FESA PRIMER

A Simple and Fast Approach
from a User's Point of View

FESA Version 2.9
3/2007

DRAFT

TABLE OF CONTENTS

1. Preface and Requirements

The implementation of front end devices of any type such as PLCs, VME modules, CCD cameras and so on, into the CERN control system is not an easy task. As the number of devices at CERN is unpredictably high, the control group decided not to do this job in every case, but developed a framework in which every user of a device is able to implement it himself. This framework is called **FESA**, the **F**ront **E**nd **S**oftware **A**rchitecture, and the principle and manner of working within this framework is described in this document from a user's point of view.

The main goal using this framework is to receive an executable program (device class), which is running on one or many user's front-end CPUs (FEC) or PCs and which performs all of the required tasks, such as getting and setting data within an adequate time frame. FESA helps with all intermediate steps regarding the timing connection, deployment and instantiation of the class. Not provided within FESA is a final graphical user interface, but a flexible test environment to check all functions of the class.
The advantages using this framework are manifold. The most important of them shall be listed here to motivate the new FESA user:

- it provide for reuse of code, which saves plenty of developing time. This code is already debugged and tested.
- the interfaces with the next upper level (middleware) are uniform and instantly accepted.
- changes and debugging are easier to handle, by the user, by others, and also up to 10-15 years in the future.
- the amount of programming (in byte and time) is exceedingly reduced due to automatic code generation.
- CVS (Concurrent Versions System) based source code management for version control, safety, and multiple developers.
- the growing FESA community provide for additional support, tips, and tricks

The FESA framework has to be used for all new devices being added to the control system. In case of problems with FESA the experts from the FESA project team are willing to assist. Exceptions to this regulation have to be discussed with the respective people in the CO-group. As with any other way of implementing devices into a control system, there are some requirements, recommendations, and restrictions:

1.1. User

- The user is a "device expert". That is, he knows what he wants to do with his device and how it works.
- The user has basic knowledge of the CERN timing system and of the machine and event names he will need to trigger his equipment.
- A hardware driver is required.
- The work with FESA requires a basic knowledge of C++.
- The user needs some time and patience to learn FESA.
- He should carefully read all existing information, which already exists on the FESA project website:

http://cern.ch/project-fesa/

The "FESA Essentials" [1] especially give a compact overview of FESA. Its contents will be partially repeated in this document.

1.2. Infrastructure

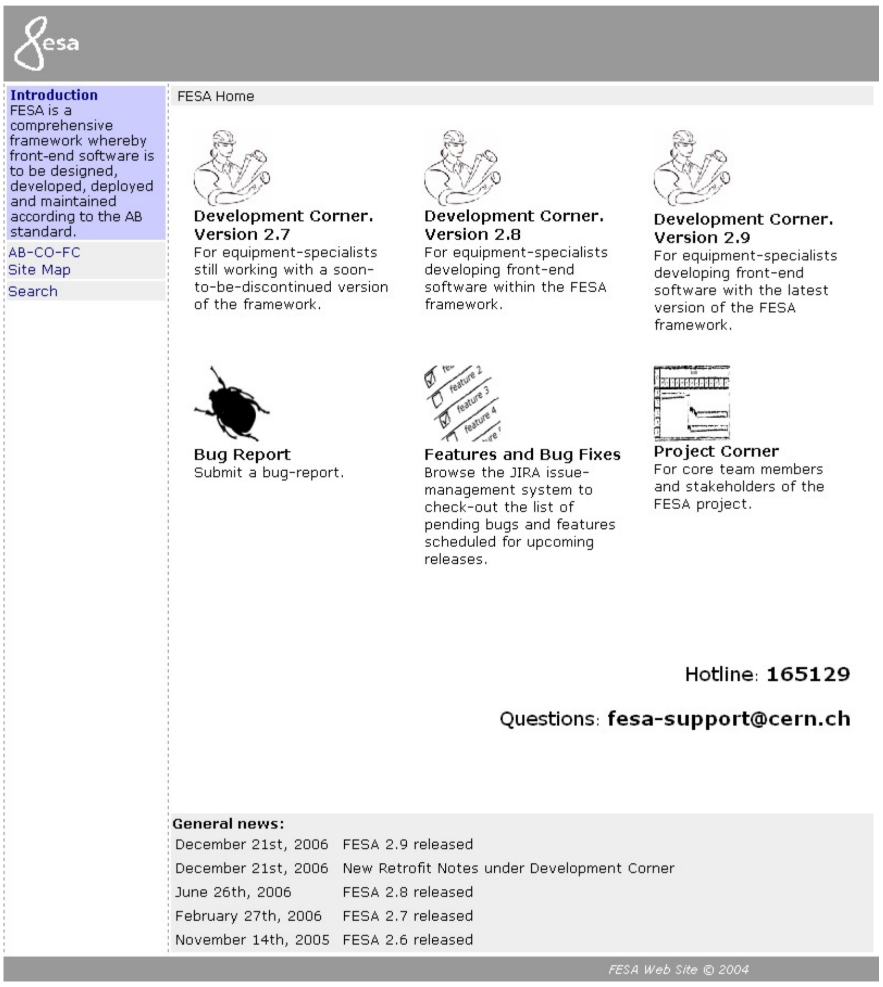
- FESA is not available from outside the CERN intranet.
- The graphical development tools of FESA are realized in JAVA, therefore Windows, Linux, and Mac operating systems are supported. The actual Java Runtime Environment must be installed on your system.
- For the source code development, an appropriate Linux Account is required.
- The supported FECs are: Motorola, PPC, Intel
- FESA device classes can run on Linux and LynxOS.

This document does not describe every button and functionality, only those which were found to be important. Play around with all the icons and menu entries to find all the features. The FESA development is permanently continued (2.9 at the moment), so parts of this document may no longer be valid. Please report mistakes to the FESA team.
The words formatted in *italics* shall point to its FESA origin.

2. The FESA Project Website

2.1. Main Page

Open the FESA homepage at http://cern.ch/project-fesa



Introduction
FESA is a comprehensive framework whereby front-end software is to be designed, developed, deployed and maintained according to the AB standard.

AB-CO-FC
Site Map
Search

FESA Home

Development Corner. Version 2.7
For equipment-specialists still working with a soon-to-be-discontinued version of the framework.

Development Corner. Version 2.8
For equipment-specialists developing front-end software within the FESA framework.

Development Corner. Version 2.9
For equipment-specialists developing front-end software with the latest version of the FESA framework.

Bug Report
Submit a bug-report.

Features and Bug Fixes
Browse the JIRA issue-management system to check-out the list of pending bugs and features scheduled for upcoming releases.

Project Corner
For core team members and stakeholders of the FESA project.

Hotline: **165129**

Questions: **fesa-support@cern.ch**

General news:

December 21st, 2006	FESA 2.9 released
December 21st, 2006	New Retrofit Notes under Development Corner
June 26th, 2006	FESA 2.8 released
February 27th, 2006	FESA 2.7 released
November 14th, 2005	FESA 2.6 released

FESA Web Site © 2004

Fig. 1: FESA main page

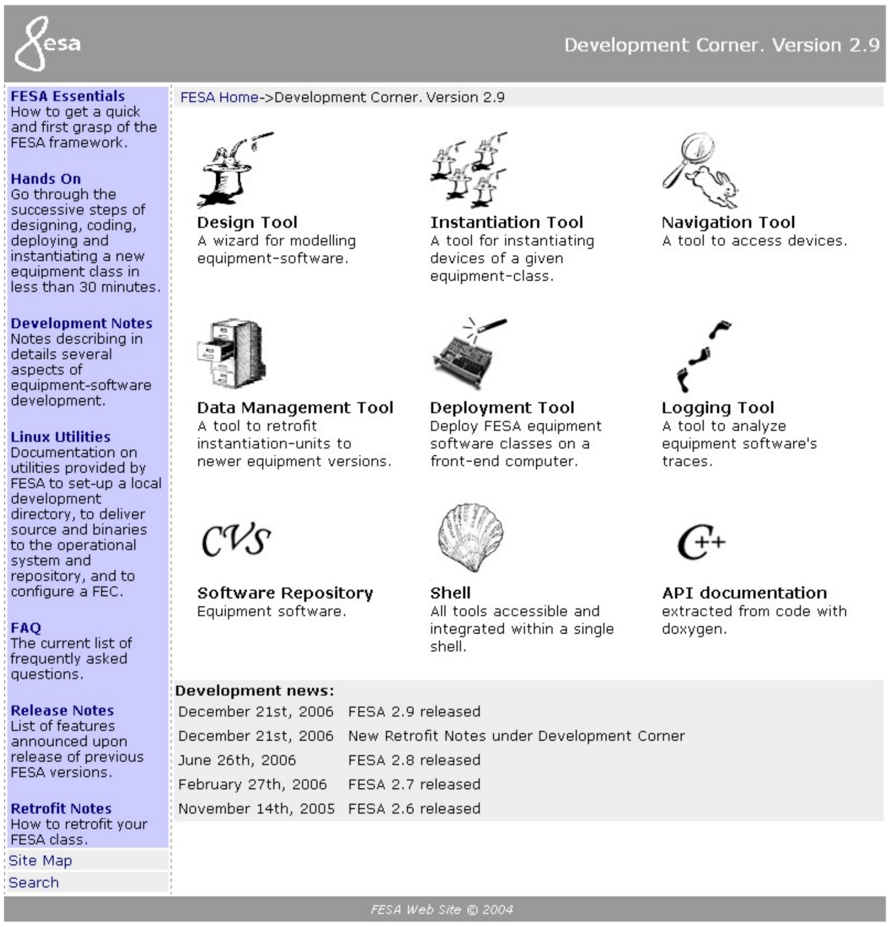
The three links labeled “Development Corner Version X.X” lead to the collection of Java tools required for FESA. For new projects the latest version should be chosen. It is recommended that users update their compilations with the latest version, as older versions (latest – 3) are not supported anymore.

The link named “Bug Report” offers the possibility of sending an email to the FESA team in case of problems with FESA. Recommendations and comments are also accepted.

Following the link “Features and Bug Fixes” gives information about all active bug reports and upcoming features. The JIRA documentation provides the possibility to track a (your) bug in a very convenient and transparent way.

The access to the link “Project Corner” is permitted only to the FESA developers.

2.2. Development Corner



Development Corner. Version 2.9

FESA Home->Development Corner. Version 2.9

FESA Essentials
How to get a quick and first grasp of the FESA framework.

Hands On
Go through the successive steps of designing, coding, deploying and instantiating a new equipment class in less than 30 minutes.

Development Notes
Notes describing in details several aspects of equipment-software development.

Linux Utilities
Documentation on utilities provided by FESA to set-up a local development directory, to deliver source and binaries to the operational system and repository, and to configure a FEC.

FAQ
The current list of frequently asked questions.

Release Notes
List of features announced upon release of previous FESA versions.

Retrofit Notes
How to retrofit your FESA class.

Site Map
Search

Design Tool
A wizard for modelling equipment-software.

Instantiation Tool
A tool for instantiating devices of a given equipment-class.

Navigation Tool
A tool to access devices.

Data Management Tool
A tool to retrofit instantiation-units to newer equipment versions.

Deployment Tool
Deploy FESA equipment software classes on a front-end computer.

Logging Tool
A tool to analyze equipment software's traces.

Software Repository
Equipment software.

Shell
All tools accessible and integrated within a single shell.

API documentation
extracted from code with doxygen.

Development news:

December 21st, 2006	FESA 2.9 released
December 21st, 2006	New Retrofit Notes under Development Corner
June 26th, 2006	FESA 2.8 released
February 27th, 2006	FESA 2.7 released
November 14th, 2005	FESA 2.6 released

FESA Web Site © 2004

Fig. 2: Development Corner

This page provides all necessary tools and available documentation to create FESA device classes. The use and function of these development tools are described in detail within Chapter 3.
The links “Design-Tool”, “Instantiation-Tool”, “Navigation-Tool”, and “Deployment-Tool” are all combined in the link “Shell” for easy access.
The “Data Management Tool” is used to delete instances from a FEC and to switch to another software version of your device class on the FEC.

The “Logging Tool” is not completely functional and should not be used.

The “CVS” (Concurrent Versions System) link gives read access to the complete FESA device class repository. Here you are able to study source code and to get an overview about the different versions of device classes.
The “API Documentation” is a helpful Doxygen based reference for FESA users in the programming phase.

All available documentation on FESA is listed in the left sidebar.

3. Principle and Technical Terms

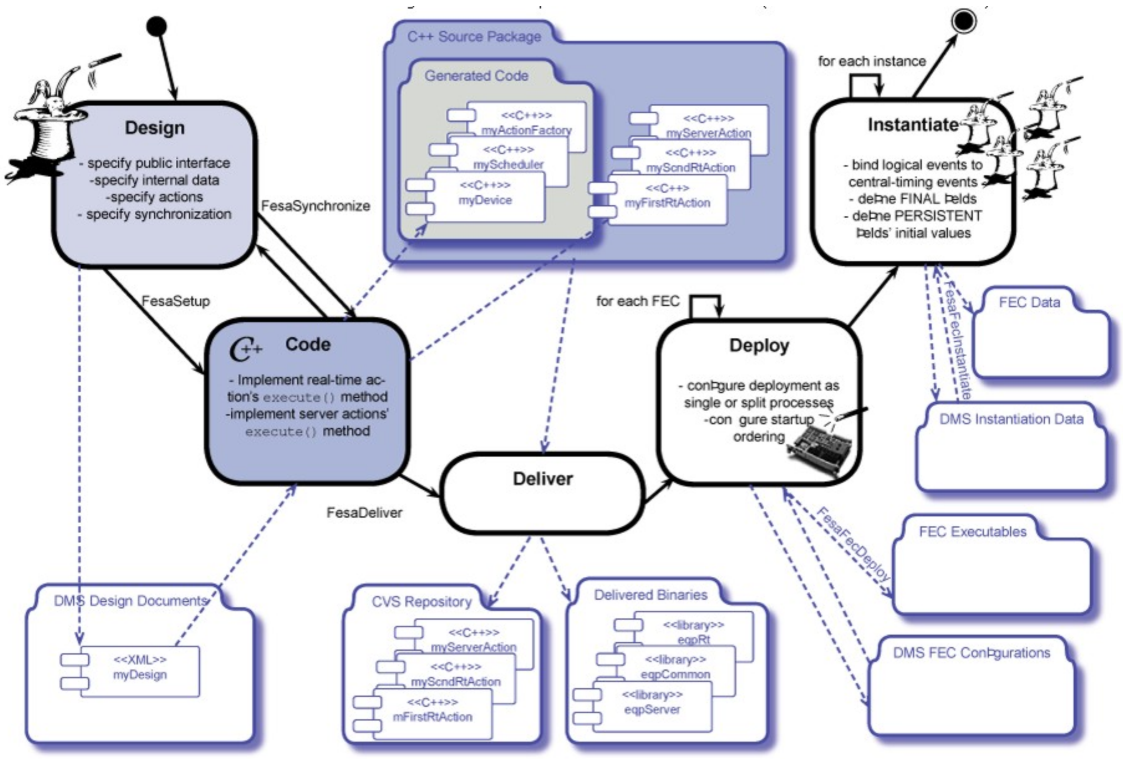


Fig. 3: Schematic overview of the FESA development workflow

The flow of a FESA class development starts with the design. Herein you specify the inputs and outputs of data, as well as the actions and timing. The conclusive and valid design forms the foundation of the C++ source code generation, which is performed automatically. The programming of the event-and-user driven actions is the next step and has to be done by the user. The source code is delivered to the CVS repository and the compiled executable and its associated files are deployed to the relevant FECs. Finally all required instances of the device class are generated and the class is ready to be tested using the “Navigation-Tool”.

3.1. Design

The design phase is the first step on the way to creating a FESA class. To create an equipment model (FESA class) the “Design-Tool” is used, which is available on the “Development Corner” as a stand-alone Java tool or as part of the “Shell”.

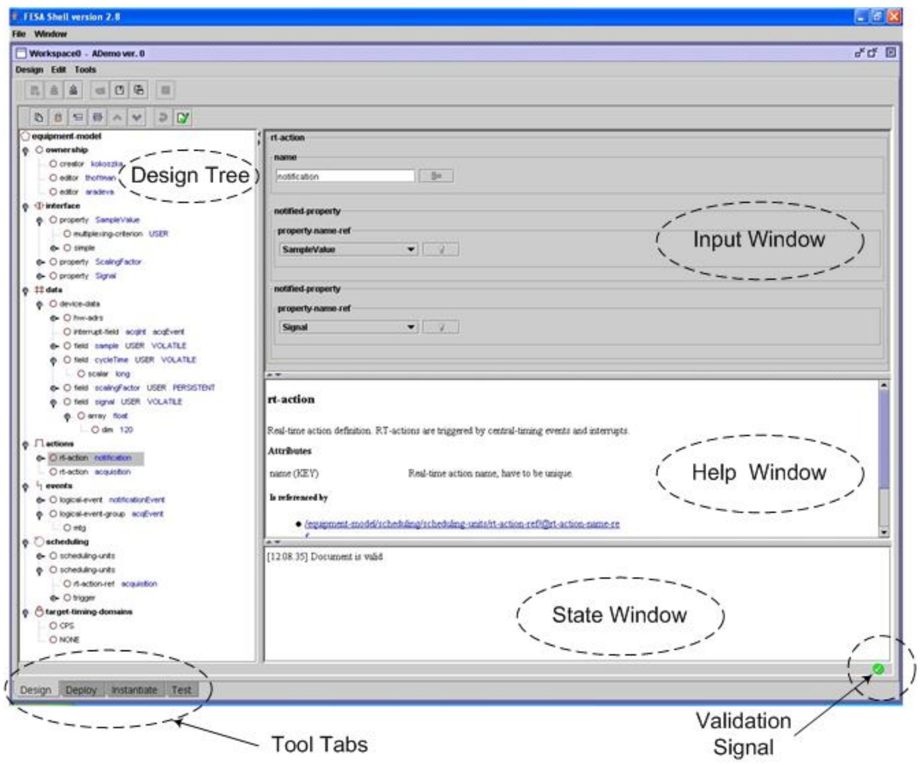
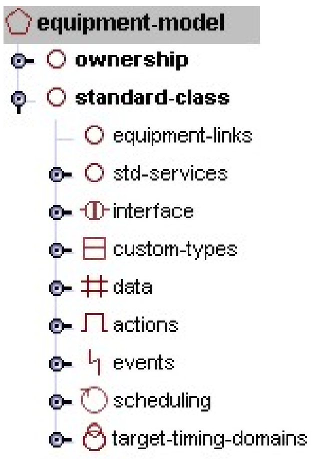


Fig. 4: Screenshot of the Design-Tool within the “Shell”

To start a new design, click “New” in the menu or press the accordant icon. You may then select a template with preconfigured settings, e.g. the full template.



This tree lists all available design parts, which are described in detail below. It is not necessary to use them all; the usage depends on the type of the class. To create a valid design, inter-dependencies between different design parts have to be solved. If not, the “Design-Tool” gives warnings and signals, e.g. red fonts at missing inputs, instantaneous at design time.

In general, if the design is valid, a ✓ appears in the bottom right corner. In fault state the warning symbol ● shows up.

Tree 1: Equipment model

After creating and editing the design, it has to be stored in the FESA class database by selecting “Store” in the menu. At the first save, a significant name has to be entered. FESA handles the user inputs in XML files. It is possible to store and open these files separately.
While designing you may use the right mouse button to add or replace features. Some entries are restricted by naming conventions; the correct pattern is then displayed in the state window.

3.1.1. Ownership

The creator and the editor(s) are entered here with their typical account name. This information is required for access control of the design.

3.1.2. Standard-class / Plc-class

The design branch *standard-class* is used for most equipment designs. In case you design a FESA class for PLC devices you may change the *standard-class* entry to *plc-class*. There are already typical PLC options available (see Chapter 3.7).

3.1.3. Equipment-Links

Usually FESA classes are developed as stand-alone classes. If it is necessary to link separate FESA classes, the “Equipment Links” have to be defined. This can be done to reduce complexity of a class or to connect to classes which are deployed on different FECs.
This special task provides a separate manual with examples. This can be found in the “Development Notes”, the link is reached via the “Development Corner” sidebar.

http://project-fesa.web.cern.ch/project-fesa/development/notes.htm

3.1.4. Std-Services

PLC

Inheriting the standard PLC interfacing service brings several pieces of information into your design. These pieces of information configure the communication-protocol which allows the FEC at one end, and the PLC at the other, to maintain a consistent state of the device they each see. This *std-service* is referenced by a *PLC-*

RTAction. If no PLC-related activities are foreseen, this service should not be set to *plc*.

GM

Selecting the standard GM interfacing service allows you to add old-style General Module properties into your design. This allows legacy C applications to access your equipment through RPC. When you decide to inherit from the GM interface service, RPC-handling code will automatically be integrated into your equipment class. This option has to be set in case a gm-property is defined.

3.1.5. Interface

The *interface* branch of the design tree defines all get-and-set functions providing data exchange from and to the outside (clients from the control-room or middle-tier software layer). Designing the *interface* means listing so-called *properties* that can be remotely accessed through the controls-middleware. You should devote great care to defining your equipment class interface as this can be viewed as the binding-agreement between your class and its external users. When you create a new version of an equipment class it is your responsibility to ensure backward compatibility with Java applications that access its interface.
For performance and usability reasons the interface should be simple and short. This can be accomplished by combining similar value-items to one property.

3.1.5.1. Properties

A set of predefined properties is available. These are:

alarm-events-property
The *alarm-events-property* (since 2.9) is designed to be completely managed automatically. It collects all information from the defined *alarm-fields* and handles the alarm as specified to operate with the LASER system. The attached items to the *alarm-events-property*, such as name-, state-, stamp-, prefix- and suffix-item provide *dim* fields, which have to be filled with the maximum amount of *alarm-fields* specified in the *device-data*. In subscription on-change (at application or middleware level), only *alarm-fields* which have a different state from the previous call will be reported to the client. If nothing has changed, nothing will be sent.

alarm-details-property

The *alarm-details-property* is designed to report dynamic details for a specific alarm. Those details are optional and can be defined individually on each *alarm-field* by adding a *standard-key* and/or a *user-key*. The client, who calls this property, specifies the *alarm-field* name on which he wants details, after which the key values of all keys defined for this particular alarm field will be reported.

std-setting-property
std-acquisition-property
std-reset-property
std-status-property

Note:
“std” is an abbreviation for “standard”, not to be mixed up with the US medical abbreviation for “sexually transmitted disease”. 😊

All the std-properties conform to the standards and guidelines of the applications group [2] and have to be used if the std-property sense matches with your concept. Editing of these properties is very restricted. All other properties are treated as expert properties and have to be adapted separately.

std-copypl-property

This property provides the possibility to copy PLS settings from one to another line. The *value-item* in this specific property is the *line* name (e.g. PILOT1)

property

This is the most common used property, or expert property, where you define its use on your own. Dependent on your requirements and your C++ coding concept, you have to select between simple and complex functionality. These two settings are described on the following pages.

gm-property

As GM is the forerunner model for FESA, this property provides the interface for FESA projects with GM based applications (GUIs).

With your property entry you have to decide whether you wish to perform a default get/set action, which reads and writes on a specified data field (see Chapter 3.1.7) without additional C++ coding (use *simple*), or defining a *complex* property, which then is linked to a custom *Server Action* (see Chapter 3.1.8).

A property can hold one or several so called “items” such as:

value_item:
complex property, can be of all data types (see Chapter 9)

filter_item:
used for data shaping, averaging and other calculations

data-field-ref-item:
standard data field (in: data, device data, field)

data-field-role-item:
like the ref-item, but used to give an alias name

state-field-ref-item:
refers to a state-field and custom-types state-enum

state-field-role-item:
like ref-item but used to give an as alias name

data-field-bit-ref-item:
refers to a single bit in a custom-types bit-enum

error_status_item and
error_message_item:
created and used by the std-status-property

Multiplexing Criterion

With any defined property a *multiplexing-criterion* has to be set to either NONE, which means this type of get or set function for all value-items is valid for all possible timing-domains (accelerators) or to USER, to read from and write to the settings of an exclusive timing-domain. If in a property a data-field-ref-item was set, then this multiplexing-criterion is already known from the data-field settings. For value-items in complex properties, a combo-box provides the selection between USER and NONE.

Simple

In most cases defining a *simple property* is absolutely sufficient. Each individual item maps on an existing field (see Chapter 3.1.7). This characteristic makes it possible to rely on a *default Server Action* (see Chapter 3.1.8) instead of having to supply specific code for serving the property. In addition a connection to a *custom Server Action* is possible via a *server-action-ref* definition.

Complex

A property is said to be 'complex' when the individual *value-item* do not simply map on internal fields (it may be still possible via a *data-field-ref-item*). A complex property may also define a 'filter' which specifies the treatment performed by a custom server-action that serves the property. It is foreseen that the regular *value-item* may be limited by range settings using min and *max*.

3.1.6. Custom-Types

You may rely on a set of custom-defined constants, enumerated types and 16/32-bits bit-patterns in your design. These custom-types are built on top of basic 'underlying types' (e.g. short, long). The controls-middleware has no knowledge of the custom-types and interprets them as their underlying type. Therefore, it is your responsibility either (1) to ensure that such custom-types are used only internally by your equipment, or (2) to make sure that any public property that would refer to the custom-types is restricted to those applications that have knowledge of how to reinterpret the underlying-types used for transmission.

Examples:

(1) Define a *constant*, name it e.g. RESOLUTION, enter a value e.g. *0.005* and select the type *const float.* Use this RESOLUTION within your source code for calculations.

*output = measValue * RESOLUTION;*

The middleware is not interested in this constant. But on change of RESOLUTION you can update its value easily within the design.

(2) Define a *bit-enum-32bits*, name it e.g. CONFIG_REGISTER, open the sub-branch and name all bits separately. In addition you can set startup states to each bit selecting *true* or *false*.
Then define a data field (see Chapter 3.1.7) named *configReg* and create a reference using *custom-type-ref* CONFIG_REGISTER. Now

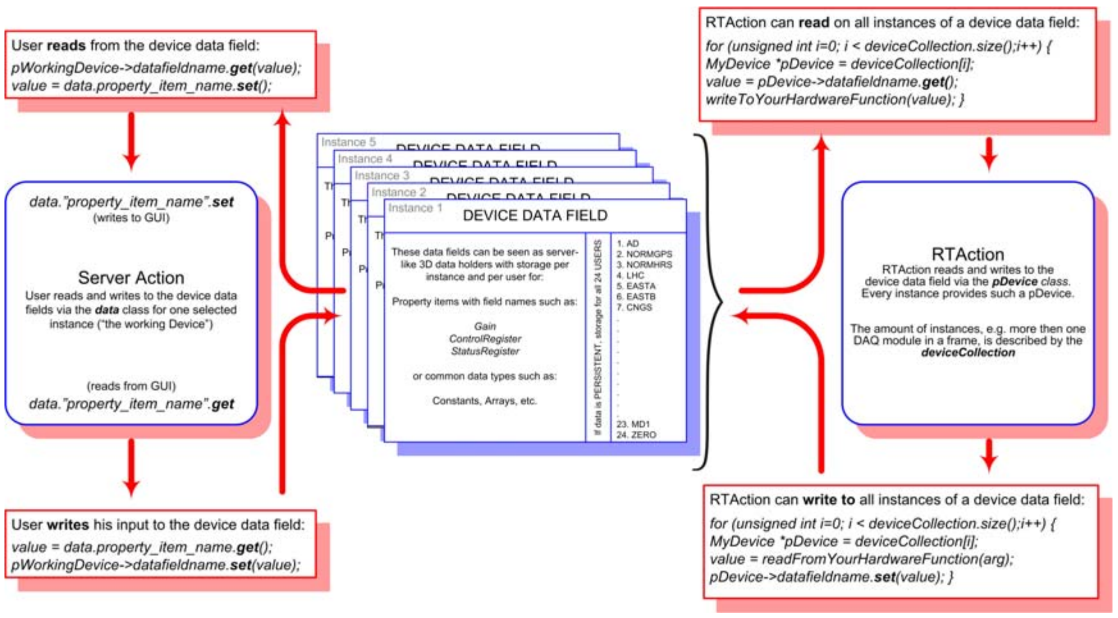
a self-defined property can perform get-and-set actions to this field *configReg*, which is based on a custom type.

3.1.7. Data

The data branch is the reloading point of all handled data and the heart of the device model. It provides the device data, the global data and the domain data branches to define and to allocate the placeholders for all possible values, results, text strings etc. required for the all-embracing interaction between the user and the hardware.
Main purposes of this data field principle are to represent the complete device data model and to buffer user inputs, which are not processed immediately. This method provides some safety as a user typically can not disturb a critical sequence of processes or *RTActions* by a direct hardware access.

3.1.7.1. Device-Data

These fields store settings, acquisitions, parameters and variables of all device-instances and beam users to provide a snapshot of the current state of the device's hardware counterpart, or to hold settings ready which have to be sent to the hardware on a specific trigger.


User **reads** from the device data field:

```
pWorkingDevice->datafieldname.get(value);
value = data.property_item_name.set();
```

User **writes** his input to the device data field:

```
value = data.property_item_name.get();
pWorkingDevice->datafieldname.set(value);
```

RTAction can **read** on all instances of a device data field:

```
for (unsigned int i=0; i < deviceCollection.size();i++) {
MyDevice *pDevice = deviceCollection[i];
value = pDevice->datafieldname.get();
writeToYourHardwareFunction(value); }
```

RTAction can **write to** all instances of a device data field:

```
for (unsigned int i=0; i < deviceCollection.size();i++) {
MyDevice *pDevice = deviceCollection[i];
value = readFromYourHardwareFunction(arg);
pDevice->datafieldname.set(value); }
```

data."property_item_name".set
(writes to GUI)
Server Action
User reads and writes to the device data fields via the data class for one selected instance ("the working Device")
(reads from GUI)
data."property_item_name".get
Instance 5
Instance 4
Instance 3
Instance 2
Instance 1
DEVICE DATA FIELD
These data fields can be seen as server-like 3D data holders with storage per instance and per user for:
Property items with field names such as:
Gain
ControlRegister
StatusRegister
or common data types such as:
Constants, Arrays, etc.
if data is PERSISTENT, storage for all 24 USERS
1. AD
2. NORMGPS
3. NORMHRS
4. LHC
5. EASTA
6. EASTB
7. CNGS
23. MD1
24. ZERO
RTAction
RTAction reads and writes to the device data field via the pDevice class. Every instance provides such a pDevice.
The amount of instances, e.g. more then one DAQ module in a frame, is described by the deviceCollection

Fig. 5: Visualization of the device data field functionality and its access techniques

Figure 5 describes how the data fields are accessed. The detailed explanation of *Server Action* and *RTAction* follows in the Chapter 3.1.8.

A set of different predefined field types is available:

The **hw-adrs** fields provide the possibility to define hardware relevant fields such as lun (logical unit number), ch (channel), or type, which can be set later within the instantiation part. Working on a PLC-Class this field is used to define a *plc-hostname.*

The **fault-fields** hold boolean values. These tell if fault-states are active or not. From within the C++ code, or a real-time or *Server Action*, you raise a fault by setting the corresponding fault-field to 'true'. You suppress the fault state by 'resetting' the field, which means setting its value to 'false'. Fault-fields are used to manage states on the front-end part, e.g. stop a server-action on a faulty state of a register. Working on a PLC-Class, there are already three predefined fault-fields which cannot be changed.

The **alarm-field** is similar to the *fault-field* with a slightly different character as an alarm may be raised also on informative states such as “scintillator moved in the beam”, which is not a fault, but really important to know. For this, the static alarm message (conform with LASER API) may be characterized through a series of optional standard or user-keys. The standard key uses the alarm-field’s name, which is being added either as prefix (in front of the alarm-message) or as suffix (behind the alarm-message) to provide this name information. The user-key can deliver more information placed in its user-key name field. In addition, for an expert rating the severity degree such as ERROR, WARNING, and OK, can be defined. For several sub-items of the *alarm-events-property* the amount of the alarm-fields has to be entered into their *dim* fields. (see also 3.8).

The **state-field** is bound to a *state-enum* variable defined within the *custom-types* branch. These predefined states couple strings such as ON, OFF etc. with numbers to be used in C++ coding.

The **interrupt-field** is required when operating with the event type *logical-event-group*. In the instantiation part this field is coupled with an LTIM/CTIM event.

The **field** is the most commonly used data field to *get-and-set* data from the user to the hardware. Out of a set of different data types, such as 1-dim-arrays, 2-dim-arrays or scalar types, the adequate type may be selected.

3.1.7.2. Global-Data

This type of data is available all over your FESA class and valid for all instances. Global-data fields cannot be used in a multiplexing context.

3.1.7.3. Domain-Data

In the **domain-data** branch, so-called *telegram-group-fields* may be defined. These fields are filled with definitions given in the Instantiation-Tool. The sense behind the *domain-data* fields is to provide specific tgm-data (telegram data within the timing-event) within your *RTAction* source code to be used for target-specific actions at runtime, for example a low intensity beam line (fixed to low intensity) for which a transformer always has to be set to a more sensible gain-range or to obtain the particle type for evaluation purposes.

3.1.8. Actions

Actions are the basic work-units of the equipment software. They come in two flavors: the real-time actions are triggered by central-timing events and interrupts. The *Server Actions* implement user request-handling. Right from the design stage, the equipment specialist has to list all the action-classes that can be executed at any time by user intervention or by triggered events.
The FESA equipment class, your project, can be described as a server. When a client, for example the controls middleware, requests a get-or-set action, this request is accepted by your server and is packaged as an event and transmitted to the *Server Action*. This is similar to all kind of actions described here. All actions provide in their source code the *execute(Event *)* method, which is then executed. This is the method which has to be filled with source code by the user.
The different types of actions and their access directions are shown in Figure 6.

3.1.8.1. Server Action

The ***Default Server Action*** performs a get and/or set operation without user defined source code. This action need not be specified under the design branch *actions*, but in the interface part, where a “simple” property-item may be defined with the get/set **default-**

action. The corresponding data field may then be set or read out automatically on user request. See also Figure 6.

The ***Server Action*** is more powerful when implemented as user-specified code such as text output, calculations, etc. The code frame (a .cpp and .h file) is generated by the Linux command *Fesa Synchronize* and is stored in the SERVER directory. The name of this file equals the name of the defined *Server Action* in the design branch “actions”.
A *Server Action* is primarily timing independent, of course there are ways to use external or user triggers to execute the *Server Action*.

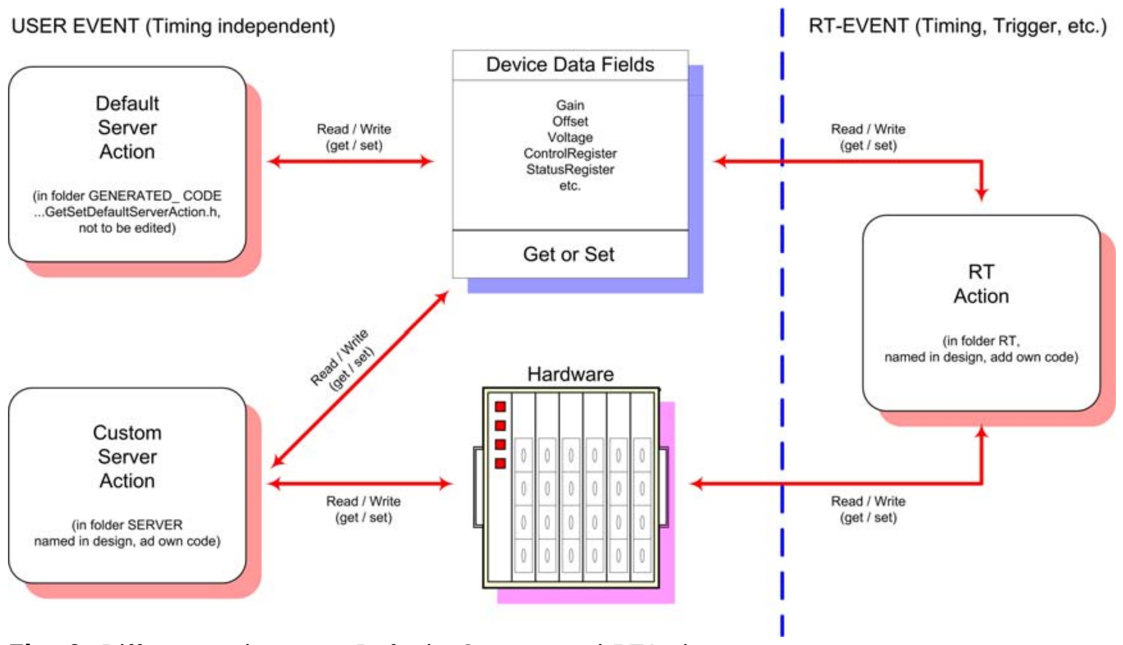


Fig. 6: Differences between Default, Custom and RTAction

3.1.8.2. RTAction

The ***RTAction*** executes user-defined code. On a standard event driven electronic or data acquisition system, the *RTAction* is the part where actions of any type, such as read on register, calculate data, initialize hardware, or move actuator, are performed in time. A started *RTAction* has maximum priority, so new incoming triggers are delayed until the *RTAction* has finished. Also, user-executed *Server Actions* are delayed when an *RTAction* is running.

DeviceCollection

In Figure 5 within the Chapter 3.1.7.1 the so called *DeviceCollection* was already mentioned. As more than one instance (e.g. 2, 3, or more modules of the same type or other certain grouping criteria) might be present in your equipment, and all provide perhaps different settings or deliver different results, they have to be treated separately. For this, the class deviceCollection provides its size and instance number for the *RTAction*. The code of an *RTAction* is enclosed in a device-loop, which leads to an access of every instance per event (see Figure 5, 3.1.7.1).

MultiplexingContext

The multiplexed usage of the different accelerators requires some special handling of your C++ code. In case the data of your acquisition is dependant on special lines (SFTPRO, EASTA, etc.), and you have entered one or more valid *target-timing-domains* in your design (LHC, LEI, PSB etc.), you have to provide the *MultiplexingContext*. For every get-and-set function you have to add this context as a parameter.

Example:

```
            MultiplexingContext* pContext ;
            pContext  =  pEv->getMultiplexingContext();
 READ:
            float hv_value = pDevice->hvValue.get(pContext);

OR WRITE:
            pDevice->hvValue.set(hv_value, pContext);
```

It is recommended to assign always the pContext as a parameter in the source code, even if not used. In case of later switching to multiplexed usage, no rework of the code is necessary.
The principle of this MultiplexingContext provides a big help for the developer as no more time has to be spent on handling the timing.

3.1.8.3. PLC-RT-Action

The **PLC-RT-Action** is a derivate of the *RTAction*, but is already equipped with special PLC procedures, such as

GetAcquisition
GetConfiguration
SetPLCCommandFields
SetPLCConfigurationFields

These actions reference the *std-services* PLC and are triggered and scheduled like the *RTActions*. The prepared source code files are stored in the folder GENERATED CODE in your Linux development path.

3.1.9. Events

The equipment is usually synchronized with the overall accelerator timing by receiving deterministic events. For each class, the equipment-specialist has to define a list of *logical events* by giving those names within the scope of the equipment-class. The binding of logical events with your *RTActions* is done in the scheduling branch. The final specification of these logical events with accelerator-, timer-, or hardware interrupt setting is done later within the Instantiation-Tool.

3.1.9.1. Logical Event

The logical events belong to the explicit event types. Explicit means the RTAction with its corresponding trigger will be set at design time. This setting is valid for all instances, independent of how many there are.

Timer

Choosing *timer* as a *logical even*t provides an internal clock which generates events in infinite constant time-steps (constant frequency). This leads to triggering your action independent of the general accelerator timing. The step size (in ms) has to be defined in the *timing-mapping* branch of the Instantiation-Tool. This feature is used for example, in getting frequent updates of a temperature or pressure measurement where accelerator timing is not an issue.

Mtg

Choosing *mtg* (master timing generator) as a *logical event* defines it as being dependent on a specific accelerator event (CTIM), which has to be specified in the *timing-mapping* of the Instantiation-Tool. Using the *mtg logical event* requires a hardware connection of your FEC with the global timing network. In addition, a timing-domain has to be specified within the *target-timing-domain* branch of the design tree.

Custom-event-source-ref

This setting references a custom-event-source, which is explained in 3.1.9.3.

User

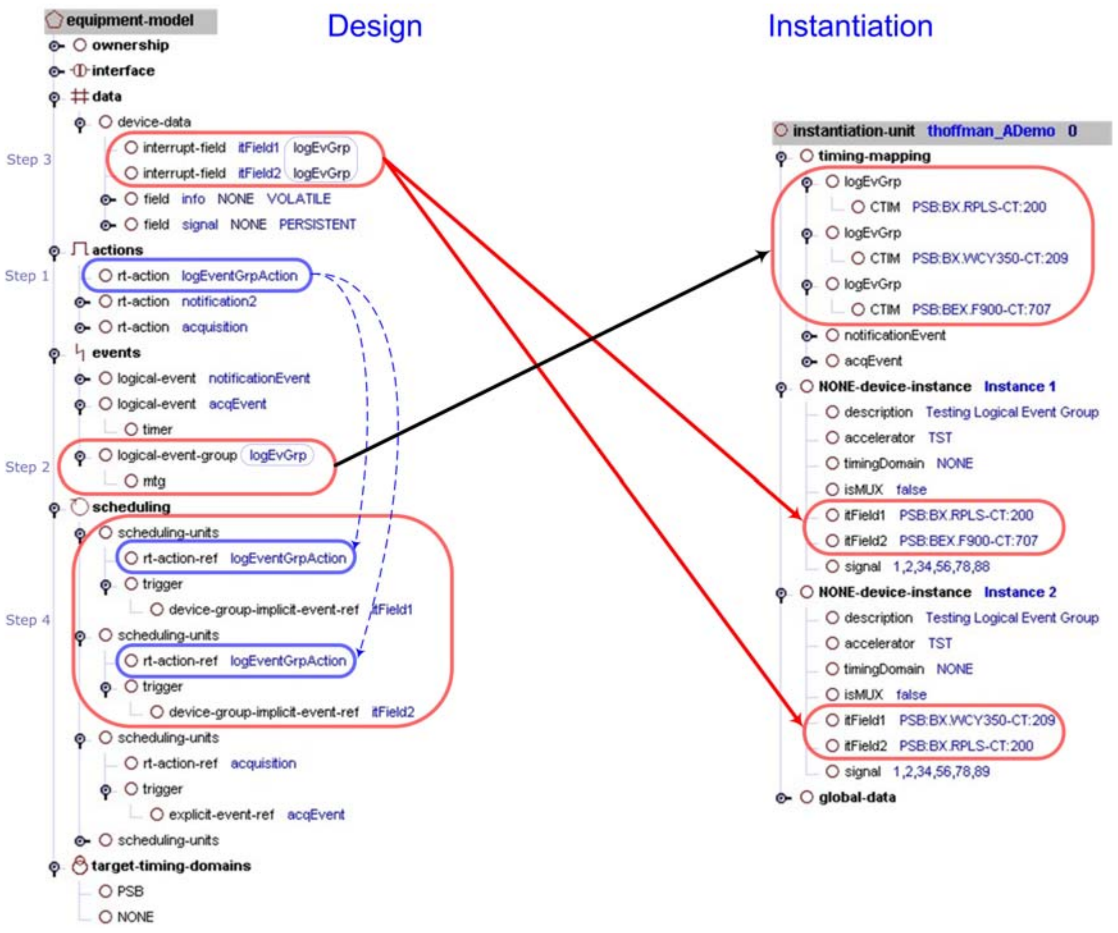
User event sources are not instantiated by the framework. An operator can trigger an *RTAction* using a *Server Action*. This type of triggering is timing-independent and also useful for sending information as a payload to the hardware.

//Not available anymore

Example FESA Class TestUserEvent: The class *Interface * pClassIntf* provides the method *fireUserEvent(evt,payload)*. This method has to be called in the *Server Action*, which shall trigger the *RTAction*. See an example in the Chapter 6.3.

3.1.9.2. Logical-Event-Group

The purpose of this event type is to associate one or more mtg events to one or several instances, and then to execute one dedicated *RTAction*.
The *logical-event-groups* belong to the implicit event types. Implicit means the final RTAction trigger will not be set at design time, but within the Instantiation-Tool. For trigger-setting, every instance then provides one or more interrupt fields. These can be filled with CTIM/LTIM triggers provided by the *logical-event-group* selection in the timing-mapping branch of the Instantiation-Tool. The required settings for the design and instantiation phase of this operation are shown in Figure 7.



```
equipment-model
  ownership
  interface
  data
    device-data
      interrupt-field  itField1  logEvGrp
      interrupt-field  itField2  logEvGrp
      field  info  NONE  VOLATILE
      field  signal  NONE  PERSISTENT
  actions
    rt-action  logEventGrpAction
    rt-action  notification2
    rt-action  acquisition
  events
    logical-event  notificationEvent
    logical-event  acqEvent
      timer
    logical-event-group  logEvGrp
      mtg
  scheduling
    scheduling-units
      rt-action-ref  logEventGrpAction
      trigger
        device-group-implicit-event-ref  itField1
    scheduling-units
      rt-action-ref  logEventGrpAction
      trigger
        device-group-implicit-event-ref  itField2
    scheduling-units
      rt-action-ref  acquisition
      trigger
        explicit-event-ref  acqEvent
    scheduling-units
  target-timing-domains
    PSB
    NONE
```

```
instantiation-unit  thoffman_ADemo  0
  timing-mapping
    logEvGrp
      CTIM  PSB:BX.RPLS-CT:200
    logEvGrp
      CTIM  PSB:BX.WCY350-CT:209
    logEvGrp
      CTIM  PSB:BEX.F900-CT:707
    notificationEvent
    acqEvent
  NONE-device-instance  Instance 1
    description  Testing Logical Event Group
    accelerator  TST
    timingDomain  NONE
    isMUX  false
    itField1  PSB:BX.RPLS-CT:200
    itField2  PSB:BEX.F900-CT:707
    signal  1,2,34,56,78,88
  NONE-device-instance  Instance 2
    description  Testing Logical Event Group
    accelerator  TST
    timingDomain  NONE
    isMUX  false
    itField1  PSB:BX.WCY350-CT:209
    itField2  PSB:BX.RPLS-CT:200
    signal  1,2,34,56,78,89
  global-data
```

Fig. 7: Exemplary structure of a logical-event-group definition

These exemplary settings are leading to the trigger sequence shown in Figure 8. The RTAction *logEventGrpAction* is executed 3 times per cycle. It provides useful data at trigger #200 for both instances and useful data on trigger #209 only for instance 2. Later in the cycle on trigger #707, useful data for instance 1 is provided.

The explanation of the Instantiation-Tool is done in the Chapter 3.4.

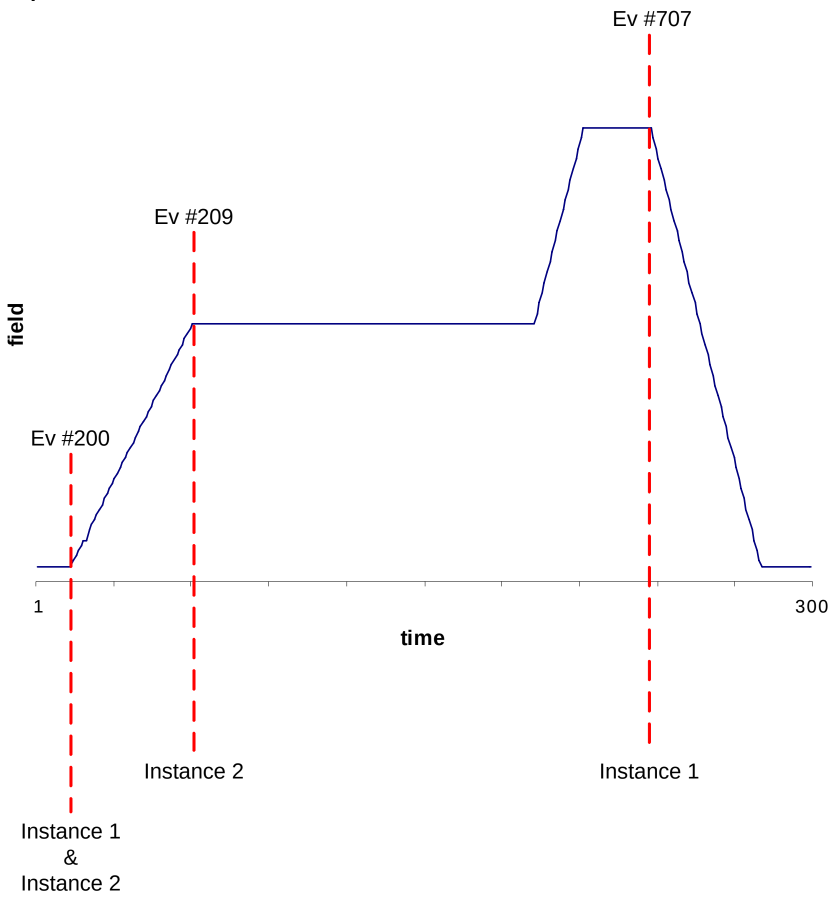


Fig. 8: Exemplary trigger sequence resulting from the settings in Figure 7. Remark: The ramping sketch and the trigger positions are fictitious

3.1.9.3. Custom Event Source

A logical event can be linked to a so called *custom-event-source*. Custom event-sources are not instantiated by the framework. It is your responsibility to write C++ code that creates the instances from within the <classname>RT::specificInit() method of your equipment class (in the file <classname>Realtime.cpp). When you define *custom-event-sources* in your design document, some C++ code templates will be automatically generated in the RT package, one for each event-source class. Implementing a custom event-source consists of filling its wait() method. This method manufactures an RTEvent object along which you may pass a string payload. Such a payload can then be accessed from within your RT actions.

3.1.10. Scheduling

Within the *scheduling* branch of the design tree the association between the *logical event* (see Events) and your *RTAction* in so called *scheduling-unit*s will be assembled.

3.1.10.1. Scheduling-Units

Most common scheduling-units consist of a reference to an *RTAction* or *PLC-RTAction* and a reference to a trigger, already defined in the events branch. This *RTAction* is executed within the main thread for the complete device-collection. If there are more than one of these scheduling-units triggered at the same time or are overlapping, those are performed sequentially.

Selection-criterion

To be more selective in triggering *RTActions* the selection-criterion provides the possibility to filter on different device-collections and their instances which fulfill the same filter criteria. These can be equal field values (hw-field, base-field, and data-field), which can be set explicit to a final value or be implicit (see selection-rule) if a device-group-implicit-event-ref with an interrupt-field is already defined. Using the selection-criterion this way executes the *RTAction* sequentially for all the instances, fulfilling the criterion.
For the handling of many parallel running *RTAction*-threads the selection-criterion is used in combination with the concurrency-layer. Therefore the per-device-group option of the concurrent flag has to be set to YES. For all instances of all device-collections matching the selection-criterion, the referenced *RTAction* is executed in parallel.

Selection-rule

implicit:
In case there is a selection condition attached to this scheduling-unit, this will be automatically augmented so as to make sure that the devices sorted by the condition also share the same interrupt-field. If there is no device-selector condition explicitly defined for this unit, the framework's action factory will create as many instances of the action as there are homogeneous groups of devices with respect to the interrupt-field's value.

concurrent:
See 3.1.10.2 Concurrency layer

Anticipated

This is a kind of flag which provokes a pretrigger-like operation. After setting *anticipated* to a *scheduling-unit*, the *RTAction* is executed one cycle earlier.
Typically these anticipated *RTActions* are used when a hardware device has to be initialized before the real measurement trigger occurs. This can be useful for a slow working or slow communicating device. Setting the flag only makes sense in a multiplexing context using mtg-events or mtg-group-events. The mtg-event must not carry any payload.

Apart from the case of the PS, the mtg-event carries a payload which dynamically identifies whether the multiplexing-context it relates to belongs to the current or to the next cycle of the telegram. In this situation such a flag is of no use.

3.1.10.2. Concurrency-Layer

In case the user wants to execute the *RTAction* in parallel threads, a concurrency-layer can be defined. Each layer then requires a scheduling-unit with reference to the *RTAction*, a reference to a trigger plus the concurrent flag with the layer name. This leads to parallel execution of instances of the *RTAction*. In case another instance of the *RTAction* shall be executed, a new scheduling-unit and a new concurrency layer have to be added. This is only reasonable for a few instances.

3.1.11. Target-Timing-Domains

Your equipment class may be deployed on one or several machines of the AB complex. Each machine is usually associated with a specific timing domain, while transfer-lines that connect different machines are usually associated to two domains. A device instance must belong to only one timing-domain, while you may instantiate different devices on different timing domains.
The instantiation schema is derived from the information you provide here, i.e. the possible timing-domain into which you will be able to instantiate your class will be restricted to the list you provide here.
If the device class does not require a connection to a timing-domain, select *None.* Available timing-domains are:

- CPS
- PSB
- ADE
- LEI
- SPS
- SCT
- LHC
- None

3.2. Linux Utilities

When your Linux environment variables are properly defined, FESA provides you with a set of scripts to automatically create and populate a C++ development directory, to deliver your equipment for operation, and to deploy and instantiate it on front-end computers. The complete description of all available Linux FESA commands can be found at:

http://project-fesa.web.cern.ch/project-fesa/development/fesaLinuxUtilities.htm

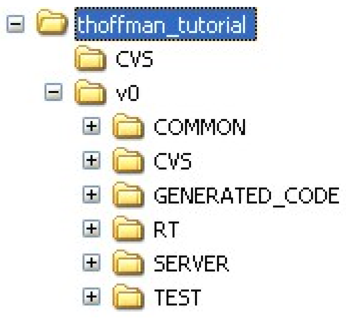
3.2.1. The FESA Commands

The most important commands are listed here:

In case a new device class is finished and stored in the database, use the command:

"Fesa Setup <devicename> <version> scratch".

This creates the complete directory structure of your new class within your Linux work folder and also creates the source code.



Tree 2: Standard directory structure

If you have later on applied changes on your device class (in the equipment model), you have to perform a:

"Fesa Synchronize <devicename> <version>"

followed by a "make", as FESA has to generate new source code out of it, which has to be recompiled.

In case you have applied changes within your instances, e.g. setting new initial values or adding new instances, you have to perform a:

“Fesa Instantiate <devicename> <version> <FEC-name> <directory>”.

If you want to store your binary on the frond-end and your source code in the CVS for publication or as backup, you have to perform :

“Fesa Deliver <devicename> <version> <CPU-type>”.

In case you just want to store your source code in the CVS repository, the commit command may be used:

“Fesa Commit <devicename> <version>”.

3.2.2. Retrofit

Retrofit means updating your FESA class from an older to a newer version of FESA. Therefore a special Perl-script is available which does all conversions automatically. For this procedure please follow the retrofit instructions at:

http://project-fesa.web.cern.ch/project-fesa/development/retrofitNotes.htm

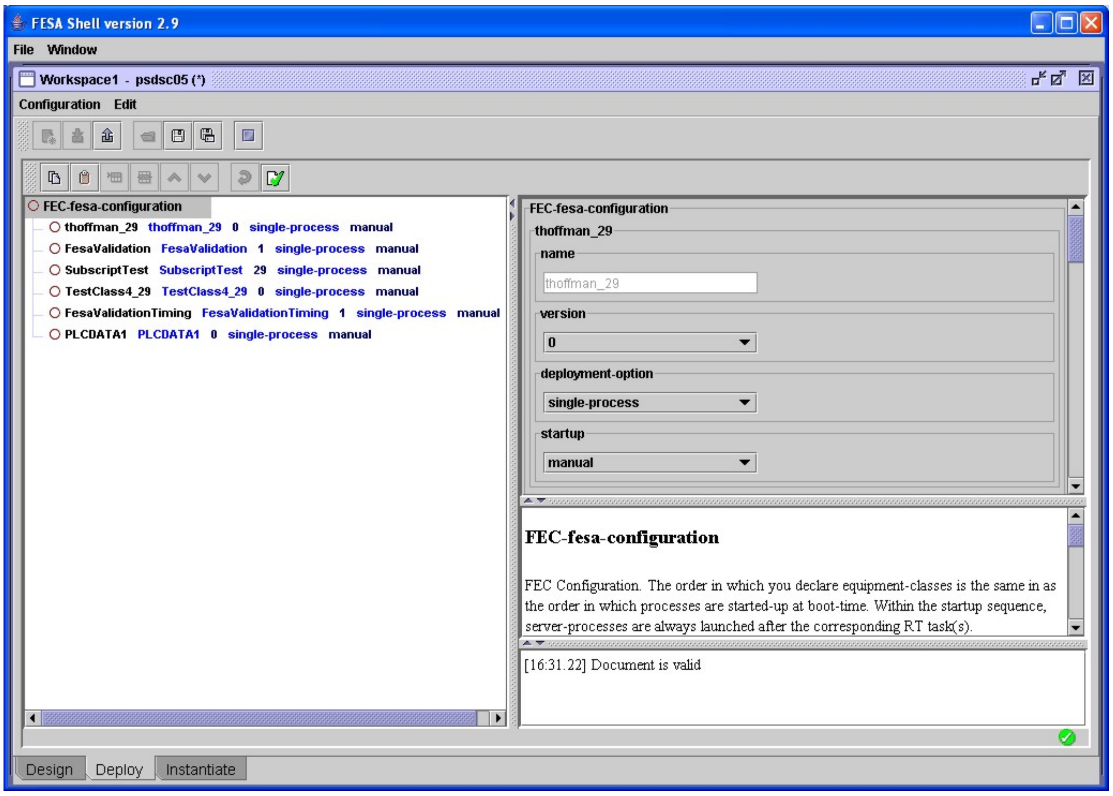
3.3. Deployment

In case you have stored your design and created the necessary source code, the class itself has to be transported to one or several FECs.

Note: Additional information on this subject can be found at [4].

For this the FESA Development-Corner website provides the link to the stand-alone *Deployment-Tool* or to the Shell.

After opening the *Deployment-Tool* and pressing *retrieve*, a pop-up window requests the selection of a connected FEC to which the FESA class has to be deployed. The graphical tree shows the already installed FESA classes on this FEC.



```
FEC-fesa-configuration
  thoffman_29  thoffman_29  0  single-process  manual
  FesaValidation  FesaValidation  1  single-process  manual
  SubscriptTest  SubscriptTest  29  single-process  manual
  TestClass4_29  TestClass4_29  0  single-process  manual
  FesaValidationTiming  FesaValidationTiming  1  single-process  manual
  PLCDATA1  PLCDATA1  0  single-process  manual
```

FEC-fesa-configuration
thoffman_29
name: thoffman_29
version: 0
deployment-option: single-process
startup: manual

FEC-fesa-configuration

FEC Configuration. The order in which you declare equipment-classes is the same in as the order in which processes are started-up at boot-time. Within the startup sequence, server-processes are always launched after the corresponding RT task(s).

```
[16:31.22] Document is valid
```

Fig. 9: Screenshot of the Deployment-Tool (Shell version)

A right-mouse-click on the FEC-fesa-configuration branch offers the *add* function, which has to be selected. From all available FESA classes you have to select the class to be deployed.
Afterwards a new entry can be found in the topmost position of the deployment tree.

As can be seen in Figure 9, the following parameters have to be set:

3.3.1. Version

Just select the version number of the class which has to be deployed.

3.3.2. Deployment Option

The deployment options have influence on the performance of the actual class and also on the whole FEC with all started classes.

<u>single-process:</u>
Server and realtime activities will run as two different threads in one single process. This is the most common option and is recommended.

<u>separate-server-split:</u>

Two separate server and RT processes are realized, which can communicate via a shared memory segment.

<u>shared-server-split:</u>
This feature also splits server activity and real-time activity into two processes, relying on a shared memory segment for communication, but in this case the server-part is merged with other FESA classes within a so called shared-server-process.

<u>shared-server-unsplit:</u>
In addition to the shared-server-split, this option also includes the RT part into the shared-server-process. This method should be avoided as only one RT task can be handled within this process.

<u>shared-server-interface:</u>
This realizes a class within a shared-server-process which has no RT part.

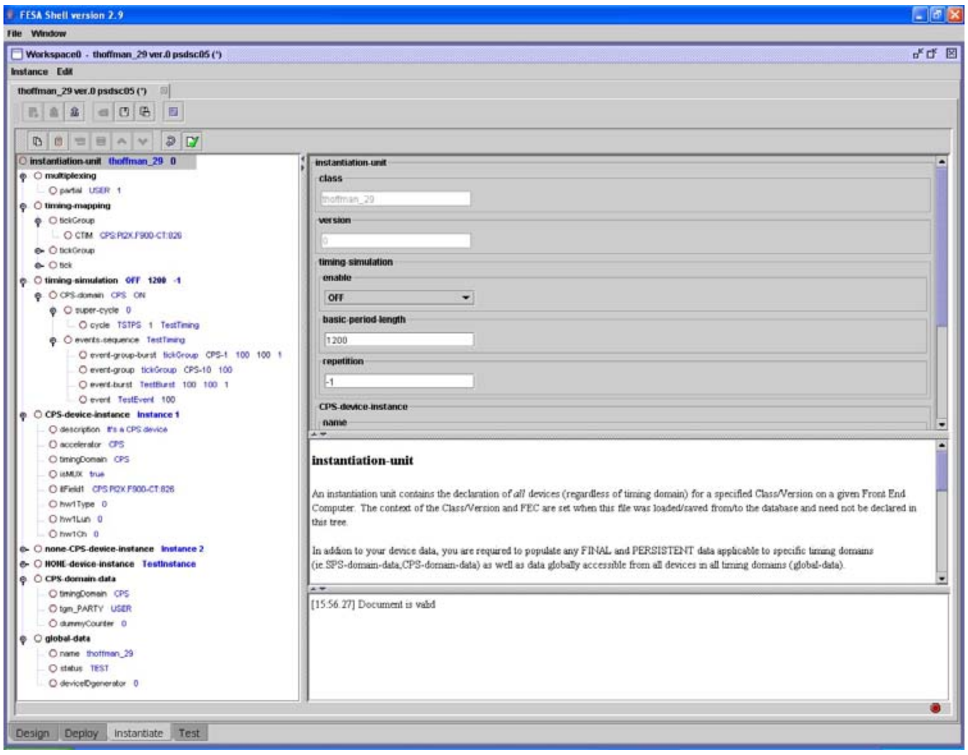
<u>separate-server-interface:</u>
Only one single process for *Server Action* without RT part is realized.

3.3.3. Startup

Selecting *manual* means starting the executable manually. This can be done with a simple ssh <FECNAME> command in a Linux terminal window. Change to the TEST directory of your class and enter ./<executablename>.
Selecting *automatic* leads to an automatic startup of the executable upon reboot of the FEC. The startup order of several FESA classes on the FEC is given by the order in the deployment tree.

3.4. Instantiation

The FESA development shell and the Development-Corner website also provide an *Instantiation-Tool*. After the design and the deployment phase of the FESA class, the instantiation is essential to define the amount and type of different devices, regardless of a timing domain. PERSISTENT and FINAL data can be defined here for single instances, or in case of *global-data*, for all instances. The main function of this tool is to couple the events from the design phase with the actual CTIM events or with an internal timer. Defining a *timing-simulation* helps testing FESA classes without connection to the real timing system. The *timing-domains* of the class, version, and FEC are set when this file is loaded from or saved to the database and need not be declared in this tree.



```
instantiation-unit thoffman_29 0
  multiplexing
    partial USER 1
  timing-mapping
    tickGroup
      CTIM CPS:PI2X.F900-CT:826
    tickGroup
    tick
  timing-simulation OFF 1200 -1
    CPS-domain CPS ON
      super-cycle 0
        cycle TSTPS 1 TestTiming
      events-sequence TestTiming
        event-group-burst tickGroup CPS-1 100 100 1
        event-group tickGroup CPS-10 100
        event-burst TestBurst 100 100 1
        event TestEvent 100
  CPS-device-instance Instance 1
    description it's a CPS device
    accelerator CPS
    timingDomain CPS
    isMUX true
    itField1 CPS:PI2X.F900-CT:826
    hw1Type 0
    hw1Lun 0
    hw1Ch 0
  none-CPS-device-instance Instance 2
  NONE-device-instance TestInstance
  CPS-domain-data
    timingDomain CPS
    tgm_PARTY USER
    dummyCounter 0
  global-data
    name thoffman_29
    status TEST
    deviceIDgenerator 0
```

instantiation-unit
class: thoffman_29
version: 0
timing-simulation
enable: OFF
basic-period-length: 1200
repetition: -1
CPS-device-instance
name

instantiation-unit

An instantiation unit contains the declaration of *all* devices (regardless of timing domain) for a specified Class/Version on a given Front End Computer. The context of the Class/Version and FEC are set when this file was loaded/saved from/to the database and need not be declared in this tree.

In addion to your device data, you are required to populate any FINAL and PERSISTENT data applicable to specific timing domains (ie.SPS-domain-data,CPS-domain-data) as well as data globally accessible from all devices in all timing domains (global-data).

```
[15:56:27] Document is valid
```

Fig. 10: Example of a fictitious instantiation-unit provided by the FESA shell

3.4.1. Multiplexing

If a *device-data* field in the Design-Tool was set to USER_PARTIALLY as *multiplexing-criterion*, the *multiplexing* branch is available to define the depth of the device-collection. This feature is used for optimization of the data-field depths as not all data values for all 24 USERS are required.

3.4.2. Timing-Mapping

All logical events defined in the design part have to be coupled here with either a dedicated timing event to be selected from combo-boxes (CTIM) or with a timer period setting (timer).

3.4.2.1. CTIM

A CTIM is a string in the style of ‘timingDomain’:'name':'code'. The user has to specify the domain dependent trigger signal to start his action.

3.4.2.2. Timer

An infinite loop of trigger signals is generated by this internal timer.

Specify the trigger period in ms. This period equals the delay time between each trigger signal. It must be a positive integer number (max. is 2.147e9). Setting the value to 0 suppresses triggering.

3.4.3. Timing-Simulation

The timing simulation is a standard service of the FESA infrastructure.
Note: additional information on the timing simulation can be found at [3]. On activation, it replaces the timing-hardware and all the software interfaces required to synchronize the deterministic process control (e.g. FESA RT-action) with an internal timing system.
The main objective is to support testing the FESA application without the real timing. This can be useful in case of missing timing-hardware, in shutdown periods, or while debugging. Also specialties such as evaluation of *telegram-data* or handling of *logical-event-groups* are provided.
The real timing and the *timing-simulation* may be defined in parallel, as one can easily switch between both timings by toggling the timing-simulation *on* or *off*. They cannot operate at the same time in parallel.
Note that the FESA Navigator (an auxiliary test environment also available within the FESA shell) works as well in the simulation mode.

The simulation is defined independently for each *timing-domain* predefined in the class design. Available *timing-domains* may be added to the simulation by a right-mouse-click on the *timing-simulation* entry.

Attributes of the *timing-simulation* are:

enable:

enable / disable simulation mode. This switches between simulation (ON) and real timing (OFF).

basic-period-length:

The length of the basic period in milli-seconds (default is 1200ms). It is a global parameter of the timing simulation.

repetition:

This defines how many cycles of the global simulation will be executed.
The global simulation can be activated once (repetition=1), several times (repetition=n), or repeated indefinitely (repetition=-1).

If one or more *timing-domains* were defined within the Design-Tool, they may be configured after adding them to the tree as follows:

3.4.3.1. <Domainname>-Domain

A <domainname>-domain, for example a **CPS**-domain, has to be defined and added as already described. This branch consists of two sub-branches called *super-cycle* and *events-sequence*. Per timing-domain there is exactly one super-cycle available, but many *events-sequences* if required.

Super-cycle

A *super-cycle* consists of one or many cycles. The executing order of the cycles equals the top-down order within the tree. The primary start of the *super-cycle* can be delayed by setting a time in milli-seconds to the attribute *shift-delay*.
To provide the best flexibility in the *timing-simulation* and to be as close as possible to the real timing, every *cycle* within a super-*cycle* is attached to an *events-sequence* which also is very flexible, as it offers several event types.

Cycle

The cycle is part of a super-cycle. The main attributes to configure the cycle are:

User

A typical telegram-group-user, such as SFTPRO, EASTA etc. has to be selected to provide information for multiplexing the PPM-data as in the real timing. Select one among the whole set of available users for the timing-domain concerned.

Basic-period-multiple:

This sets the length of the cycle, expressed in numbers of basic-periods. The duration of the *basic-period-length,* expressed in ms, is specified in the timing-simulation node, e.g. setting a 5 leads to 5x1200ms=6000ms or 6s cycle-length.

Events-sequence-name-ref

This refers to the *event-sequence* which is linked to the *cycle*. The *cycle* does not produce the trigger events; it only defines the time-frame being filled with an *events-sequence*.
Note: If you want to define a 'ZERO' cycle, you have to create and refer to an empty *event-sequence*.

If you are using telegram-data and you need to extract additional information such as particle type (PARTY) or destination (DEST), a *telegram-data* branch can be added to the *cycle*.

Events-sequence

As already mentioned in the *super-cycle* description, the cycle itself has to be linked to an *events-sequence*. In case no events are required, as for the ZERO cycle, you have to create an empty events-sequence.

One sequence can be referred by several cycles if those require the same event types. Conversely, a set of sequences can be declared but not used. This allows you to increase the number of test configurations.

In the real timing mode, you define a concrete CTIM or LTIM event for each logical event (explicit of implicit) you declared in the Design-Tool. The specific characteristics of each event (name and delay in particular) are programmed à priori and registered in a database.
In the simulation mode, you have to define each event characteristic yourself, as well as register this information within the Instantiation-Tool directly. This allows multiple adjustments and simplifies upstream application developments.

The *events-sequence* has to be named. This name is used as the *events-sequence-name-ref* as a cycle attribute.

For the design of the simulated timing behavior four event types are available:

event:

It simulates a simple CTIM or LTIM event linked with an explicit logical-event for which mtg is specified in the design. At least one

logical-event using mtg has to be designed. If not, the timing-simulation makes no sense.
The event has to be linked to a logical-event by selecting the name from the combo-box. In addition the delay time in milli-seconds has to be added. This delay is the time between cycle start and execution of the trigger.
Note: Setting the delay to a negative value means that it is works as a pretrigger.

<u>event-burst:</u>

Choosing event-burst leads not only to one but to (n) trigger signals within the given period and number of occurrences. The event burst simulates an LTIM pulse-train linked with an explicit logical-event/mtg specified in the design. The first event of this train is triggered after the fixed delay from the cycle start. Then (n-1) events are generated in regular intervals corresponding to the specified period.
Note: Setting the delay to a negative value means that it works as a pretrigger.

<u>event-group:</u>

The *event-group* event type has to be chosen if an implicit *logical-event-group* using mtg was designed and shall be simulated. It works like the *event* described above, except in addition, a *sub-name* to differentiate between each events is required (sub-name format: [timing-domain]-[index], example: LEI-01, LEI-02, PSB-01, CPS-01, etc.).
If *timing-simulation* is switched on, the contents of the *interrupt-fields* (required for using the *logical-event-group* functionality) of the instances is changed from a CTIM event-name to this *sub-name* as a placeholder.
The event is triggered after the fixed delay time to be entered in milli-seconds, beginning from the cycle's start.

<u>event-group-burst:</u>

The *event-group-burst* works like the *event-burst*. This event type has to be linked with an implicit *logical-event-group/mtg* specified in the design, and also *sub-names,* such as those used for the *event-groups* have to be defined.

3.4.4. Instances

There are three different types of instances, which can be added or removed easily. The sub-branches of the instances are dependent on the defined device-data fields in the design part. Whenever a device data field is defined as FINAL or PERSISTENT, an entry in each instance for this field is created and has to be populated. This is also valid for the global-data, but within a different branch.
In addition all types of instances provide branches with non-editable information, such as isMUX and timingDomain.

3.4.4.1. <Domainname>-Device-Instance

This type of instance is used in a multiplexed context within a specific timing-domain (<DOMAIN> replaced by i.e. CPS, LEI etc.). <Domain>-device-instances are available, when *target-timing-domains* are specified in the Design-Tool. By default a set of instances is created within the instantiation-unit. If some of these instances are not required, they can be deleted or replace by other types of instances.

3.4.4.2. None-<Domainname>-Device-Instance

This type of instance is not used in a multiplexed context, but within a timing-domain.

3.4.4.3. NONE-Device-Instance

In case no global timing at all is required, this type of instance may be chosen.

3.4.4.4. <Domainname>-Domain-Data

This branch is dedicated to populate the *telegram-group-fields* specified in the Design-Tool. A right-mouse-click on the *instantiation-unit* branch offers all available <Domain>-domain-data entries. Of course this only applies when one or more *target-timing-domains* are defined at design time.

3.4.5. Global-Data

If a *global-data* field was defined as FINAL or PERSISTENT, it may be set in this branch.

3.5. Navigation-Tool

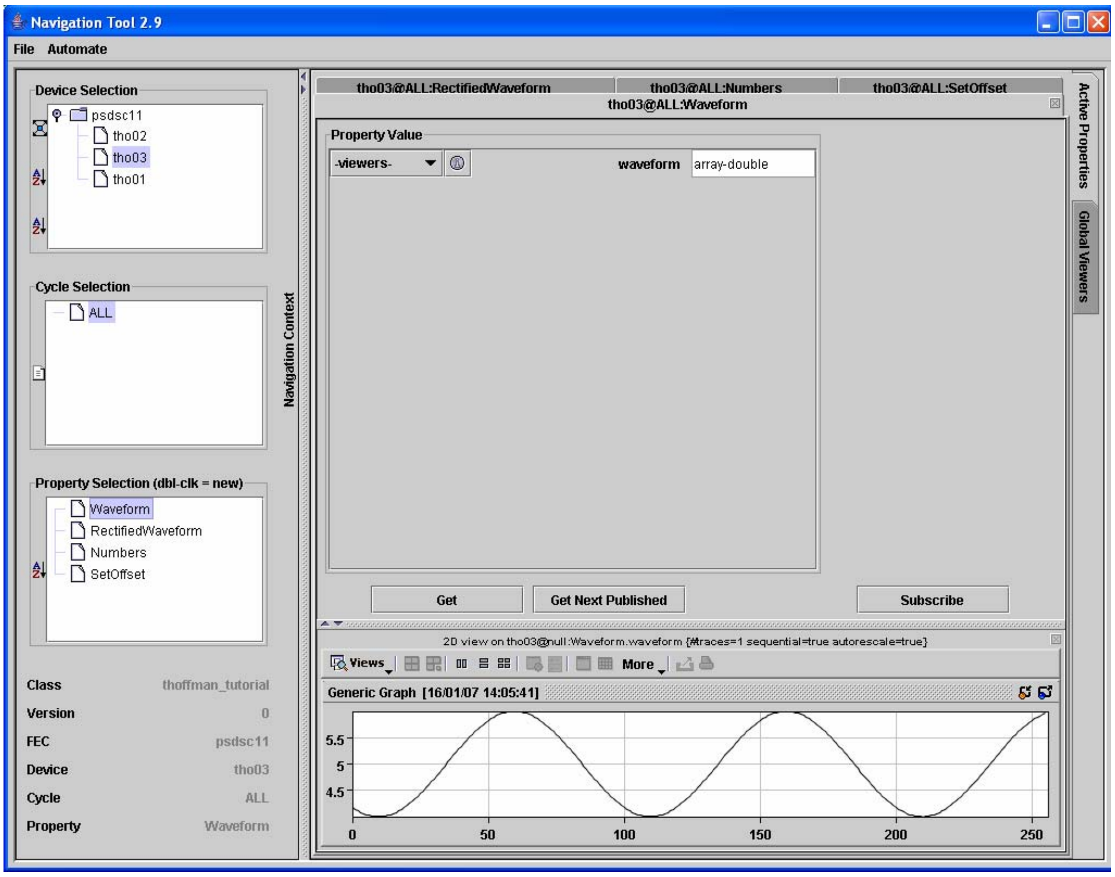
After having completed all steps beginning from the design until the final instantiation phase, the FESA class is ready to be tested. On the FEC the binary is started manually or automatically while the boot process. To get in contact with the executed binary the *Navigation-Tool* can be used. This is found within the Shell or as standalone tool on the Development-Corner website.
The *Navigation-Tool* provides access to all defined properties. Dependent on the specified get-or-set-action while the design phase the buttons *get* and/or *set* appear in the navigator window.

Working with the *Navigation-Tool* requires following steps:

1. Select a FEC in the *Device Selection* window.
2. Chose a beam target in the *Cycle Selection* window.
3. Select a property in the *Property Selection* window.

After this the navigation window will pop-up. It offers several different viewers such as 2D-plot (see Fig. 11), tables, or text logger and already useful data acquisition features such as printing, data storage, zooming, etc.

The *get*-and-*set* actions can be performed manually by pressing the equivalent buttons or by subscribing to the class which leads to automatic and frequent viewer updates.



waveform array-double

2D view on tho03@null:Waveform.waveform {#traces=1 sequential=true autorescale=true}

Class	thoffman_tutorial
Version	0
FEC	psdsc11
Device	tho03
Cycle	ALL
Property	Waveform

Fig. 11: Navigation-Tool for testing purposes

3.6. Data Management / Source Code Generation

This paragraph is not essential for working with FESA. It provides condensed background information on the data processing principals for the interested user.

All the FESA GUIs are written in Java. Therefore it is almost platform independent. The contents of the GUI, edited by the user in this obvious tree structure, are stored in XML files. XML files are human and processor readable ASCII files, also platform independent, with a fixed structure which allows parsing for a valid consistency of this file.
The files are stored on the one hand in so called CLOBS (character large objects) as reference on the other hand the data is shredded and stored in relational database tables. This provides not only storage but also data access for other services such as the Alarm Monitor.
Additionally they can be exported to regular XML files on the hard disk.

By executing Linux Perl scripts (Linux commands, e.g. *Fesa Setup*...) the stored XML files (e.g. from the design tree) are sent to an XSLT processor, which operates similar to a classical compiler such as gcc. This XSLT processor transforms the contents of the XML files to valid C++ source code with help of predefined templates.

This data flow is organized by the FESA Data Management System (DMS). Its dependencies to the FESA packages are shown in Figure 12.

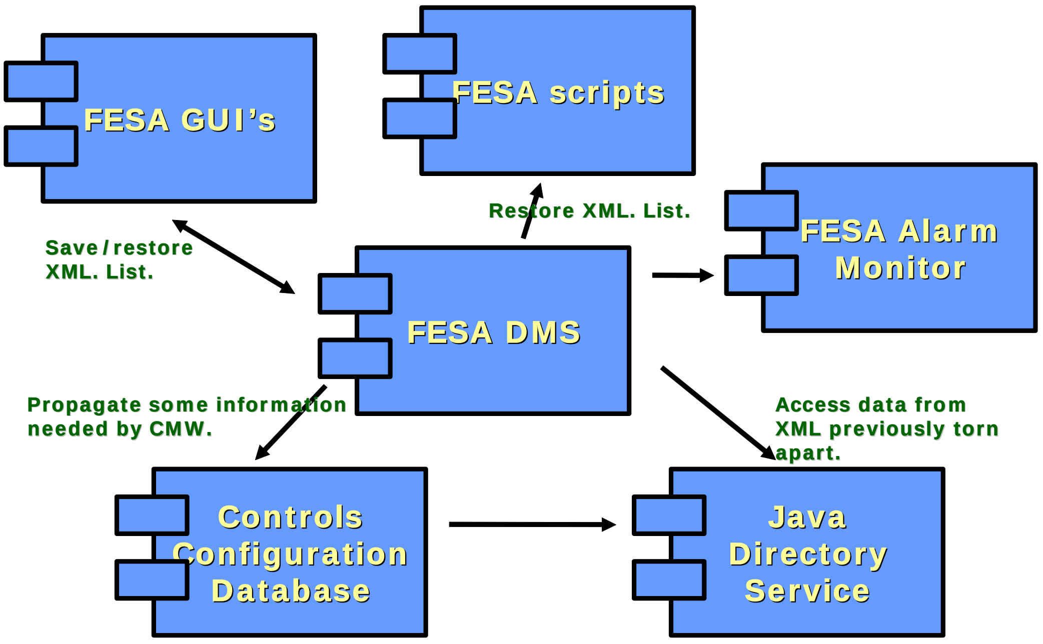

FESA GUI's
FESA scripts
FESA Alarm Monitor
FESA DMS
Save / restore XML. List.
Restore XML. List.
Propagate some information needed by CMW.
Access data from XML previously torn apart.
Controls Configuration Database
Java Directory Service

Fig. 12: FESA Data Management System (DMS) dependencies.

3.7. PLC-Interface

As FESA is dedicated to support easy equipment implementation into the control system the use of a specific PLC interface within the FESA Design-Tool is foreseen to fully integrate the large amount of PLC devices.
The usage of the PLC integration is well described in the documents "PLC integration for FESA 2.9" [5] and "PLC configuration for FESA 2.9" [6] available at the Development Notes link on the FESA Development -Corner website.

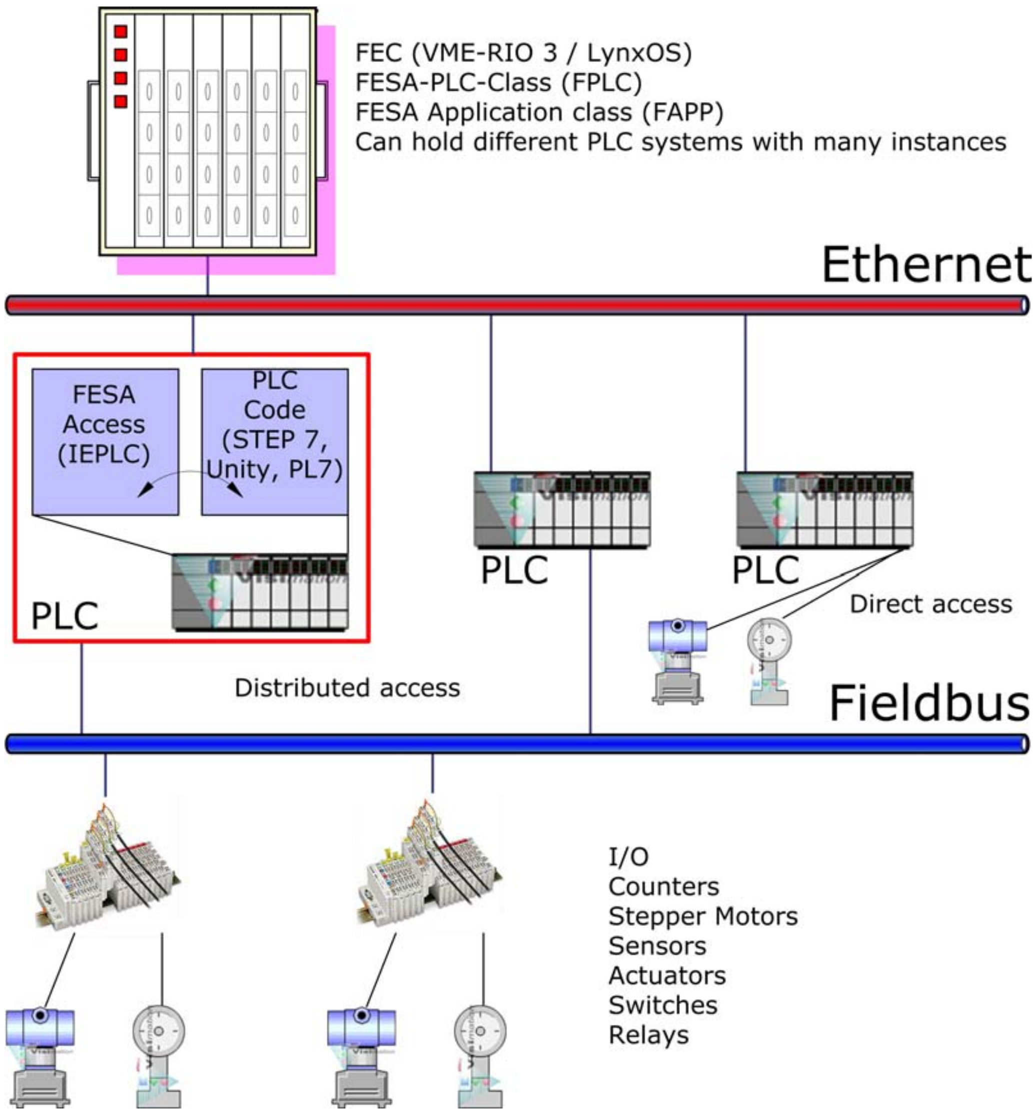


Fig. 13: Scheme of typical PLC installations.

The concept is to create first a dedicated PLC class (FPLC) which describes the PLC device and its processes. Already at startup of a new equipment-class-design a template for a PLC class is offered. All necessary properties and data fields are implemented there. This FPLC can be run separately for testing purposes and offers already the features “*status*”, “*settings*”, “*acquisition*” and “*configuration*”.
In a second step after creating the FPLCs a special FESA application class (FAPP) should be designed to bundle many FPLCs with same functionality or due to other reasonable causes. This FAPP handles the FPLCs as a master class. It offers the possibility to evaluate the acquired data and to operate various equipment types (from different PLCs eventually) from the operator’s point of view. The connection between FAPP and the FPLC is organized by the design option *equipment-links* (see 3.1.3). The *std-service PLC* has to be set for the FAPP. Both, the FPLC and the FAPP are usual FESA classes which have to be deployed to a FEC which handles the PLC devices.
To configure the PLC hardware for proper FESA access an online tool may be used to generate the device memory mapping for the PLC side programming (only the interface part, not the data acquisition). This tool is called “IEPLC Configuration” and is found at:

http://ieplc-configweb.web.cern.ch/IEPLC-ConfigWeb/v2.9

Usually for Siemens PLCs the software environment Step 7 (PL7 and Unity for Schneider PLCs) is used. In addition to the operational software which is developed within Step 7 by the user, the output code of the IEPLC source code has to be added to reserve the memory space on the PLC device for FESA access.
A typical PLC setup is presented in Figure 13. The FESA classes, FPLC and FAPP, are deployed to a FEC and several instances are built. Access of the PLC is possible via Ethernet and the IEPLC specific configuration. The devices which have to be controlled are either connected directly to a PLC unit or accessed via the fieldbus, e.g. FIPIO or Profibus.

3.7.1. PLC Requirements

In addition to the already mentioned PLC descriptions [5] and [6] some important issues should also be listed here.

One has to take care of the vertical order of the properties within a FPLC class design which is important for the correct mapping with the PLC device. This is also necessary for the vertical order of instances within the Instantiation-Tool.

Within the PLC class the USER event which may be used for manual triggering of the RTAction is created automatically. No additional source code has to be implemented.

The PLC-Expert can prepare a CSV list, a simple ASCII text file, with the correct order and parameters for the PLC mapping. This text file is later on processed by a script called “generatePlcClass.pl” which generates a valid design xml-file to be imported to the Design-Tool. A typical CSV-file consists of following entries:

PLC-property-type: CFG, CMD and AQN
PLC-property-name: any valid name
IEPLC-data-types: see Chapter 9
Dimension: n x type or array size

CSV-Example:

```
CFG, cfg_1_r,    REAL,    1
CMD, cmd_1_ba, BYTE,    3
AQN, acq_1_ca,  CHAR,    1024
AQN, acq_2_ca,  CHAR,    1024
...
```

Note: When deploying a FPLC class it is important to know that if this class is going to be used as a stand-alone class without a connected FAPP the *deployment-option single-process* in the Deployment-Tool should be set. In the other case the option *separate-server-interface* should be selected.

3.8. Alarm-Interface

FESA provides a front-end layer to the underlying LASER alarm-system. The usage, requirements, and constraints of the Alarm handling within FESA are well described in the document “Alarm interface for FESA 2.9” [7] which can be found in the Development Notes link of the Development-Corner website.

4. Example FESA Class

To demonstrate, how FESA works it is helpful to process all required steps on the basis of a real equipment device.

Note: This example shows only one of infinite possible ways of fabricating a FESA class. It can always be done in a different way. Not every feature of FESA is used here. This example is meant to be a first approach.

The class can be found in the FESA class repository Version 2.9 entitled as *TransferlineTrafo.* The source code is stored under the same name in the CVS repository.

4.1. The Module TRIC

The real VME device named “TRIC” [8] (**TR**ansformer **I**ntegrated **C**ard) has to be implemented into the control-system using FESA.
The device is a CERN-made VME module to integrate transformer currents. The connected transformer is installed in the transfer-line between the PSB and the PS. The measurement principle is as follows:

1. Receive a control event to initialize the module.
2. Receive a start event to trigger the measurement procedure.

The TRIC-output delivers finally the intensity of the proton beam in a cycle-by-cycle mode.

Of course this is not that simple such as listed above. The module provides a large set of properties which have to be set correctly and the required hardware driver already exists as it was used for developing and testing the module. The driver was prepared with the DriverGen [9] framework. The address mapping of all registers is organized within the driver. To get in contact with the hardware via C++ a device-handle had to be implemented. This piece of source code is added in the addendum. The VME crate also hosts a RIO3 PPC4 processor and a TG8 timing module.

Note: Hardware tests should be done in a stand-alone situation outside of FESA.

The measurement procedure shown in Fig.13 requires the setting of different time gates. This is a typical user-input (set).

The gate **t2** (beam signal is measured) as an example starts **t1**µs (length of **t1**) after the acquisition-trigger and lasts **t2**µs.

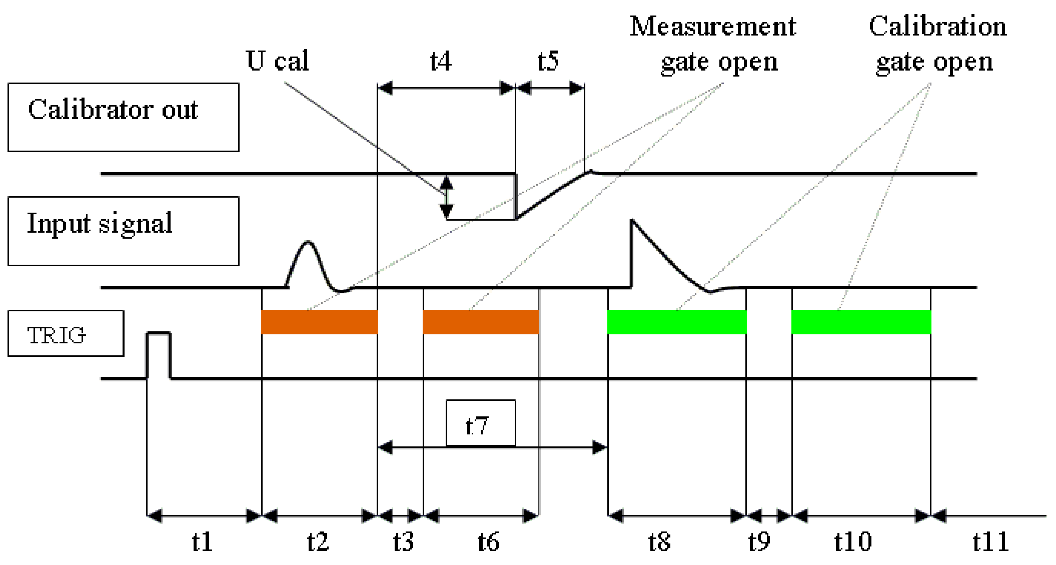


Fig. 14: TRIC-measurement procedure**.** All time steps t1-t11 and the calibration voltage (Ucal) have to be set. t1 - Measurement gate delay (from trigger), t2 - Measurement gate length, t3 - Measurement offset delay, t4 - Calibration gate delay (from the end of measurement gate), t5 - Calibration pulse length, t6 – Measurement offset gate length (the same as t2), t7 - Calibration pulse delay (from the end of measurement gate), t8 - Calibration gate length, t9 - Calibration offset delay, t10 - Calibration offset gate length (the same as t8), t11 - Test auto-run timer period, Ucal - High Voltage value

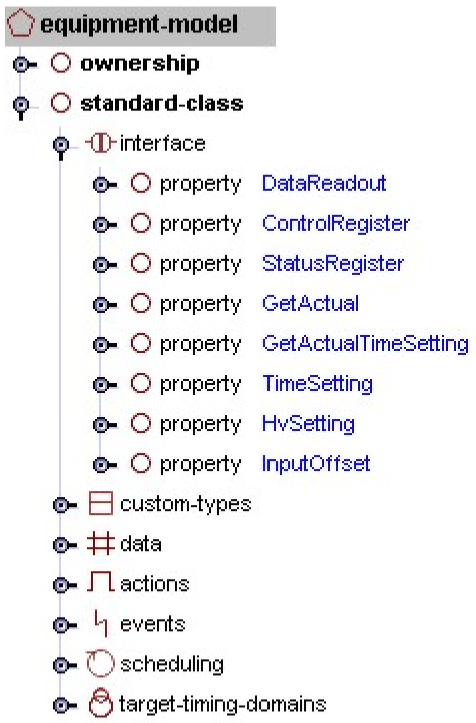
Figure 14 shows that after a trigger signal has occurred the beam intensity and afterwards the offset signal without beam is measured (**t1-t6**). The same is done with an internal calibration pulse and its offset (**t8-t10**).
After this procedure all data is stored in registers which have to be read out, calculated, and given out by your FESA class.

Module specialties such as the internal calibration, offset correction, and HV settings are omitted, as they are not required for learning FESA.

4.2. The Example Design

A design is not a fixed procedural method. You may switch from the *interface* part to the *data* part via *actions* back to *interface* and so on, as you whish and as it is required.

The relevant steps for a valid design may look like as follows:

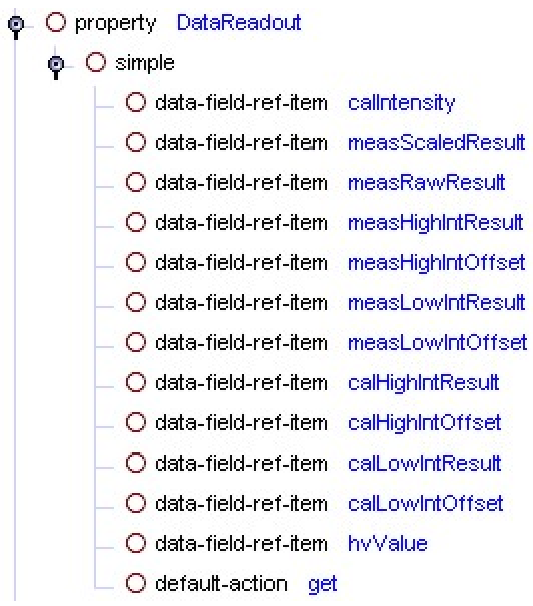


```
equipment-model
  ownership
  standard-class
    interface
      property  DataReadout
      property  ControlRegister
      property  StatusRegister
      property  GetActual
      property  GetActualTimeSetting
      property  TimeSetting
      property  HvSetting
      property  InputOffset
    custom-types
    data
    actions
    events
    scheduling
    target-timing-domains
```

Tree 3: All required interfaces

After opening a new design within the shell or Design-Tool, the tree should look like this and has to be populated now. Defining the ownership and the editor is simple but mandatory for security reasons.
We have to think about the properties which have to be accessed by an operator to get or set data. The final readout of acquired data is such a property.

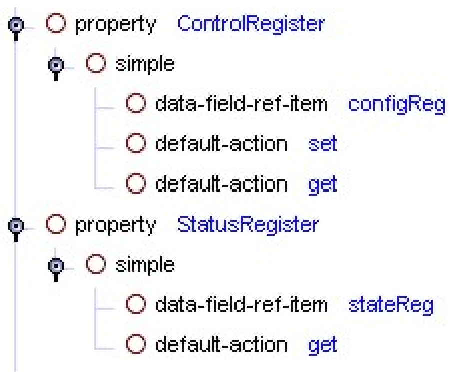
Therefore we add a simple property *DataReadout* and define several *data-field-ref-items* for all necessary data fields.
As we can only read the acquired data the *default-action get* is defined.



```
property  DataReadout
  simple
    data-field-ref-item  calIntensity
    data-field-ref-item  measScaledResult
    data-field-ref-item  measRawResult
    data-field-ref-item  measHighIntResult
    data-field-ref-item  measHighIntOffset
    data-field-ref-item  measLowIntResult
    data-field-ref-item  measLowIntOffset
    data-field-ref-item  calHighIntResult
    data-field-ref-item  calHighIntOffset
    data-field-ref-item  calLowIntResult
    data-field-ref-item  calLowIntOffset
    data-field-ref-item  hvValue
    default-action  get
```

Tree 4: The property DataReadout

Control (read/write) and status (read only) registers are used very often. The operator can change the behavior of the hardware by switching bits of a control register. In addition he can read back his settings. The status register may be used for observation of a running process, e.g. a bit is set to 1, when a measurement has finished. In our design it looks like shown in Tree 5.


```
property ControlRegister
  simple
    data-field-ref-item configReg
    default-action set
    default-action get
property StatusRegister
  simple
    data-field-ref-item stateReg
    default-action get
```

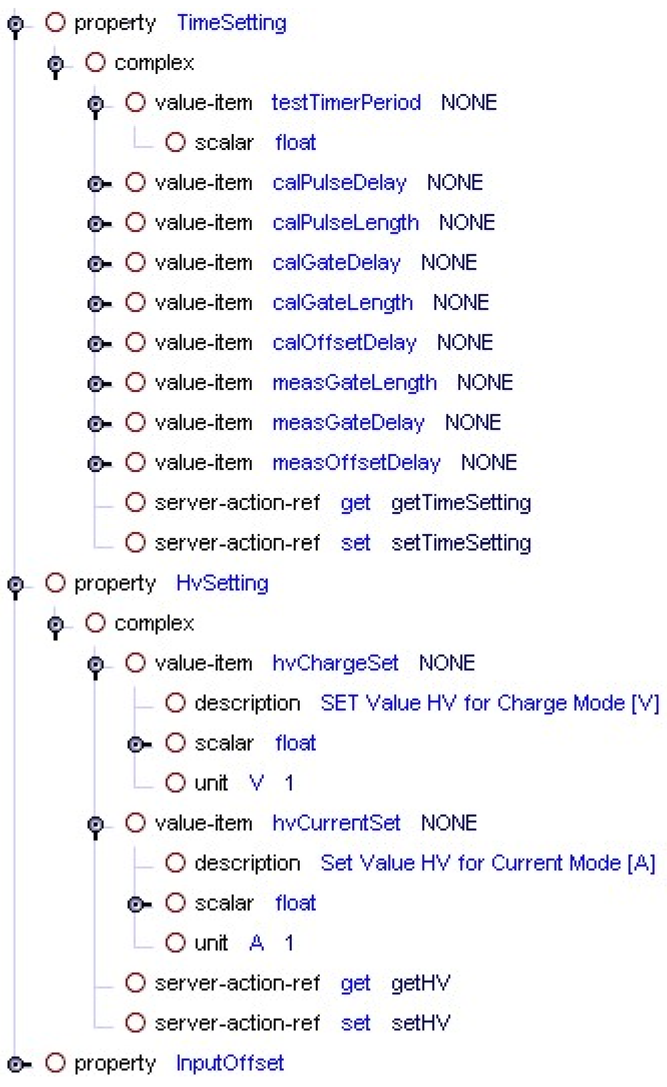
Tree 5: Control and status register

To provide a read-and-write access to the control register the default-actions get-and-set are chosen. The status register is limited to a read access.

As already mentioned, the behavior of the measurement procedure is described by time settings. We have to take care of that operator inputs in this case are done in seconds or micro-seconds, which can be *float* values. The hardware register instead accepts only *long* words. A conversion of these values is done in the C++ code while reading and writing to the hardware (*RTAction*).

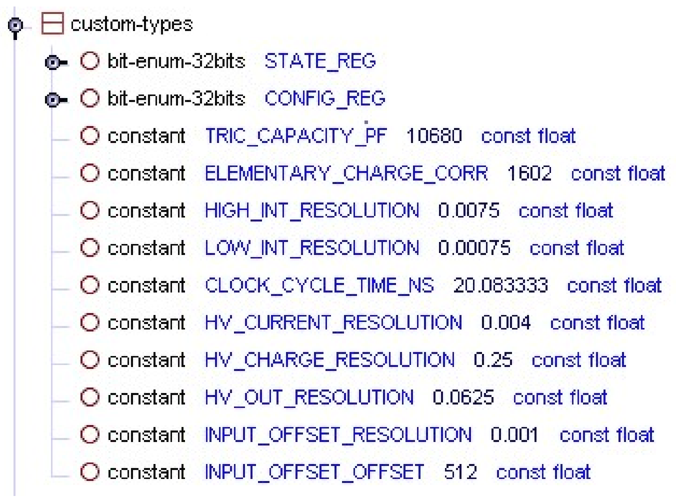
To accept more sophisticated variables the property type *complex* is used (see Tree 6). Doing this requires also specifying the *multiplexing-criterion* (here NONE) and the variable type (float).

Additional information can be added such as a short description or the used dimension (unit) and order of magnitude. Note that these settings are only for documentation.


```
property TimeSetting
  complex
    value-item testTimerPeriod NONE
      scalar float
    value-item calPulseDelay NONE
    value-item calPulseLength NONE
    value-item calGateDelay NONE
    value-item calGateLength NONE
    value-item calOffsetDelay NONE
    value-item measGateLength NONE
    value-item measGateDelay NONE
    value-item measOffsetDelay NONE
    server-action-ref get getTimeSetting
    server-action-ref set setTimeSetting
property HvSetting
  complex
    value-item hvChargeSet NONE
      description SET Value HV for Charge Mode [V]
      scalar float
      unit V 1
    value-item hvCurrentSet NONE
      description Set Value HV for Current Mode [A]
      scalar float
      unit A 1
    server-action-ref get getHV
    server-action-ref set setHV
property InputOffset
```

Tree 6: Complex properties.

For this project we have to use some constant values which are necessary for calculations such as from a time setting to a long word. As the board is synchronized with an onboard quartz oscillator with a frequency of 48 MHz, the resulting step size for time conversions is 20.083333ns. This value is used for all calculations and can be provided by defining it in the branch *custom-types* as a constant. See Tree 7, CLOCK_CYCLE_TIME_NS.

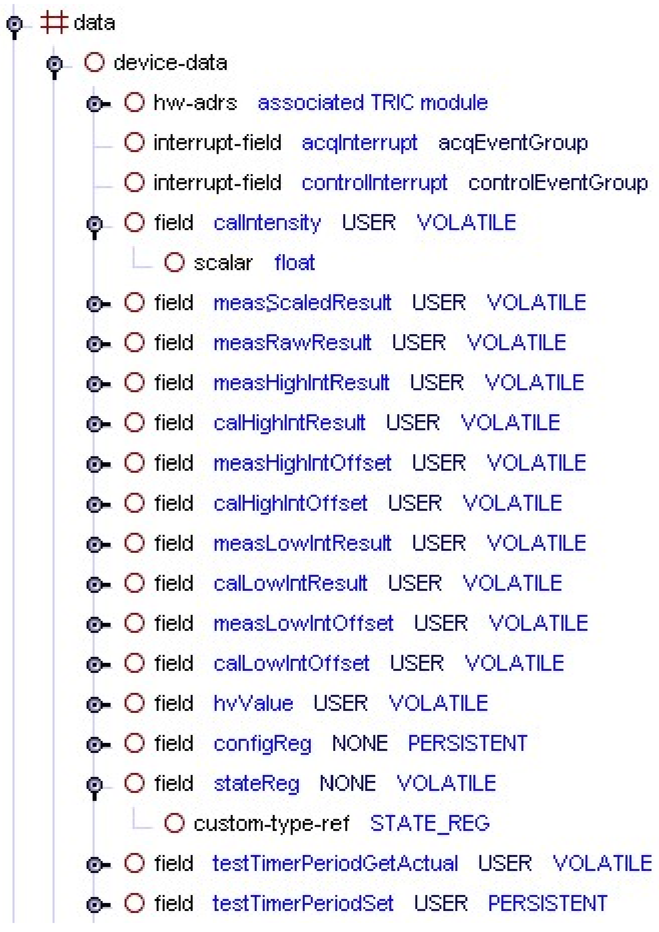


```
custom-types
  bit-enum-32bits  STATE_REG
  bit-enum-32bits  CONFIG_REG
  constant  TRIC_CAPACITY_PF  10680  const float
  constant  ELEMENTARY_CHARGE_CORR  1602  const float
  constant  HIGH_INT_RESOLUTION  0.0075  const float
  constant  LOW_INT_RESOLUTION  0.00075  const float
  constant  CLOCK_CYCLE_TIME_NS  20.083333  const float
  constant  HV_CURRENT_RESOLUTION  0.004  const float
  constant  HV_CHARGE_RESOLUTION  0.25  const float
  constant  HV_OUT_RESOLUTION  0.0625  const float
  constant  INPUT_OFFSET_RESOLUTION  0.001  const float
  constant  INPUT_OFFSET_OFFSET  512  const float
```

Tree 7: List of custom-types.

For constant variables it is recommended to use *custom-types* as they can be changed easily for all instances in one step, if required. Very helpful is the use of the *bit-enum-xxbits* types, as every bit of a word can be defined by name and later touched separately in the GUI or used in the source code.

The heart of the design is the data branch where all possible data fields have to be defined. Whenever a value has to be delivered to or received from either the operator or the hardware, it has to be defined as a data field. In addition, the *multiplexing-criterion*, the *data* type, and its *persistency* have to be defined. A part of the data field list is shown in Tree 8. Note that the *custom-type* STATE_REG (Tree 7) is referenced by the field stateReg (Tree 8) as the status register is not just a *long* or *float* but a *bit-enum*.



```
data
  device-data
    hw-adrs  associated TRIC module
    interrupt-field  acqInterrupt  acqEventGroup
    interrupt-field  controlInterrupt  controlEventGroup
    field  calIntensity  USER  VOLATILE
      scalar  float
    field  measScaledResult  USER  VOLATILE
    field  measRawResult  USER  VOLATILE
    field  measHighIntResult  USER  VOLATILE
    field  calHighIntResult  USER  VOLATILE
    field  measHighIntOffset  USER  VOLATILE
    field  calHighIntOffset  USER  VOLATILE
    field  measLowIntResult  USER  VOLATILE
    field  calLowIntResult  USER  VOLATILE
    field  measLowIntOffset  USER  VOLATILE
    field  calLowIntOffset  USER  VOLATILE
    field  hvValue  USER  VOLATILE
    field  configReg  NONE  PERSISTENT
    field  stateReg  NONE  VOLATILE
      custom-type-ref  STATE_REG
    field  testTimerPeriodGetActual  USER  VOLATILE
    field  testTimerPeriodSet  USER  PERSISTENT
```

Tree 8: List of data fields.

In Tree 9 the *actions*, *events*, *scheduling*, and *target-timing-domains* are listed. Already in Tree 6 the server-actions such as *getTimeSetting* were referenced. In the *actions* branch they are declared. Two *rt-actions* are defined, one to initialize and prepare the hardware module at an earlier event (*controlRTAction*), and another one to execute the measurement procedure and the intensity calculation (*acqRTAction*). As the acquired data has to be delivered to the operator, the property *DataReadout* is notified after the *RTAction* is processed.

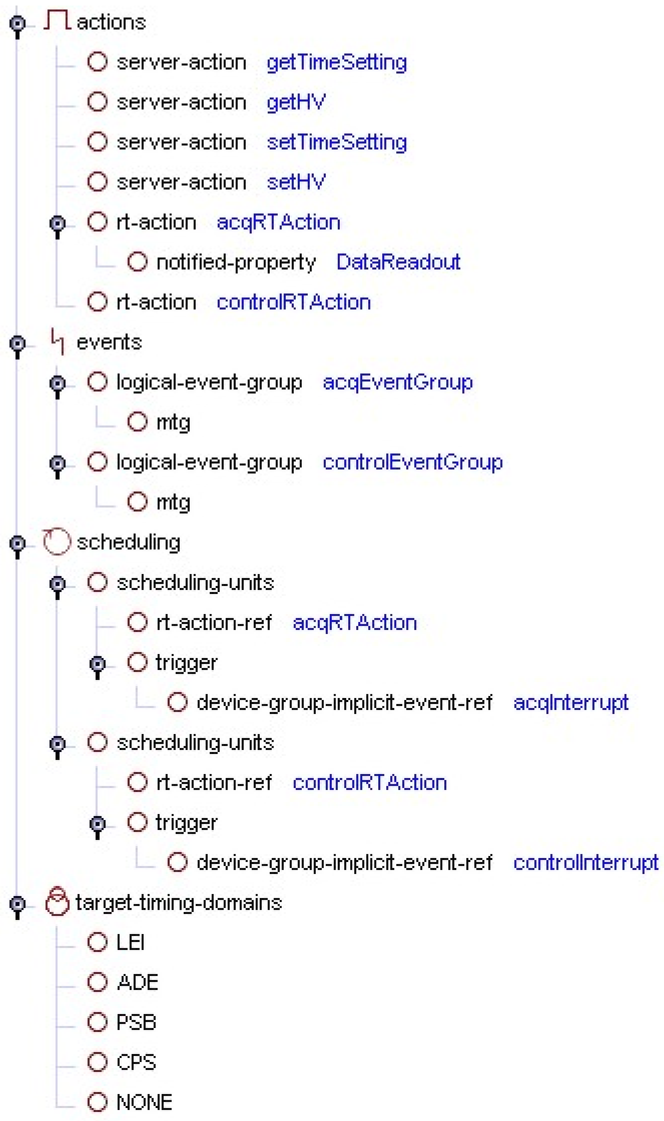
This measurement is multiplexed which means that beam to diverse targets may have different settings per cycle and user.

To receive accelerator-dependant triggers the *mtg* as an *event-source* was chosen. The *logical-event-group* was selected, to be prepared, if several transformers (instances) have to be triggered on the same event executing the same *RTAction*. Therefore *interrupt-fields* are required which are defined in the *scheduling* branch as a *device-group-implicit-event-ref*.

Finally the *target-timing-domain* has to be selected. This can be more than one. The final *timing-domain* is selected within the Instantiation-Tool per instance.

If the design is valid, it can be saved and the code production started using the Linux utilities.

A design can be changed any time, for example adding the error and alarm handling if required. Of course the source code is updated, without destroying already written code.



```
actions
    server-action   getTimeSetting
    server-action   getHV
    server-action   setTimeSetting
    server-action   setHV
    rt-action   acqRTAction
        notified-property   DataReadout
    rt-action   controlRTAction
events
    logical-event-group   acqEventGroup
        mtg
    logical-event-group   controlEventGroup
        mtg
scheduling
    scheduling-units
        rt-action-ref   acqRTAction
        trigger
            device-group-implicit-event-ref   acqInterrupt
    scheduling-units
        rt-action-ref   controlRTAction
        trigger
            device-group-implicit-event-ref   controlInterrupt
target-timing-domains
    LEI
    ADE
    PSB
    CPS
    NONE
```

Tree 9: Actions, events, scheduling and target-timing-domain settings.

4.3. Source Code Production and Coding

Once the design is valid and stored, you have to switch to the Linux world. Create a new directory for your FESA classes and change to this directory.
Type "Fesa Setup MyClassName 0 scratch" and press enter. The "Fesa" script will install a new directory structure and creates source code. The most relevant source code rudiment for own coding is found in the *RT* and *SERVER* subdirectories. The .cpp files of the actions have the same name defined in the design.

Note: The complete programming part cannot be explained here. Only the approach is documented. In the HowTo part of this document you will find examples for useful methods.

To read from and write to the hardware usually memspace has to be declared. The complete register mapping of a VME module has to be known. If more than one module of the same type is installed in a crate, each has to be identified correctly (*LUN, CH*). Usually a driver is required, where all these mappings are organized. This driver is compiled with FESA by including (#include) it. In addition, a device handler accessing the correct module is required. The sample code for such a device handler for the TRIC module is added in the addendum.

4.3.1. Server Action

To work with the module all settings have to be entered by an operator. These are in our example timing settings and high voltage settings. Therefore some Server Actions were designed and generated, which have to be filled with code. The sense behind is, that the data is copied from the user input to the data field or vice versa by pressing a *get* or *set* in the Navigator (GUI).

Example of a set action (copy from input to data field):

```
float timer_period = this->data.testTimerPeriod.get();
        pWorkingDevice->testTimerPeriodSet.set(timer_period, pContext);

cout << "Set TimerPeriod: " << pWorkingDevice>testTimerPeriodSet.get(pContext) << endl;
```

As all values are multiplexed, the MultiplexingContext (here pContext) has to be added. In case you switch back to non-multiplexed mode for your property (e.g. testTimerPeriod) this additional variable does not disturb.

A Server Action is usually (it can) not connected to the timing and can be executed anytime.

4.3.2. RT Action

Our example uses two *RTActions*: One for updating all settings before the real measurement procedure is started and one to read the acquired data and to calculate the beam intensity. The measurement is not started by an *RTAction*. Therefore a hardware trigger input on the module is used, which executes the automatic onboard procedure. The *RTAction* only checks at the beginning, if the measurement has finished before making the readout.

Example of a read action from the TRIC hardware:

```
measResLo = ReadTricRegister(handle,MEAS_LO_I_RES_ID);
measResHi = ReadTricRegister(handle,MEAS_HI_I_RES_ID);

         pDev->measLowIntResult.set(measResLo, pContext);
       pDev->measHighIntResult.set(measResHi, pContext);
```

The function ReadTricRegister (see source code in the addendum) writes a register contents from the module to the variable measResLo. After this, it is written to the corresponding data field in the device collection.

Example of a write action to the TRIC hardware:

```
// Set Measurement Gate Delay (from Trigger to MeasGate)

value = (int)fabs((pDev>measGateDelaySet.get(pContext))/CLOCK_CYCLE_TIME_NS);

cout <<  "MeasGateDelay : reg[" <<MEAS_GATE_DEL_ID << "]  =  "<< value << endl;

WriteTricRegister(handle,MEAS_GATE_DEL_ID, value);
```

Of interest can be the handling and use of *bit-enum* types. In the example one dedicated bit of the config register has to be cleared (set to 0) to update a setting. If the bit is 1, then it has to be cleared. If it is 0, it has to be switched to 1 and back to 0 to perform the required update.
For this following code may be used:

Example of a bit-wise toggle of the config register of the TRIC module:

```
// toggle bit D8 of config register

CONFIG_REG::CONFIG_REG flag = (CONFIG_REG::CONFIG_REG)pDev->configReg.get(pContext);

 if (flag & CONFIG_REG::MeasFlagClear){
       (long)  flag &= ~CONFIG_REG::MeasFlagClear;
                WriteTricRegister(handle, CONFIG_REG_ID,(int)flag);

cout << "******************************************* Cleared Bit D8: " << flag <<
endl;

 }
  else {
       (long)  flag |= CONFIG_REG::MeasFlagClear;
                WriteTricRegister(handle, CONFIG_REG_ID, (int)flag);
                long)  flag &= ~CONFIG_REG::MeasFlagClear;
                WriteTricRegister(handle, CONFIG_REG_ID, (int)flag);

cout << "***************************Toggle bit D8 from 0 to 1 to 0 : " << flag <<
endl;
 }
```

4.3.3. The SpecificInit Class

In our case the file is called TransferlineTrafoRealtime.cpp and it is also stored in the RT directory. This file is important as it hosts the specificInit class. This class may be used to perform only one time specific source code, for example writing a default value to the module, to assure that it does not start with wrong data.

Example of a specificInit usage:

```
TransferlineTrafoRT::specificInit(int argc, char ** argv) {

vector<TransferlineTrafoDevice*>* pDevCol=TransferlineTrafoDevice::getDeviceCollection();
       for (unsigned int i=0; i < pDevCol->size(); i++){
         TransferlineTrafoDevice * pDev = (*pDevCol)[i];

         HANDLE handle = devHandle::getDevHandle(pDev->name.get());

         int config_value = 0x193; //Default startup value of Config register

         WriteTricRegister(handle,CONFIG_REG_ID, config_value);

cout<< hex << " TransferlineTrafoRT::specificInit is called, setting ControlRegister! "<<
config_value << endl;
       }
}
```

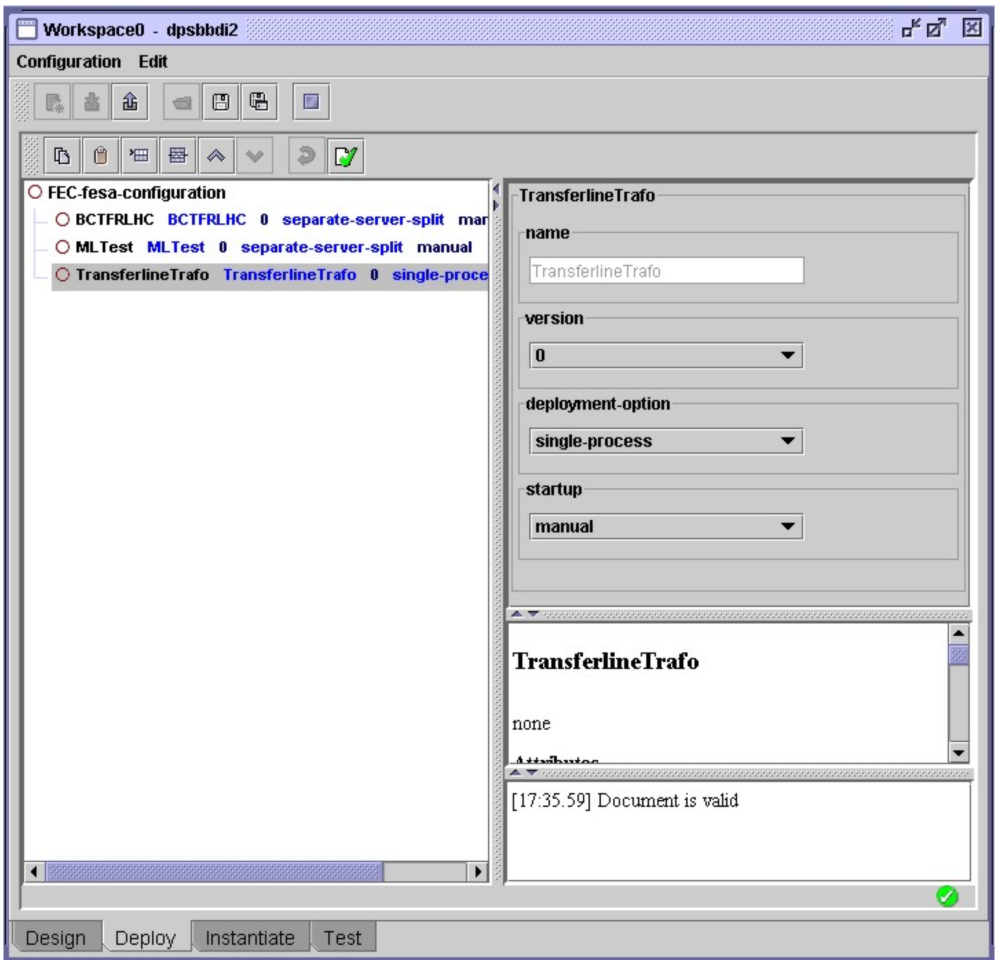
To perform the WriteTricRegister function, the device handler has to be defined here as well.

Anytime while being in the coding phase a compilation may be performed by typing “make”. If you are in a subfolder (e.g. RT) only

these files are compiled. If you switch to the main folder of your version (v0), you are able to compile the whole class.
To continue with the FESA class you have to perform different Linux commands, such as “Synchronize + make” in case you have also made changes to the design or “Commit” to update the CVS repository or “Deliver” to prepare the binaries being deployed to a FEC. Please read also [10].

4.4. Deployment of the Example Class

To prepare the designated FEC for the use of the FESA class the Deployment-Tool has to be started. Select “Retrieve” and select from the FEC. The appearing tree shows all already to this FEC deployed classes. A right-mouse-click on the FEC-fesa-configuration branch allows adding a new class. As shown in Figure 15 this is already done for the FEC named dpsbbdi2.



```
FEC-fesa-configuration
  BCTFRLHC         BCTFRLHC         0  separate-server-split  mar
  MLTest           MLTest           0  separate-server-split  manual
  TransferlineTrafo  TransferlineTrafo  0  single-proce
```

TransferlineTrafo
name: TransferlineTrafo
version: 0
deployment-option: single-process
startup: manual

TransferlineTrafo

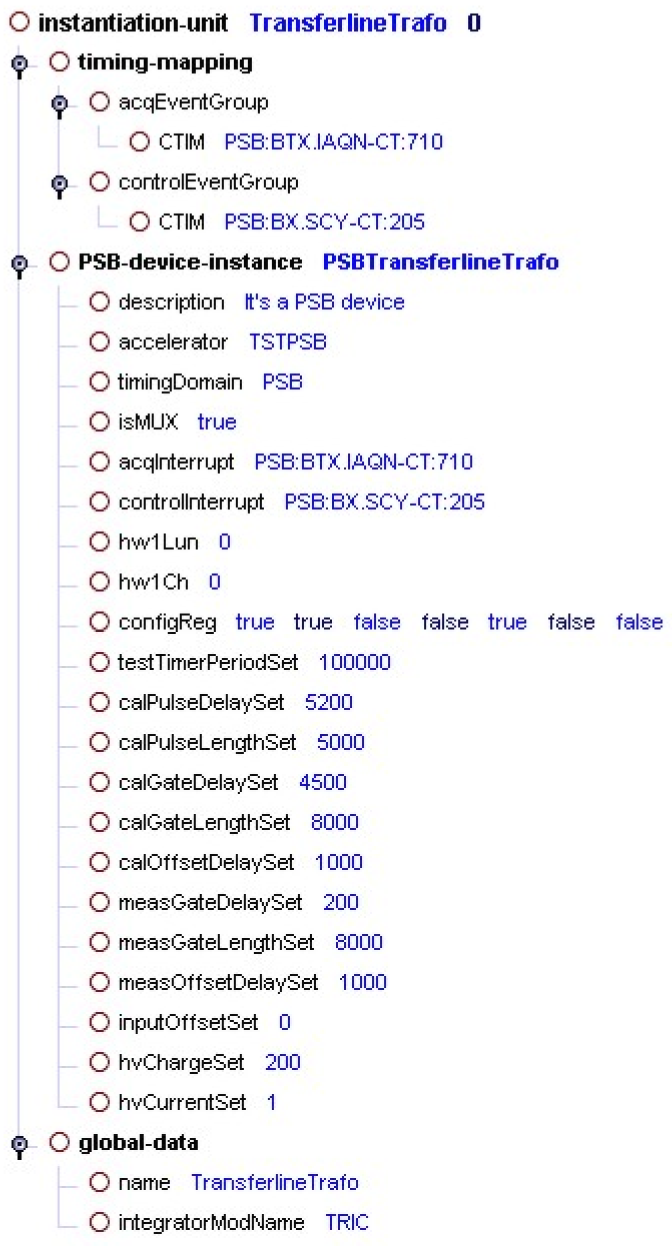
none

[17:35.59] Document is valid

Fig. 15: Deployment-Tool with installed TransferlineTrafo device class

As deployment option the “single-process” was chosen as this is sufficient for the TransferlineTrafo class. The startup type was set to "manual" due to the actual development status of this class which is not used in permanent operation at this time. Press store when you have finished all settings. Now you can switch to the Linux and enter the TEST directory of the class and execute "make".

4.5. Instantiation of the Example Class



```
instantiation-unit  TransferlineTrafo  0
  timing-mapping
    acqEventGroup
      CTIM  PSB:BTX.IAQN-CT:710
    controlEventGroup
      CTIM  PSB:BX.SCY-CT:205
  PSB-device-instance  PSBTransferlineTrafo
    description  It's a PSB device
    accelerator  TSTPSB
    timingDomain  PSB
    isMUX  true
    acqInterrupt  PSB:BTX.IAQN-CT:710
    controlInterrupt  PSB:BX.SCY-CT:205
    hw1Lun  0
    hw1Ch  0
    configReg  true  true  false  false  true  false  false
    testTimerPeriodSet  100000
    calPulseDelaySet  5200
    calPulseLengthSet  5000
    calGateDelaySet  4500
    calGateLengthSet  8000
    calOffsetDelaySet  1000
    measGateDelaySet  200
    measGateLengthSet  8000
    measOffsetDelaySet  1000
    inputOffsetSet  0
    hvChargeSet  200
    hvCurrentSet  1
  global-data
    name  TransferlineTrafo
    integratorModName  TRIC
```

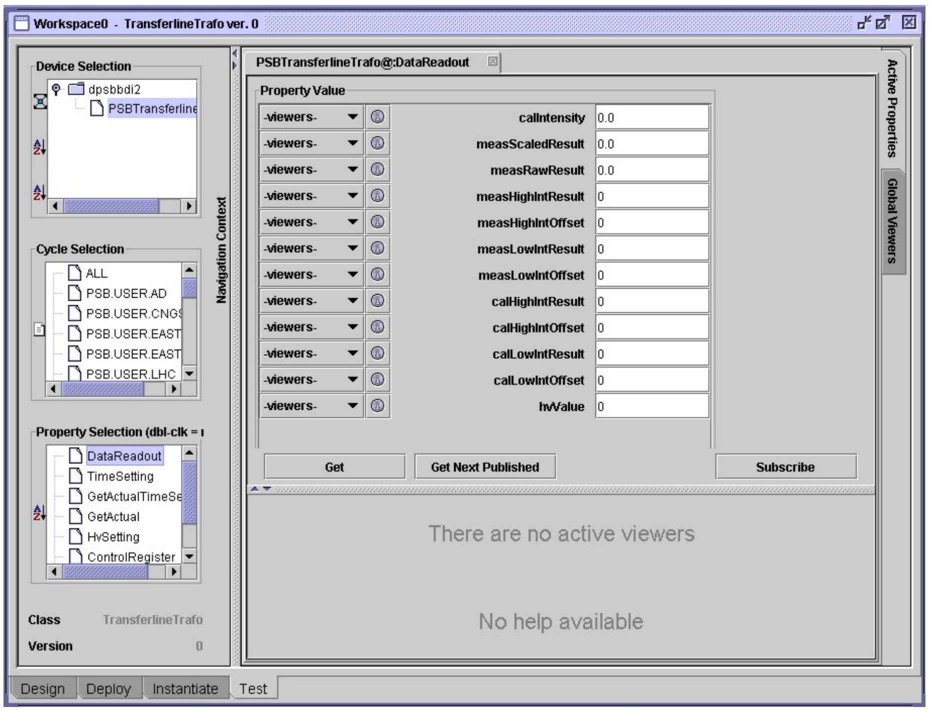
Tree 10: Instantiation-Tool.

Within the Instantiation-Tool the final settings have to be entered. Load first the FESA class by selecting "Retrieve". The amount and type of instances with all timing settings (event, domain, etc.) and startup values for PERSISTENT and FINAL data has to be defined. New instances are added by doing a right-mouse-click on the instantiation-unit branch. Press store when you have finished all settings.

The last step is to switch to Linux into the TEST directory and perform the Fesa Instantiate command. This command must be executed every time changes were made to the instantiation file. Login on the FEC, e.g. ssh dpsbbdi2, switch to the TEST directory and execute the binary manually.

4.6. Testing of the Example Class

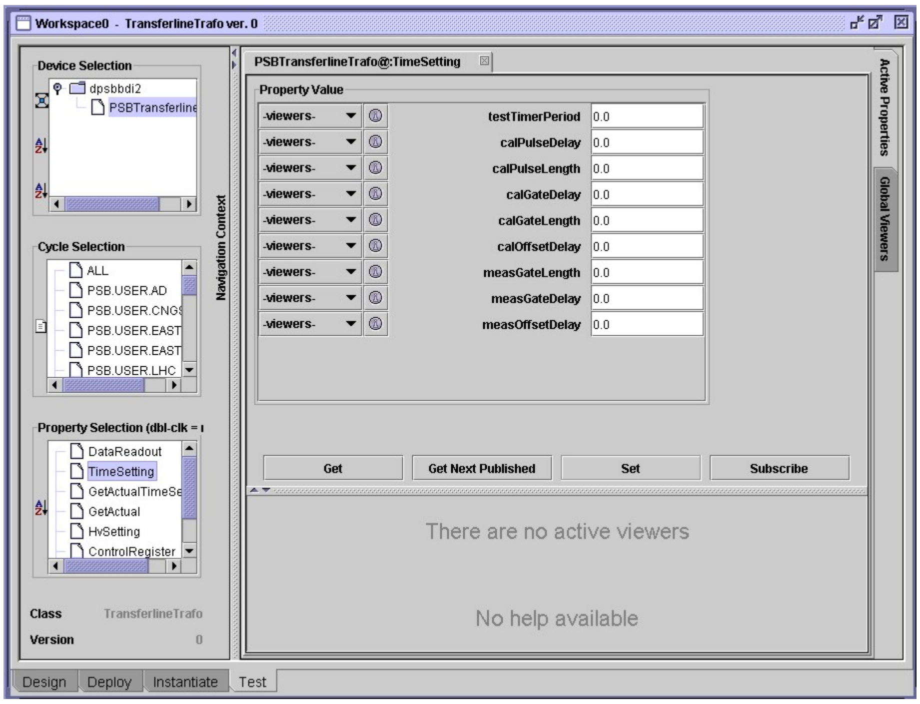
To connect to the running binary, the *Navigation-Tool* has to be opened. This can be done directly from the shell where the Tric class is already available.



calIntensity	0.0
measScaledResult	0.0
measRawResult	0.0
measHighIntResult	0
measHighIntOffset	0
measLowIntResult	0
measLowIntOffset	0
calHighIntResult	0
calHighIntOffset	0
calLowIntResult	0
calLowIntOffset	0
hvValue	0

Class TransferlineTrafo
Version 0

Fig. 16: Navigation-Tool showing all read-out properties (read-only)



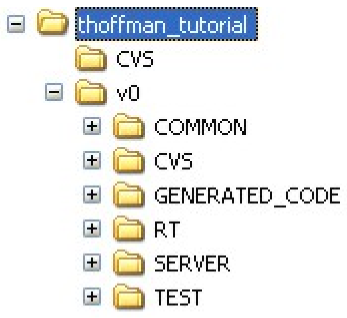
testTimerPeriod	0.0
calPulseDelay	0.0
calPulseLength	0.0
calGateDelay	0.0
calGateLength	0.0
calOffsetDelay	0.0
measGateLength	0.0
measGateDelay	0.0
measOffsetDelay	0.0

Class TransferlineTrafo
Version 0

Fig. 17: Navigation-Tool showing all timing settings with read-and-write access

5. Files in Project

Whenever a FESA class is installed locally in your Linux environment, a directory-tree containing a set of source code files is created.



The top-directory is named like the FESA class. Below this the class contents is placed in a folder labeled with the version number. In this folder (here v0) you will find also the main Makefile for the whole class. In every directory a CVS folder appears which is necessary for the CVS archiving procedure. Its data is of no use for the FESA class itself

Usually all the files created automatically are not foreseen to be edited except the RT and Server Action files. The RT, SERVER and TEST folder are containing a so called Make.specific file, which is used to add custom .cpp or .h-files or to add alternative include directories. These Make.specific settings are also interpreted as flags beneath the Makefile by the make command.

5.1. Common

This folder can be used to store user-created source code, i.e. own classes and functions. If these self-created files shall be linked to your FESA class, the adequate names and paths have to be added into the Make.specific file as already mentioned.

5.2. CVS

This folder is not of interest.

5.3. GENERATED_CODE

This folder contains the most generated files, mainly .cpp and .h files. Most of the definitions and settings made in the Design-Tool are realized here in class declarations and definitions.

<class-name>GlobalStore.h
Contains the class <class-name>GlobalStore with declarations for all data fields defined in *global-data* in the Design-Tool.

<class-name>Device.cpp
The *fault-fields* of the class are processed here; it provides a helper function for computing standard status values.

<class-name>Device.h
This file contains the device-class. All fields defined in the *device-data* are declared here.

<class-name>DomainStore.cpp
Usually not used. This routine includes the <class-name>-DomainStore.h file.

<class-name>DomainStore.h
This file contains the DomainStore-class. All fields defined in the *domain-data*, such as telegram-group-fields are declared here.

<class-name>EquipmentDefaultInterface.cpp
Interface handling, all properties and Server Actions are configured here and the ServerActionFactory is instantiated.

<class-name>EquipmentDefaultInterface.h
Declaration-file for the count/config variables related to properties and server-actions.

<class-name>EquipmentDefaultRealtime.cpp
RTAction and event configuration, instantiation of the RTActionFactory.

<class-name>EquipmentDefaultRealtime.h
Declaration-file for the count/config variables related to *RTActions* and events-sources, and for std-services.

<class-name>EventSourceFactory.cpp
Definition of the createCustomEventSource class.

<class-name>EventSourceFactory.h
Definition of the EventSourceFactpry class.

<class-name>GetSetDefaultServerAction.h
All default actions are realized here. This can be used to study coding of data transactions from the device to the data class and vice versa.

<class-name>TypeDefinition.cpp
Some rda declarations.

<class-name>TypeDefinition.h

Definitions of all data classes, especially for the custom-types. For enum definitions the namespaces are declared here.

<class-name>ServerActionFactory.cpp
ServerActionFactory definition

<class-name>ServerActionFactory.h
ServerActionFactory declaration

<class-name>RealtimeActionFactory.cpp
RTActionFactory definition

<class-name>RealtimeActionFactory.h
RTActionFactory declaration

FesaCompilationInfo.cpp
Version information

5.4. RT

<class-name>Realtime.cpp
Contains the Constructor and Destructor of the <class-name>RT class. Most important is the method specificInit, in which initial functions can be run once (i.e. open sockets or hardware handles), before the RT actions are executed.

<class-name>Realtime.h
The <class-name>RT class declaration.

5.5. SERVER

<class-name>Interface.cpp
Contains the Constructor and Destructor of the <class-name>Interace class. Most important is the method specificInit, in which initial functions can be run once (i.e. open sockets or hardware handles), before Server Actions are executed.
This can be also done in a separate custom initializing Server Action with the advantage, that a user can start this action manually.

<class-name>Interface.h
The <class-name>Interface class declaration.

5.6. TEST

<class-name>DeviceData.xml
Stores settings made within the instantiation unit.

<class-name>PersistentData.xml
In case data fields are defined as PERSISTENT, all the values, also for all 24 users, are stored here.
Note: To clean these values, this file may be deleted any time.

deploy<class-name>.cpp
Handles the deployment options set within the deployment unit such as single-process or separate-server-split etc.

6. Howto.....

6.1. Use the FAQ

On the FESA Development-Corner website a link to the FAQ pages is installed. On this page some complex issues are explained, exemplified with code examples. Following subjects are treated there:

- How can I compile and link my custom hardware library for test purposes?
- How can I configure an operational FEC to link equipment-classes against a specific hardware library?
- How can I configure an operational FEC to start equipment-classes automatically upon reboot?
- How can I pass command-line arguments to a FESA executable?
- I want to implement a property with a custom server action. Shall I define a simple or a complex property?
- How do I setup the local directory structure for a group of tightly-coupled classes relying on equipment-links?
- Why my custom-event source does not work ?
- How can I generate timing-context from my custom-event-source?
- Does FESA puts any restriction on the use of the std library?

- How can I clean up a FESA class-version?
- How can I force a FESA version from the FEC's makefile?
- The device instances I've just created don't show-up in the Navigator or applications can't see them.

http://project-fesa.web.cern.ch/project-fesa/development/FAQ.htm

6.2. Use BitEnum

Using *bit-enum-16Bits/32bits custom types* is sometimes helpful, if special bits have to be named and accessed. They are declared in a separate namespace to prevent mismatch with other bit-enums. It may happen easily that two bit-enums use the same name for a bit such as ON, OFF, ERROR, etc. so this way of different namespaces was chosen. A short example of how to use it is given in Chapter 4.3.2.

6.3. Trigger an RTAction on User request

from TestUserEvent Class Server Action

```
#include <TrigIncrementCounter.h>
#include <TestUserEventDevice.h>
#include <TestUserEventGlobalStore.h>

#include "TestUserEventInterface.h"


using namespace TestUserEvent;

TrigIncrementCounter::TrigIncrementCounter(const string& name,
AbstractServerAction::ServerActionConfig& serverActCfg) :
        ServerAction<TestUserEventGlobalStore, TestUserEventDevice,
TrigIncrementCounter_DataType >(name, serverActCfg){}

void TrigIncrementCounter::execute(RequestEvent * pEv){
                MultiplexingContext* pContext ;
        pContext = pEv->getMultiplexingContext();

        // to access an array or array2D field:
        // Type* pLocalVar= pWorkingDevice->field.get( pContext );
        // to access a scalr field:
        // Type localVar = pWorkingDevice->field.get( pContext );

        TestUserEventInterface * pClassIntf =
               dynamic_cast<TestUserEventInterface
*>(AbstractEquipmentInterface::getEqpIntfFromClassName("TestUserEvent
"));
       UserEvent evt = UserTrigIncCounter; // compile-time check

     string payload;
     if (data.counterId.get() == CounterID::COUNTER_1)
          payload = "0";
     else if (data.counterId.get() == CounterID::COUNTER_2)
          payload = "1";

//      int frequency = (int)data.frequency.get();
//      int count = (int)data.count.get();
//      ostringstream os;
//      os << UserTrigActionType::CALIBRATION << " " << frequency <<
" " << count;
//      pGlobalStore->startCalib.set(true);

       pClassIntf->fireUserEvent(evt, payload);      // no
multiplexing-criterion is inherit

}
```

6.4. Create a DeviceCollection

If you are in need to access a DeviceCollection in a part where it is not provided, e.g. in the specificInit function of the <classname> Realtime.cpp file then you may operate like in the given example (extract of example in 4.3.3):

SpecificInit:

```
vector<TransferlineTrafoDevice*>*pDevCol=TransferlineTrafoDevice::getDevice
Collection();
      for (unsigned int i=0; i < pDevCol->size(); i++){
       TransferlineTrafoDevice * pDev = (*pDevCol)[i];

             HANDLE handle = devHandle::getDevHandle(pDev->name.get());
             ...
```

7. References

[1] FESA Essentials
http://project-fesa.web.cern.ch/project-fesa/development/essentials.htm

[2] Guidelines and conventions for defining interfaces of equipment developed using FESA, 15.06.2005
https://edms.cern.ch/file/581892/1.1/LeirPropertyGuildelines-final-1.0.1.pdf

[3] Description of the FESA Timing Simulation, Development Notes:
http://project-fesa.web.cern.ch/project%2Dfesa/binaries/documents/TimingSimulationEssentials.pdf

[4] FEC startup sequence configuration manual
http://project-fesa.web.cern.ch/project%2Dfesa/binaries/documents/FesaFecStartup.pdf

[5] PLC integration for FESA 2.9
http://project-fesa.web.cern.ch/project-fesa/binaries/documents/PLCintegrationFS.pdf

[6] PLC configuration for FESA 2.9
http://project-fesa.web.cern.ch/project-fesa/binaries/documents/HowToIeplc.pdf

[7] Alarm interface for FESA 2.9
http://project-fesa.web.cern.ch/project-fesa/binaries/documents/AlarmInterfaceFS.pdf

[8] TRIC module: Developed by Grzegorz Kasprowicz and David Belohrad, AB-BI.

[9] DriverGen: a framework to prepare hardware drivers automatically. Contact: Yury Georgievskiy and Alain Gagnaire, AB-CO-FE

[10] FESA Linux Utilities Description
http://project-fesa.web.cern.ch/project-fesa/development/fesaLinuxUtilities.htm

8. Glossary

A.

Actions: Design branch, specify RTAction and PLC-RTAction for triggered, event driven actions or Server Action for timing independent get/set operations.

B.

Branch: sub-group or part of the design tree.

C.

Concurrency-Layer: Scheduling option to execute several instances of an RTAction in parallel.

Concurrent: Flag within a scheduling-unit to use the concurrency-layer

CTIM: Central Timing Event, distributed timing all over the accelerator complex

CSV: Comma separated value, an ASCII list of values ordered in columns, separated by commas or semicolons.

CVS: Concurrent Versions System, a version control and backup system for large software developments

D.

Deployment: To deploy a FESA class means preparing the FEC and copying the class to it.

Deployment-Tool: A Java application provided on the Development Corner website as stand-alone tool. It is also included in the “Shell” on the same website.

Design: The user must describe his new equipment interface, internal structures and real-time behavior. This process is called “design” using the Design-Tool.

Design-Tool: A Java application provided on the Development Corner website as stand-alone tool. It is also included in the “Shell” on the same website.

Developer: in most cases the FESA user, but also possible the FESA developer, who takes care of the FESA itself. See User or Equipment-Specialist.

Device: The software abstraction of an underlying hardware device. The primary role of a FESA equipment class is to ensure that the hardware device on one hand, and its software counterpart (device) on the other hand, continuously reflect each other's state. Device is also used to describe the instance of the equipment class. Also called equipment-software component.

Device Collection: A class that holds all instances of your equipment software, it provides the sum (deviceCollection.size()) of all instances and the actual instance number for the device loop (deviceCollection[i]).

DMS: FESA Data Management System, for data handling between FESA, the database, the scripts and XML storage.

DSC: Device Stub Controller, the unit of a mainframe (VME, cPCI, etc.) and an inserted CPU board.

E.

Equipment-software component: the FESA project, see Device

Equipment-Specialist: see User.

F.

FAPP: Master FESA class (application part) for PLC classes.

FEC: Front End Computer (also known as DSC). The destination for deployed FESA classes.

FESA: Front-end software architecture

Field: An object that contains specific device-data (input, output, parameter or state-variable). They can be multiplexed with respect to some pre-defined criterion (e.g. a specific user or particle-type) and are associated with some persistency attributes. Field naming is valid within a particular device's scope.

Final: see Persistency

FPLC: FESA PLC class (in the Design-Tool).

G.

GM: General Module, forerunner of FESA framework.

H. -

I.

IEPLC: Protocol and client library for PLC communication. Required for FESA-PLC connection.

Instantiation: On a FEC several instances of a class may be defined. Example: one FEC hosts several channels for current-transformers, which are installed in different timing-domains. For all these the settings can be different. Every BCT gets its own instance with its own settings. The settings are done in the Instantiation-Tool.

Instantiation-Tool: A Java application provided on the Development Corner website as stand-alone tool. It is also included in the “Shell” on the same website.

it-Field: Interrupt field for logical-event-groups, belongs to the device-data fields.

J. -

K. -

L. **Line:** Targets or users like SFTPRO, EASTA, ZERO etc.

Logical Event Group: Event type to organize RTAction triggers for separate instances.

LTIM: Local timing events, selfmade timing logic, like bursts etc.

M.

Multiplexing: The accelerator complex is a resource shared by different users. Sharing is achieved by a time-multiplexing scheme (a.k.a PPM) whereby users are allocated specific time-slots during which they are somehow granted ownership of the accelerator's sensors and actuators. By extension, this multiplexing scheme may accommodate several multiplexing dimensions with respect to which settings (resp. acquisitions) of actuators (resp. sensors) are associated with a particular usage context. For instance, the settings of a bending-magnet may differ according to the type of particle the beam is made-of, or according to the destination at which it is targeted.

MultiplexingContext: within your code this is used as a parameter passed to a function which is multiplexed. It provides the actual *line.*

N.

Navigator: an auxiliary test environment available within the FESA shell. All designed properties are accessible, especially within a timing-context, if defined. Data output can be visualized (graph, table, etc.). See Navigation-Tool.

Navigation-Tool: A Java application provided on the Development Corner website as stand-alone tool. It is also included in the “Shell” on the same website. See Navigator.

O. **Operator**: Person in charge of beam and accelerator control, working in the CCC, Cern control center. He is a user of the GUIs on the upmost tier of the control system.

P.

Payload: standard name for a data package which can transport values, strings, objects, etc. from class to class or Server Action to RTAction and so on.

Persistency: a data field must be specified as VOLATILE, PERSISTENT, or FINAL.
Volatile means you have to initialize your field value at startup, e.g. in the Instantiation-Tool or in specificInit (RT folder, DeviceClassNameRealtime.cpp) function. *PERSISTENT* means you only initialize the field once, and then the last used value is stored in a file called “DeviceClassNamePersistentData.xml” for all instances. *Final* means once set treats a given value as a constant.

Persistent: see Persistency

PLC: Programmable Logic Controller, a small computer used for automation of industrial processes, often used in harsh environments. Also known as SPS (the German acronym).

PLC-RTAction: see Actions

PLS: Program Line Sequencer, Syntax: timing-domain.groups.line, example:
PSB.USER.SFTPRO

PPM: Pulse-to-pulse modulation. Every pulse is a cycle which can have different settings.

Property: The standard-form of a service published by a FESA equipment class. A property has a name, a type and can be accessed in get-or-set mode. Invocation of a property implies passing its subject (a device name) as well as its context (a timing selector) to the equipment class.

Q. -

R.

Real timing: The global accelerator timing based on CTIM and LTIM events. The alternative can be the timing-simulation within the Instantiation-Tool.

Retrofit: A procedure to update a FESA class from an older to a newer FESA version. Usually this can be done by using script commands.

RPC: Remote procedure call, a kind of remote method invocation from one computer to another.

RTAction: see Actions

S.

Selection-criterion: An option within the scheduling-unit to filter instances with equal criteria, can be used in combination with concurrency-layer.

Server Action: see Actions

T.

TGM: telegram data within the timing-event.

Timing Domain: Each machine is operating in a special time domain. The available time-domains are: CPS, PSB, ADE, LEI, SPS, SCT and LHC.

Timing-Simulation: this test function is provided by the Instantiation-Tool, in case no real timing is available.

Tree: The graphical Java tools to design, deploy, or instantiate FESA classes use trees with branches and sub-branches which have to be filled with life.

U.

User: in most times the person in charge of installing a device in FESA, but also the beam targets or lines such as SFTPRO, ZERO or EASTA are called users (24 Users max.). Also, a timing-domain such as LHC or LEI is sometimes called user. Be careful and watch the context where “user” is used.

V.

Volatile: see Persistency

W. -

X. -

Y. -

Z. -

9. Data Types

FESA data-types are currently restricted to the ones supported by JAVA. These are:

Scalar Types	Uni-dimensional array Types	Bi-dimensional array Types
bool	bool	bool
byte	short	byte
short	long	short
long	long long	long
long long	float	float
float	double	double
double	string	string

In some cases the type definitions for IEPLC, PLCs, C++, and JAVA are different and have to be converted by the system. The differences are listed here:

IEPLC Types	SCHNEIDER PL7	SCHNEIDER Unity	SIEMENS SIMATIC	bit size	C/C++ Types	JAVA Types
CHAR	**WORD**	**WORD**	Yes	8	*char*	*char[...]*
BYTE	Yes	Yes	Yes	8	*unsigned char*	*short*
WORD	Yes	Yes	Yes	16	*unsigned short*	*long*
DWORD	Yes	Yes	Yes	32	*unsigned long*	*long long*
INT	**WORD**	Yes	Yes	16	*short*	*short*
DINT	**DWORD**	Yes	Yes	32	*long*	*long*
REAL	Yes	Yes	Yes	32	*float*	*float*
DT*	Yes	Yes	Yes	64	*double*	*double*

*DT is equivalent to an 8 byte array (64 bits) used to format the date and time.

10. Table of Figures

11. Addendum

11.1. Example source code for the TRIC module

Additional code for the device access called devHandle.cpp and devHandle.h based on the TRIC-module driver.

```
#include <devHandle.h>
#include <TransferlineTrafoDevice.h>
#include <FesaException.h>

using namespace TransferlineTrafo;

devHandle *devHandle::pSingleInstance=0;

devHandle::devHandle(){

  access_mode = IOCTL;

  vector<TransferlineTrafoDevice*>* pDevCol =
TransferlineTrafoDevice::getDeviceCollection();
  char *modName =(char *)TransferlineTrafoDevice::pGlobalStore->integratorModName.get();
  handle = new HANDLE[pDevCol->size()];
  for (unsigned int i=0; i< pDevCol->size(); i++) {
    int lun=(*pDevCol)[i]->hw1Lun.get();
    int chanN=(*pDevCol)[i]->hw1Ch.get();
    handle[i] = DaEnableAccess(modName, access_mode, lun, chanN);
  }
}
HANDLE devHandle::getDevHandle(const string &devName) {
  if (pSingleInstance == 0)
    pSingleInstance = new devHandle();
  vector<TransferlineTrafoDevice*>* pDevCol =
TransferlineTrafoDevice::getDeviceCollection();
  for (unsigned int i=0; i< pDevCol->size(); i++) {
    if (!strcmp((*pDevCol)[i]->name.get(),(const char *)devName.c_str()))
      return pSingleInstance->handle[i];
  }
  throw FesaBadParameter("TransferlineTrafo: ",-1,"Bad device Name");
}
```

devHandle.cpp

```
#ifndef _TransferlineTrafo_devHandle_H_
#define _TransferlineTrafo_devHandle_H_

#include <fesa/Fesa.h>
#include "TransferlineTrafoDevice.h"
#include "TransferlineTrafoGlobalStore.h"
extern "C" {
#include "Tricinclude/TricRegId.h"
#include "VMEinclude/DrvrAccess.h"
}
namespace TransferlineTrafo {

class devHandle {
        public:
         devHandle ();
        static HANDLE getDevHandle(const string &devName);
         private:
        static devHandle *pSingleInstance;
        HANDLE *handle;                       /* library handle */
        METHOD access_mode;
} ;
}
#endif
```

devHandle.h

Definition and declaration of helpful functions for read and write actions to the hardware:

```
#include <devHandle.h>
#include <TransferlineTrafoDevice.h>
#include <FesaException.h>
#include "measFunc.h"

using namespace TransferlineTrafo;


int TransferlineTrafo::ReadTricRegister(HANDLE handle, int regid){

  int nmemb, elSize, retCode, value;

  nmemb  = DaGetRegDepth(handle, regid); //MEAS_LO_I_RES_ID = MEAS_HI_I_RES_ID
  elSize = DaGetRegSize(handle, regid);
  retCode = DaGetRegister(handle, regid, &value, (elSize*nmemb));
  if (retCode <= 0)
    throw FesaBadParameter("TransferlineTrafo: ",-1,"Bad device Name");

  return(value);

}

int TransferlineTrafo::WriteTricRegister(HANDLE handle, int regid, int &value ){

  int elSize, retCode;


  elSize = DaGetRegSize(handle, regid);
  DaSetRegister(handle,regid,&value,elSize);
  if (retCode <= 0)
    throw FesaBadParameter("TransferlineTrafo: ",-1,"Bad device Name");

  return 0;
}
```

measFunc.cpp

```
#ifndef _TransferlineTrafo_measFunc_H_
#define _TransferlineTrafo_measFunc_H_

#include <fesa/Fesa.h>
#include "TransferlineTrafoDevice.h"
#include "TransferlineTrafoGlobalStore.h"
extern "C" {
#include "Tricinclude/TricRegId.h"
#include "VMEinclude/DrvrAccess.h"
}


namespace TransferlineTrafo {

  int ReadTricRegister(HANDLE, int);
  int WriteTricRegister(HANDLE handle, int regid, int &value);
}
#endif
```

measFunc.h

The source code files above have to be linked to the FESA device class by adding them into a make.specific file.

```
#
#  FESA framework             June 2004.
#


# specific sources (.c file)
SPECIFIC_CSRCS =
# specific headers (.h file)
SPECIFIC_CSRCSH =
# specific sources (.cpp file)
SPECIFIC_CLSRCS = measFunc.cpp devHandle.cpp
# specific headers (.h file)
SPECIFIC_CLSRCSH = measFunc.h devHandle.h
# specific path for include files (-I/...)
SPECIFIC_CXXFLAGS =
# specific library paths
SPECIFIC_LIBPATH=
# specific libraries
SPECIFIC_LIBS=
```

make.specific

Source code of the acqRTAction for the TRIC module: