string nodeName = fwFsmAtlas_getNodeDPEName(domain, device);
  dpConnect("AtlasExampleDeviceMixedDU_callback", nodeName, value1, value2, flag, ping);
}

xyzAtlasExampleDeviceMixedDuFsm_callback(string nodeName, string tnode, string dpe1, float
value1, string dpe2, float value2, string flagDPE, bool flag, string pingDPE, bool ping)
{
  string domain, device;
  fwFsmAtlas_getNodeNameComponents(nodeName, domain, device);

  // determine State
  //
  string state = "impossible"; // should be impossible
  if (!ping) state = "UNKNOWN";
  else if (flag) state = "TRIPPED";
  else if (value1 <= 0.) state = "OFF";
  else if (value1 < 10.) state = "RAMPING";
  else state = "ON";

  fwFsmAtlas_setDUState(domain, device, state);

  // determine Status
  //
  string targetStatus = "OK";

  if (value2 > 2.) targetStatus = "FATAL";
  else if (value2 > 1.) targetStatus = "ERROR";
  else if (value2 > 0.5) targetStatus = "WARNING";

  fwFsmAtlas_setStatus(domain, device, targetStatus);
}
```

In this case, the `valueChanged()` part of the DU type implementation can be left empty:

```
AtlasExampleDeviceMixedDU_valueChanged( string domain, string device, float value1, string
&fwState)
{}
```

The command execution is not affected. Note two important limitations:

- the name of the 2nd DP is derived from the device name since the mechanism has to work generically for all devices of this type
- the callback function has to include an FSM-internal DP in order to be able to derive the domain and device names since WINCCOA does not allow to pass constant parameters in a `dpConnect()` call.

Usage of Alert States instead of Value Changes

The valueChanged or custom callback function described above is called on every change of each considered DPE value. If only certain ranges of the value correspond to an actual change of the STATE or STATUS of the device, it if highly recommended to connect the callback function to the actual alert states of the value DPE. For example, having defined an alert configuration with the alert text corresponding to the STATUS, i.e. OK, WARNING etc., the script from the previous section would then look as follows:

```
xyzAtlasExampleDeviceMixedDuAlternativeFsm_initialize(string domain, string device)
{
  dyn_string nameComponents = strsplit(device, '/');
  string ping = nameComponents[dynlen(nameComponents)]+"OPC.ping:_online.._value";
  string value1 = device+".value1";
  string alert = device+".value2:alert_hdl.._act_text";
  string flag = device+".flag";
  string nodeName = fwFsmAtlas_getNodeDPEName(domain, device);
  dpConnect("AtlasExampleDeviceMixedDUAlternative_callback", nodeName, value1, alert, flag,
ping);
}

xyzAtlasExampleDeviceMixedDUAlternativeFsm_callback(string nodeName, string tnode, string
dpe1, float value1, string alertDPE, string alert, string flagDPE, bool flag, string
pingDPE, bool ping)
{
  string domain, device;
  fwFsmAtlas_getNodeNameComponents(nodeName, domain, device);

  // determine and set State
  ...

  // determine Status
  //
  string targetStatus = "OK";

  targetStatus = alert;
  fwFsmAtlas_setStatus(domain, device, targetStatus);
}
```

Instead of using the `alert_hdl.._act_text` one can of course use other alert config attributes, such as `alert_hdl.._act_state_color`.

STATUS Object Type for Devices

A logical object type has to be defined which implements the STATUS for device units. It is installed as type *ATLAS_DU_STATUS* together with the *AtlasFsmExample* component.

Control Unit Types

High-Level CUs

The top level and the partition CUs must use the simple set of states READY, NOT_READY, SHUTDOWN, UNKNOWN and the corresponding commands GOTO_READY, GOTO_SHUTDOWN, RECOVER. Their states directly depend on the states of their children within the hierarchy which is implemented using the DEN generating SMI-logic

scripts (“when-conditions”). If you don’t inherit your CU type from the supplied ATLAS_CU type, please carefully check the state conditions. Note some pitfalls here:

- If the FSM of a CU child is contained in a distributed system it may not run and will be in DEAD state. This special state must be included in the logic of the parent CU type, i.e. the CU STATE should be set to UNKNOWN.
- In the hierarchy, every CU will have at least one child which is of type ATLAS_STATUS such that any when-condition which check for “ALL FwChildren” will also consider the STATE of the STATUS object (OK, WARNING, ERROR or FATAL). This means that statements like

  ```
  when ALL FwChildren in_state READY then ...
  ```

  will not have the desired effect since there is no READY state of the STATUS object. Instead a statement like

  ```
  when ALL children of type ATLAS_CU in_state READY then ...
  ```

  should be used.

The actions defined for each individual state have to be implemented using the DEN, either by using the wizard or editing the action script directly. For CUs, only command propagation to its children is allowed.

Device-Level CUs

CUs which have devices as children must use a different CU type since the STATEs of the DUs don’t match the STATEs of the controlling CU, resulting in different when-conditions. As above, take care of the DEAD state if the CU and its children reside on different WINCCOA systems and remember that the STATUS objects are children of the CU.

STATUS Object Type for Control Units

As for the STATUS object for devices, a logical object type *ATLAS_STATUS* is installed with the *AtlasFsmExample* component for use as STATUS object for CUs. It lacks the possibility to set the STATUS of the CU from outside the object. That means that the STATUS of CUs can only change if there is a change STATUS of any child.

Setting the STATUS Depending on the STATE

In some cases the severity of a problem can depend directly on the operational mode of a different system. For example depending on whether the STATE of a HV device is ON or OFF, different severities can be attributed to a problem related with a cooling system. Let’s assume the example shown in Figure 10.

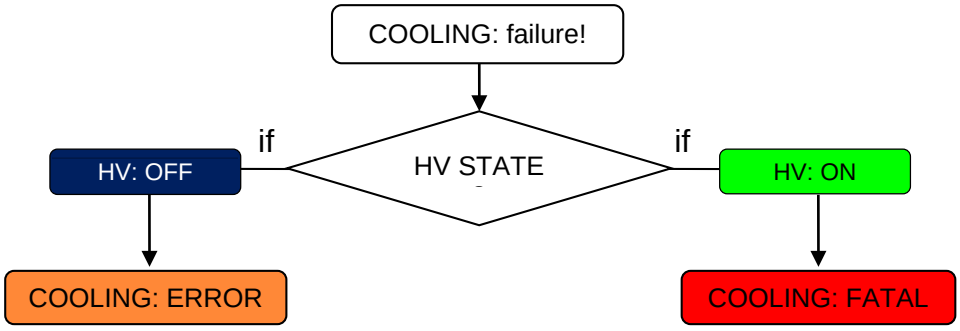


Figure 10: Interdependence of STATUS and STATE of different units.

To implement this dependence STATUS-STATE, the STATE of the HV CU/DU can be queried in the callback script corresponding to the COOLING DU. In order to check the HV STATE the developer can use one of the following functions:

- `fwCU_getState(string node, string &state)`: if the HV state is obtained from a CU.
- `fwCU_getObjState(string node, string obj, string &state)`: if the HV state is obtained from a LU or a DU inside a CU domain.

A.3. Creation of the FSM Hierarchy Tree

To actually create the hierarchy shown in Figure 7 with instances of the object types discussed in the last section, the recommended method is to write a tree creation script. Use the function fwFsmAtlas_createNode() to create an individual object. It will create the associated STATUS object automatically:

```
// fwFsmAtlas_createNode(parent, name, type, label="", panelName = "",
objectFlag=1, isReference=false, system = "", checkExists = false)
//
//   parent:       name of the parent object
//   name:         name of the object, if it is a reference use
"domain::object"
//   objectFlag:   0 - LU, 1 - CU, 2 - DU
//   isReference:  the object to be added is a reference
//   system:       for cross-system references: system name where the
referenced object is located
//   checkExists:  if node is already existing, do not create it
```

Example for the creation of a sub-detector tree with one partition contaning one sub-system and one sub-system device:

```
fwFsmAtlas_createNode("FSM", "ATL_XYZ", "ATLAS_CU", "ATLAS_XYZ", 1,
false, "", true);

fwFsmAtlas_createNode("ATL_XYZ", "XYZ_PARTITION1", "ATLAS_CU",
"XYZ_PARTITION1", 1, false, "", true);

fwFsmAtlas_createNode("XYZ_PARTITION1", "PARTITION1_HV",
"ATLAS_CU_DEVICE", "PARTITION_HV", 0, false, "", true);

fwFsmAtlas_createNode("PARTITION1_HV",
"AtlasExample/AtlasExampleDevice1", "AtlasExampleDevice",
"AtlasExampleDevice", 2, false, "", true);
```

In case you want to create a small test tree you can use the DEN to create it and perform the following steps (not recommended):

Control Units

1. Add an object of type *ATLAS_CU* as child of the root-node of your system and name it ATLAS_TOP1:

   `[In the FSM panel of the DEN select and right-click on the topmost node]` ⇒ *`Add…`* ⇒ *`Objects`* ⇒ *`Add New Object`* ⇒ `[Choose type, enter name]` ⇒ `check` *`as Control Unit`*`].`

2. Repeat step 1. for ATLAS_PARTITION1 and 2 as children of ATLAS_TOP1 (`right-click on ATLAS_TOP1`), and for ATLAS_CU1 and 2 as children of ATLAS_PARTITION1.

Device Units

The devices must exist in the hardware or logical view of the DEN in order to create a DU instance. If this is not the case, refer to the framework documentation on how to create a fw device type.

3. Add all devices of their types as children of ATLAS_CU1:

   [In the FSM panel of the DEN select and right-click on ATLAS_CU1] ⇒ *Add…* ⇒ *Devices* ⇒ *Add Device(s) from Logical/Hardware View* ⇒ [Choose type e.g. *AtlasExampleDeviceDU*, select devices to add] ⇒ ↵, leave *as Control Unit* unchecked].

STATUS Units

4. Add a logical object of type *ATLAS_STATUS* as a child of each CU:

   [In the FSM panel of the DEN select and right-click on the CU, e.g. ATLAS_TOP1] ⇒ *Add…* ⇒ *Objects* ⇒ *Add New Objects* ⇒ [Choose type *ATLAS_STATUS*, enter name *STATUS_*<CU name>] ⇒ ↵, leave *as Control Unit* unchecked].

   See Figure 11 for the last dialog to confirm.

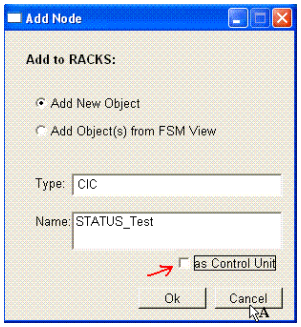


Figure 11: Adding the STATUS object. The checkbox "as Control Unit" must remain unchecked.

5. Add a logical object of type *ATLAS_DU_STATUS* for each DU as a child of the CU which is the parent of the respective DU, here as a child of ATLAS_CU1. Perform the same steps as in 4. but with the different type and the name of the DU instead of the CU name.

6. Add references of the STATUS objects to all STATUS objects one level above in the hierarchy, e.g. add a reference of STATUS_ATLAS_CU1 and 2 as a child of STATUS_ATLAS_PARTITION1:

   [In the FSM panel of the DEN select and right-click on the parent STATUS object, e.g. STATUS_ATLAS_PARTITION1 ⇒ *Add…* ⇒ *Objects* ⇒ *Add Objects(s) from FSM View* ⇒ [Choose type *ATLAS_STATUS*, select object to add] ⇒ ↵, leave *as Control Unit* unchecked].

   References to the STATUS objects of the DUs have to be childs of the parent CU STATUS object as well, i.e. childs of STATUS_ATLAS_CU1 in the example.

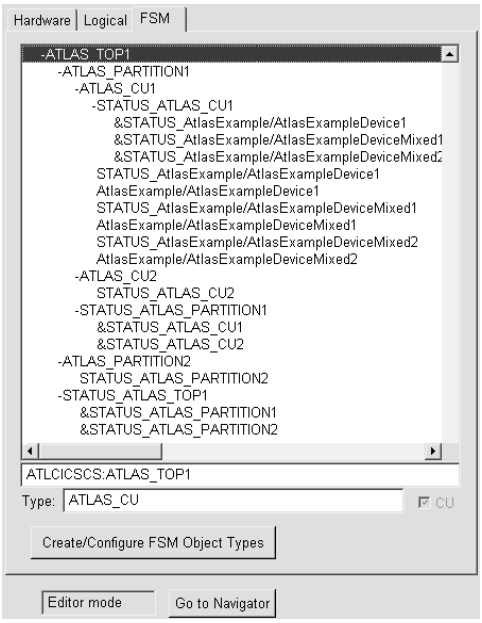


Figure 12: Example hierarchy as it appears in the *DeviceEditorNavigator*.

The final hierarchy as it should appear in the DEN is shown in Figure 12.

A.4. Useful Functions

fwFsmAtlas_startTimeout: This function should be used for each command in the device unit action script. With this function, a time-out trigger is inserted in order to limit the time to switch from one state to another. Further it is possible to switch to the "UNKNOWN" state in case the target state or any new state change does not occur within a given delay.

```
if (command == "RAMP_UP")
{
	/* fwFsmAtlas_startTimeout(delay,domain,device,errorState,targetState); */
	fwDU_startTimeout(10,domain,device,"UNKNOWN","ON");
}
```

fwDU_getAlarmLimits: The Alert handling thresholds of a certain DP can be re-used to define the Device Unit thresholds. This function should be *only* used for those small numbers of DPs which change the alert handling configuration during operation. The use of this function in the DU script significantly decreases performance.

```
FwElmbAi_valueChanged( string domain, string device, float value, string &fwState)
{
	dyn_float limits;
	fwDU_getAlarmLimits(device, "value", limits);
	if	(value < limits[1])
		fwState = "OK";
	else if ( ( limits[1] <= value ) && ( value < limits[2] ) )
		fwState = "WARNING";
	else
		fwState = "ERROR";
}
```

fwFsmAtlas_openPanel: This function permits to open any panel associated to a certain object in the FSM hierarchy (For more information see Appendix B).

```
fwFsmAtlas_openPanel(string node, string obj, string topObj, bool isMainPanel)
```

fwCU_connectObjState: During the preparation of the User Interface panels, if one is interested in knowing the STATE and STATUS from a certain node one can use this function.

More functions in: http://lhcb-online.web.cern.ch/lhcb-online/ecs/fw/FW_FSM.HTML

► User support is only provided for functions documented in the web page.

Appendix B. Building the Operator Interface

B.1. Panel Organization

All main panels and secondary panels associated with a certain FSM node must follow the naming convention (see WINCCOA Panel Names).

When organizing your FSM panels in your own system and exporting them to other distributed systems you must include them in two folders called:

- "panels/fwAtlasMainPanels": panels to be displayed in the main module.
- "panels/fwAtlasSecondaryPanels": panels to be displayed in the secondary module.

This process needs first to be done locally on the system during debugging. Later the final production panels will be exported to the ATLAS central repository.

B.2. Including your panels

At this point your main and secondary panels have been created taking into account:

- The standards panel dimensions.
- FSM panels naming convention.
- Organization of panels into folders.

To include your panels into the common ATLAS Operator Interface (see Figure 4) the following steps have to be done.

1. To attach a main panel to a certain FSM node one must follow the normal procedure. In Editor mode, select a node, right click → Settings → Chose inside the folder "FSMmainPanels" the panel that corresponds to the selected node (if the naming convention has been followed the name should be the same, see Figure 13).

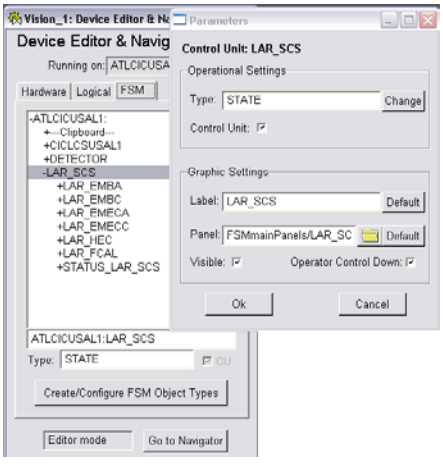


Figure 13: Setting a main panel to a FSM node.

2. The secondary panels are opened from the FSM module relying on the naming convention. The secondary panels use the same name as the main panels adding the suffix "_info".

B.3. Panel Creation and Common Widgets

The look & feel aspect becomes of primary importance when integrating the different sub-detectors. Figure 14 shows an example panel integrated into the FSM operator screen.

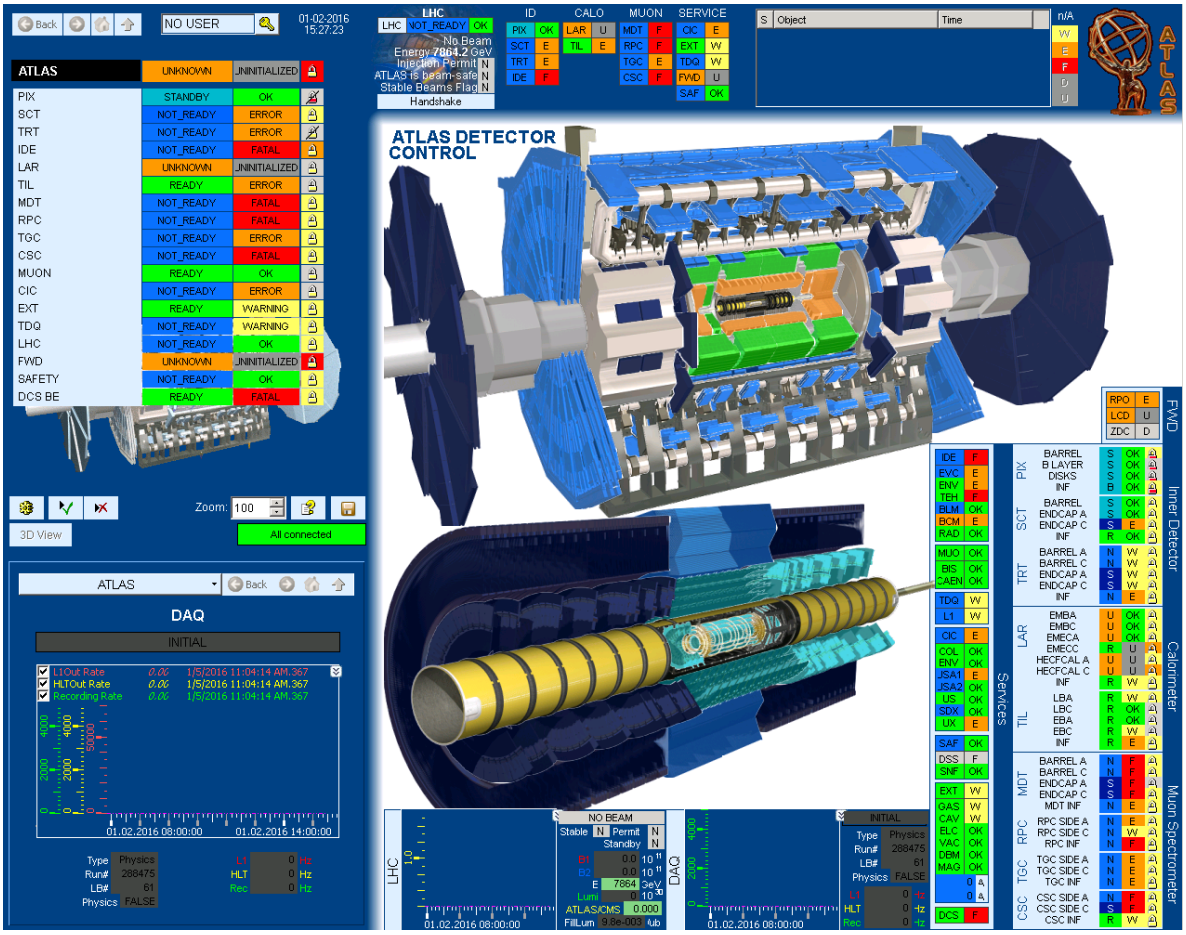
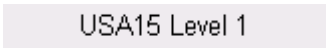


Figure 14: Example of a panel layout.

In order to facilitate the work to the developer and the understanding of the panels by the final operators in shift a set of reference panels (widgets) has been created which can be found in the path: panels/objects/fwFsmAtlas. The individual widgets are documented in the following.

Object Link Button

Can be used to navigate to an arbitrary FSM object on left-click. Right-click navigates to the target in the secondary module.



File: `objects/fsmAtlas/fwObjLinkButton.pnl`

Dollar parameters:

`$label`	`string`	label shown on the button
`$target`	`string`	target FSM object, `<domain>::<object>`
`$width`	`int`	width of the button in # of pixels

Control Unit Widget

Similar to the control field within the FSM module. The label has the same functions as the Object Link Button.

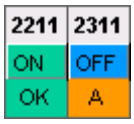
File:
`objects/fsmAtlas/fwFsmAtlas_cu.pnl`

Dollar parameters:

`$label`	`string`	label shown on the button
`$domain`	`string`	domain name of the FSM object
`$obj`	`string`	object name of the FSM object

Device Unit Widget

Same as the Control Unit Widget but without control lock and vertically arranged. The label has the same functions as the Object Link Button.



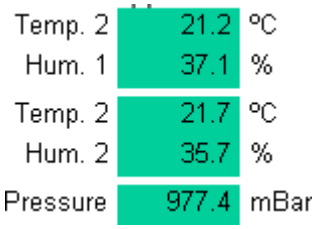
File: `objects/fsmAtlas/fwFsmAtlas_rack.pnl`

Dollar parameters:

`$label`	`string`	label shown on the button
`$domain`	`string`	domain name of the FSM object
`$obj`	`string`	object name of the FSM object
`$shortenStatus`	`int`	default: 0, set 1 if short form of STATUS should be used, e.g. “W” instead of “WARNING”
`$width`	`int`	width in # of pixels

Parameter-Value Widget

Widget to display arbitrary parameter/value pair connecting to a specific DPE. Hovering over the parameter name shows the name of the DPE as tooltip. If alert handling is defined and active for the DPE, the value background is set to the current alert color, and the tooltip shows the alert text. A right-click on the value opens the value trend in a small child window.



File: `objects/fsmAtlas/parameter.pnl`
`objects/fsmAtlas/parameter_small.pnl`

Dollar parameters:

`$parameter`	`string`	parameter label
`$dpe`	`string`	full DPE name, remark: don’t forget system name!
`$format`	`string`	WINCCOA format string, e.g. “[3.2f,,,]” for a float with 2 decimals, see WINCCOA documentation
`$unit`	`string`	optional unit name
`$color`	`string`	text color for parameter label
`$width`	`int`	width of the value area in # of pixels

Appendix C. Interaction with TDAQ Control

Interaction between the Detector Control System and TDAQ is arranged with the assumption that the latter is the master while the former is slave. The TDAQ control applications are capable to send commands for DCS and acquire their results. On other hand, DCS is able to inform TDAQ about the states preventing normal data taking also asynchronously of commands.

In the following, the standard procedure in order to set-up the FSM for an interaction with TDAQ is explained.

At a certain level, the sub-detectors are split into TTC partitions (see Figure 1). The purpose of this partitioning is to allow, the master DAQ, to operate the DCS depending in its own partitions.

It is foreseen that the number of DDC controllers running for a sub-detector should correspond one-to-one to the number of TTC partitions of that sub-detector. Similarly, should exist one FSM domain per TTC partition, and one DDC DU per FSM domain (see Figure 21). Thus, within each TTC domain there is one Device Unit interacting with a certain DDC controller.

(1 TTC partition) x (1 DDC controller) x (1 TTC FSM domain) x (1 DDC DU)

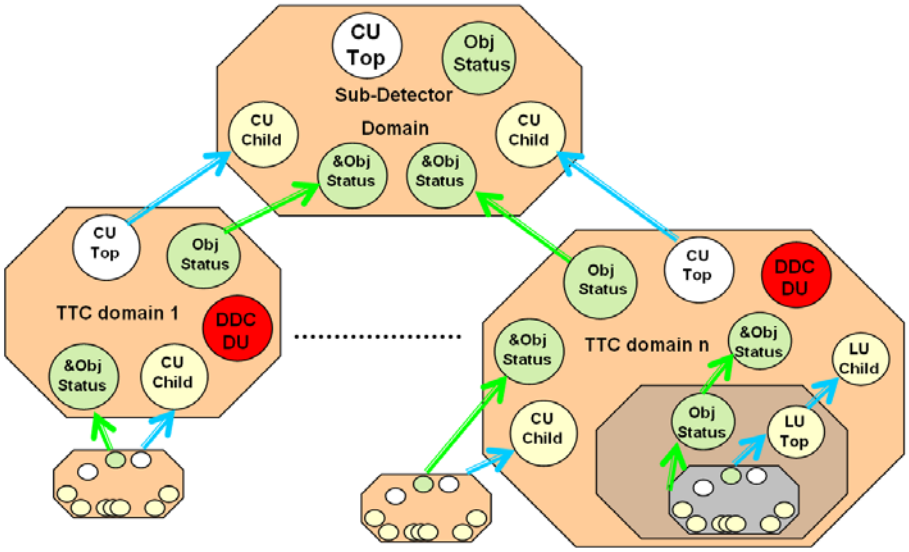


Figure 15: One DDC DU within each TTC FSM domain

During the installation of the “fwFsmAtlas” package, a DP Type as well as a Device Unit Type, both called *FwFsmAtlasDDC*, is created (see Figure 22). The main duties of this Device Unit are:

1. Reporting the DCS state. Asynchronously it must report to TDAQ any occurrence of conditions preventing the data taking. The granularity is the TTC partition. Thus, the DDC Device Unit checks the state of its TTC domain and sets a flag (DPE: “notDataTaking”) that reports the state of the detector for a certain TTC partition.

   The standard FSM states for a TTC partition are:

   - READY : TTC partition ready for data taking
   - STANDBY/NOT_READY/SHUTDOWN: TTC partition not ready for data taking.

► Note: Be consistent when propagating the states upwards in the hierarchy to the TTC level. If the DCS is not READY, it means that the whole TTC partition is not ready for data taking!!!

2. TDAQ Command Execution. TDAQ can operate the DCS executing transition commands by means of the FSM. The set of FSM commands to be issued by TDAQ are meant to be pretty general (i.e. “PREPARE_FOR_RUN”).

   When a command is triggered (DPE: “trigger”) from the TDAQ, the DDC Device Unit reads a set of parameters and it sends the selected command to the selected node. The parameters (DPE: “fsmParametrs”) are filled by TDAQ in the following format:

   ```
   FSM_domain|FSM_command|time_out
   ```

   being the “FSM_domain” and the “FSM_command” mandatory fields.

► In case the command is sent to a single DU (which should not be the normal case) the device object must be specified in the DPE “parameters”.

The response (DPE: “response”) to the command execution depends on the timeout:

- If the timeout value is 0, the response is set to a good state automatically.
- In case of an existing timeout, the response is set within the timeout interval to either a good or bad state. The response depends on the transition caused by the command execution.

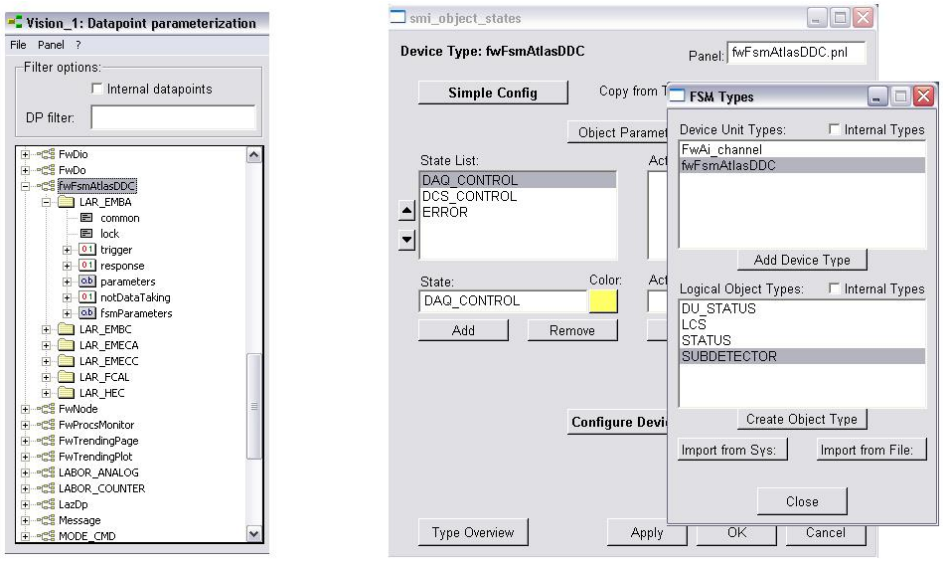


Figure 16: FwFsmAtlasDDC

C.1. Work in the WINCCOA side

The developer has to create a data point of type “FwFsmAtlasDDC” for each TTC partition (DDC controller). The DDC controller, together with the FSM DU, will be in charge of R/W of this data point.

C.2. Work in the FSM side

The only work to do in the FSM side is to insert the DU previously created in its corresponding TTC FSM domain. The DU unit type is already edited with the functionality explained above. Thus, if this functionality is enough for the sub-detector no extra work needs to be done.

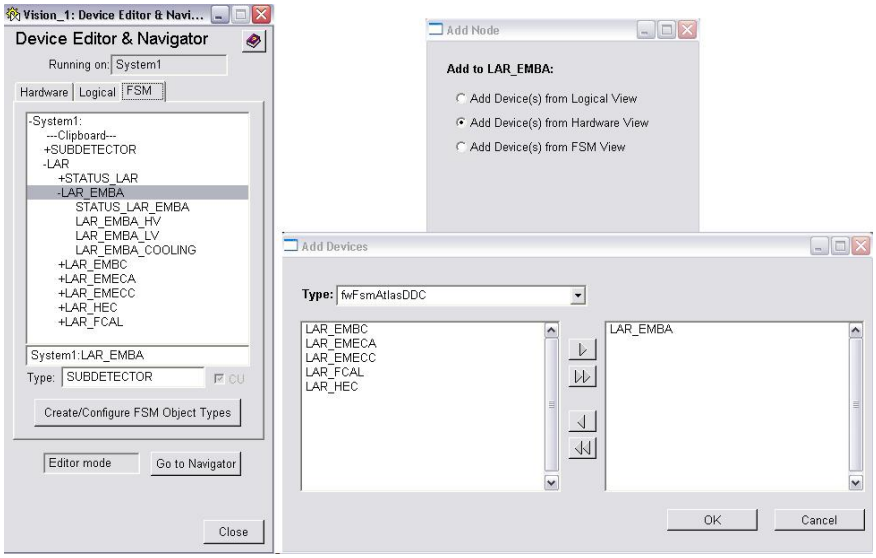


Figure 17: Adding the DDC device unit.

A detailed description of the design and actual implementation of the DAQ-DCS communication can be found in [1] and [2].

Appendix D. fwFsmAtlas

The *fwFsmAtlas* component is released with the common versioning scheme `fwFsmAtlas-<major#>-<minor#>-<patch#>`. Each release will correspond to a tag with the same name (e.g. `fwFsmAtlas-1-2-3`) of the module inside the *atlasdcs* SVN repository (`SVNROOT=svn.cern.ch/reps/atlasdcs`). Recent releases can be found under https://twiki.cern.ch/twiki/bin/view/Atlas/DcsSoftware#ATLAS_Finite_State_Machine. You can check out a release directly from SVN or install it directly from the Point1 repository at `/det/dcs/fwComponents/fwFsmAtlas-?.?.?`.

Appendix E. References

[1] ATLAS DAQ – DCS Communication Software. User's Guide
https://edms.cern.ch/file/684955//DDC_UG.pdf

[2] Subdetector Controls Interaction with TDAQ Controls