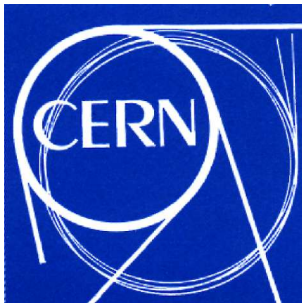




UNIVERSITÀ DI PISA
FACOLTÀ di
INGEGNERIA



Development of a software module for the Simple Analogue Function Generator used by the CERN Complex Accelerator

Giulia Taurelli

Relatori:

Ing. A. Domenici
Prof. C. A. Prete
Dott. F. Di Maio

Pisa, 26 Ottobre 2005

Index

Abstract	7
Chapter 1 Introduction	9
Chapter 2 Physic research and CERN.....	11
2.1 CERN	11
2.2 Matter and Anti-matter	13
2.2.1 Bricks of the Universe: the Building Blocks of Matter.....	13
2.2.2 The Standard Model	14
2.2.3 Unveiled mysteries and limits of standard model	16
Chapter 3 accelerator complex	19
3.1 Accelerator as ultimate microscope	20
3.1.1 Brief description of how an accelerator works	20
3.1.2 the CERN accelerator complex	21
3.2 Experiments at CERN	25
Chapter 4 LHC Control System	27
4.1 Overall Architecture	28
4.2 Network	31
4.3 Equipment access	32
4.3.1 The VME and PC Front End Computers	32
4.3.2 The PLCs	33
4.3.3 The Supported Fieldbuses	33
4.3.4 Servers and operator consoles	33
4.4 Machine timing and UTC	35
4.5 Data Management	38
4.5.1 Offline and Online Data Repositories	38
4.5.2 Control System Configuration	39
4.6 Communication and software frameworks	40
4.6.1 FESA: FEC Software Framework	40
4.6.2 Controls Middleware	40

4.7 Control Room Software	42
4.7.1 Software for LHC Beam Operation	42
4.7.2 The Software Development Process	42
4.7.3 Services for operations	42
4.7.3.1 Analogue Signals Transmission	42
4.7.3.2 Alarms	43
4.7.3.3 Logging	44
4.7.3.4 Post Mortem	44
4.7.3.5 Support for Real-Time Feedback Loops	46
4.7.3.6 Control System Monitoring and Diagnostic Tools	46
Chapter 5 Introduction of the GFAS	47
5.1 Assignment	47
5.2 Requirements	50
5.2.1 Representation of a function	50
5.2.1.1 Basic Function	50
5.2.1.2 Standard Function	51
5.2.1.3 The hardware representation	51
5.2.1.4 The internal representation	53
5.2.1.5 Virtual functions	53
5.2.2 Function management	54
5.2.3 Possible states of a function	55
5.2.4 Function generation	57
5.2.5 Recurrent Mode	63
5.2.6 Declaration of the GFAS	64
Chapter 6 GFAS Users	67
6.1 Server users	68
6.2 Real time user	72
6.3 Real time context	73
Chapter 7 Hardware specification	77
7.1 GFAS input/output	77
7.2 GFAS memory organization	78
7.2.1 Function table	78
7.2.2 Command Mail Box	78
7.2.3 Status Mail Box	79

Chapter 8 FESA Framework	81
8.1 Framework Overview	81
8.2 Framework packages	86
8.2.1 Cmw	87
8.2.2 Core	87
8.2.3 Interface	89
8.2.4 RT	92
8.2.5 Datastore	96
8.2.6 Persistency	99
8.2.7 Synchronization	100
8.2.8 Exception	101
8.2.9 Utility	102
8.2.10 Logging	102
8.2.11 Record	103
8.2.12 Sorting	103
8.2.13 PLC	105
8.3 Steps to create a FESA Equipment module	106
8.3.1 Design	106
8.3.2 Code	107
8.3.3 Deliver	107
8.3.4 Deploy	107
8.3.5 Instantiation	110
8.3.6 Test	110
Chapter 9 Software Specification	113
9.1 Server part	114
9.2 Real time part	118
Chapter 10 Code design	123
10.1 Fesa design	123
10.2 GFAS classes	129
10.2.1 GENERATED classes	129
10.2.2 SERVER classes	134
10.2.3 RT classes	136
10.2.4COMMON classes	137
Chapter 11 Conclusions	151
Reference	153
Acknowledgments	155

Abstract

This thesis describes the development of a software module for the control of devices in the CERN particle accelerators.

This module is called GFAS (Générateur de Fonction Analogique Simple – Simple Analogue Function Generator), as the board it drives, and its purpose is to generate different functions (i.e., voltage or current waveforms) at different times as needed by the CERN accelerators. Accelerator technicians define a set of different functions by several parameters and a real time system selects the right function at the right time according to the information given by control messages sent by the main timing system.

The software has been developed with the FESA (Front-End Software Architecture) framework developed at CERN.

This work was done by the candidate within a CERN Technical Student Program at the AB (Accelerator and Beams) department, CO (COntrols) group, FC (Front-ends and Communications) section of CERN.

Chapter 1

Introduction

For this thesis I developed a software module for a hardware device that is used in the PS accelerator at CERN and that will be used also in the LEIR, PSB and SPS accelerators (see ch.3). It is based on a previous version still in use on the PS (see ch.3). The module is named GFAS (Générateur de Fonction Analogique Simple - Simple Analogue Function Generator), after the device that it controls. It is used to generate different analogue functions, according the needs of the accelerator, and it is in charge of two main functions:

- the *server activities*, that handle requests to read or modify operating parameters from users of the control room and,
- the *real time activities*, that control the behaviour of the module to satisfy the needs of the accelerators.

This thesis was developed at the AB (Accelerator and Beams) department, CO (COntrols) group, FC (Front-ends and Communications) section.

The AB-CO (Controls) group designs and maintains the control systems driving different accelerators, experimental areas and test facility. It is developing the control systems needed for the commissioning and operation of the LHC and its chain of injectors (see Ch. 3). It provides and supports the numerous control systems that drive the test benches for measuring the LHC magnets, both at CERN and at the magnet manufacturing sites. [1]

The Front-end and Communication section supports the software on the front-end computers and communications equipment. The front-end computers are VME/LynxOS systems. PC/Linux systems are supported for analogue signals observation (Compact PCI) and for gateways (FIP or Ethernet). [1]

The main services offered by AB-CO-FC are:

- To provide the development and operating environment for the control software on the front-end computers.
- To provide drivers and I/O libraries for the I/O boards supported by the AB/CO group.
- To provide support for software development and operation the equipment.
- To provide communication services for the accelerator's equipment. [1]

I will now briefly describe the following chapters.

Chapters 2, 3 and 4 are just an introduction to the physical problems being studied at CERN, to the accelerator complex and to the control system used for LHC accelerators. These chapters are not meant to be exhaustive but just show the context in which I developed my work.

In chapter 5 I have a first description of the GFAS (Simple Analogue Function Generator) and its functionalities. In chapter 6 I analyse the GFAS from the point of view of the users, describing their needs. And, in chapter 7 I show some hardware details needed to develop the software module.

Chapter 8 is totally dedicated to describing the FESA (Front-End Software Architecture) framework and all the tools that I used to realize the module.

Chapter 9 deals with a detailed description of the design that I have chosen for the GFAS software module.

In Chapter 10 I show how I realized my design using the FESA infrastructure and the last chapter deals with the conclusion of my work at CERN.

CAPTER 2

Physic research and CERN

This chapter will be an introduction to CERN and its aims and it is mainly a summary based on reference [2]. There will be also a brief description of the physics problems that are now being studied.

2.1 CERN

CERN is the European Organization for Nuclear Research and it was founded in 1954; the laboratory was one of the first European Joint Ventures and it now includes twenty member states. (Italy was a founding member.)

CERN is also the largest particle physic centre where accelerators and detectors are available to let physicists to study what the matter is made of and what forces hold it together.

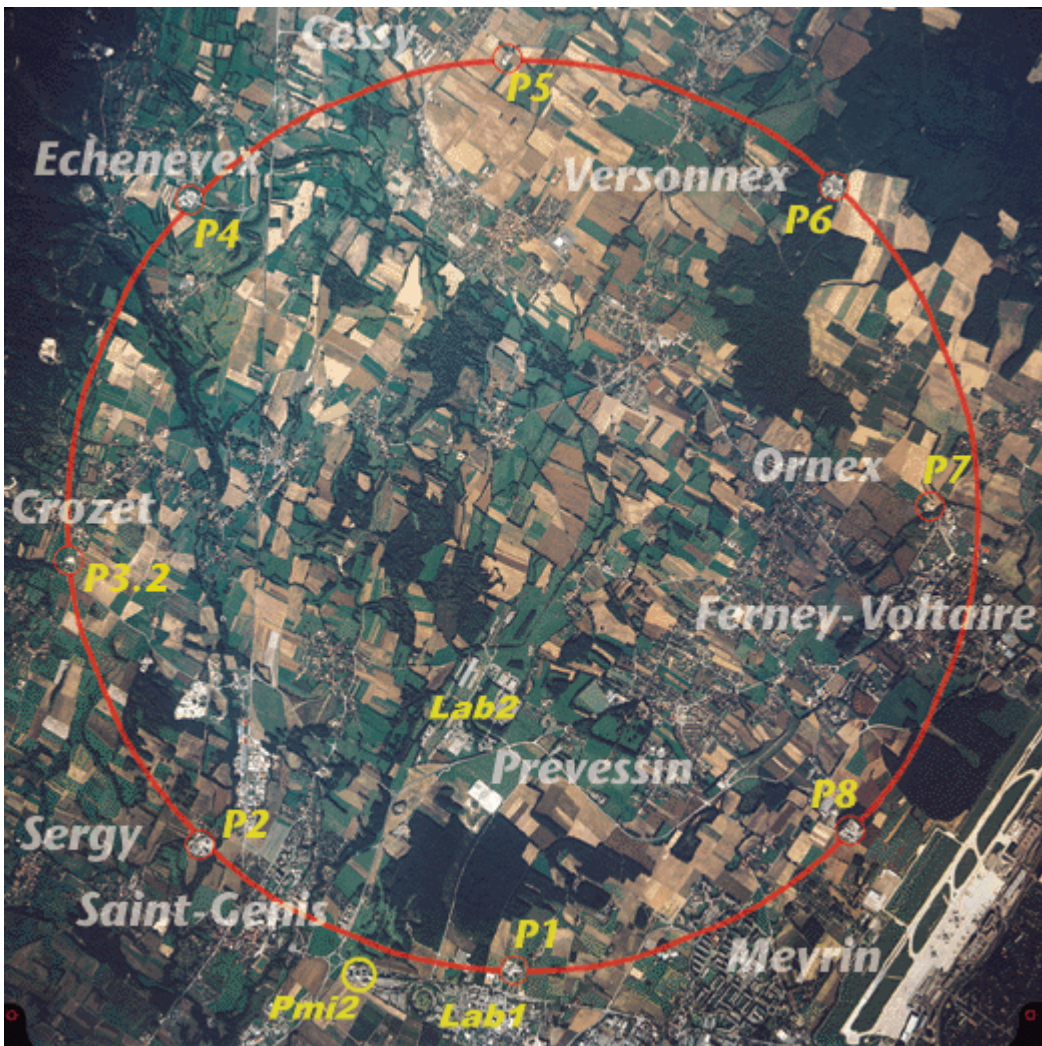


Figure 2.1, CERN map (from www.cern.ch).

The accelerators are needed to accelerate particles to almost the speed of light before they are collided together in high energy reactions. The detectors are used to “see” the particles and to study the results of their collisions.

The accelerator complex at CERN is a series of machines with increasingly higher energies.

The beam from one is injected into the next, which takes over to bring the beam to an even higher energy, and so on.

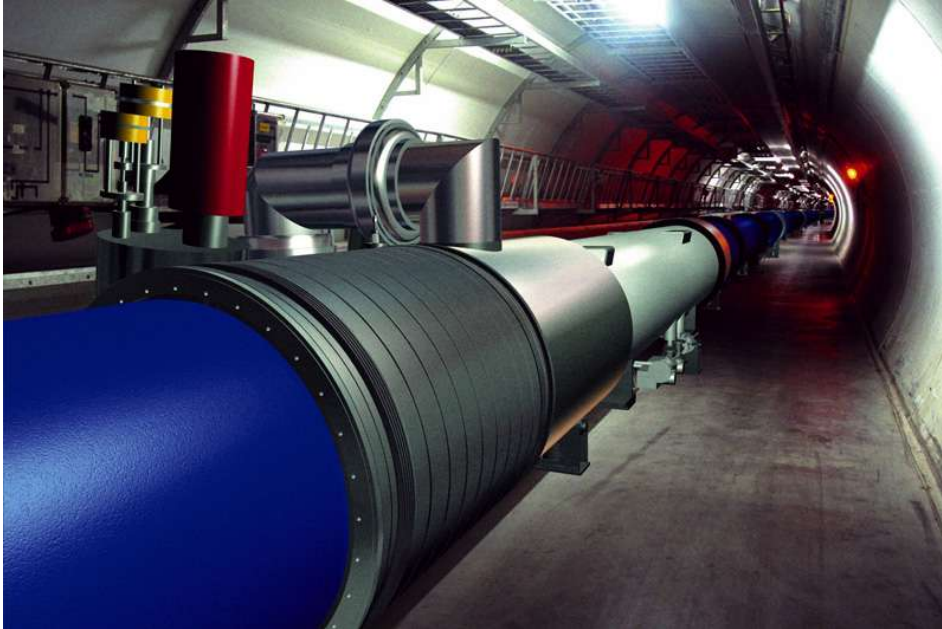


Figure 2.2, LHC photo (from www.cern.ch).

The flagship of the complex will be the Large Hadron Collider (LHC).
I will describe it more detail later.

2.2 Matter and Anti-matter

Let us see what the current studies at CERN are, after a summary of what we should know about particles theory.

2.2.1 Bricks of the Universe: the Building Blocks of Matter

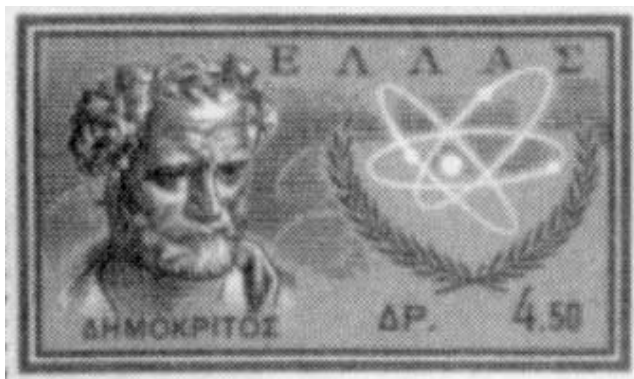
Scientists have found that everything in the Universe is made from a small number of basic building blocks called elementary particles, governed by a few fundamental forces.

There is a wide variety of possible matter in the universe from the few stable particles that make up nearly everything we see around us to very unstable particles that exist for only fractions of a second. In addition, for every one of these particles there is a matching antiparticle.



Figure 2.3, Matter and Antimatter (from www.cern.ch).

All of them coexisted for a few instants after the Big Bang. Since then, only the enormous concentration of energy that can be reached in an accelerator at CERN can bring them back to life. Therefore, studying particle collisions is like "looking back in time", recreating the environment present at the origin of our Universe. The idea that matter should be made out of fundamental 'building blocks' is more than 2000 years old. The blocks were assumed to be simple and structureless and not made of anything smaller.



"... the nature of the perpetual things consist of small particles infinite in number... the particles are so small as to be imperceptible to us, and take all kinds of shapes and all kinds of forms and differences of size. Out of them, like out of elements (earth, air, fire, water) he now lets combine and originate the visible and perceptible bodies..." ~ 450 B.C. Democritus

We know today that all matter in the Universe is built from nearly a hundred different types of atoms, each one made up of Matter particles known as: electrons, protons and neutrons. The electron seems to have no internal structure. Protons and neutrons are composite particles, each containing three quarks. Like the electron, the quarks appear to have no structure. Only two types of quark, called "up" and "down", are needed to build the proton and neutron.

One more structureless particle must be added to complete the picture: a neutral and very light particle called the neutrino. It plays a vital role in reactions that convert neutrons to protons and vice versa. Such reactions allow matter to stay in the stable form we observe and are also important in fuelling the Sun and the other stars.

2.2.2 The Standard Model

The Standard Model of Particles and Forces is a picture of the fundamental structure of matter conceived by physicists after centuries of study. The model requires 12 matter particles and four force carrier.

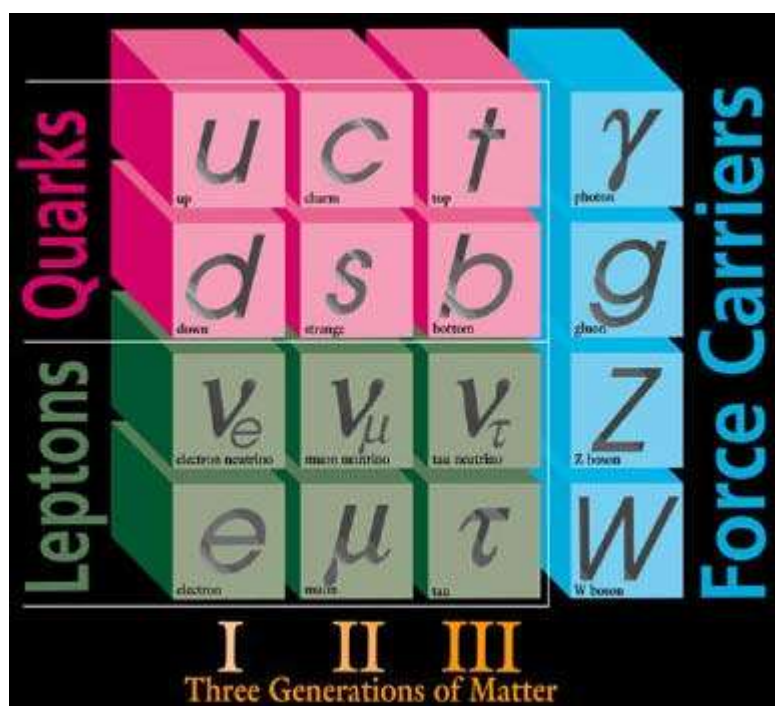


Figure 2.4, Standard Model Chart (from <http://science.howstuffworks.com/atom-smasher.htm>).

Particles

We can group particles into two main families:

- the quarks
- the leptons

There are six quarks, which are usually grouped in three pairs because of their mass and charge proprieties: up/down, charm/strange, and top/bottom.

There are then six leptons, three with a charge and a mass - electron (e^-), muon (μ) and tau (τ) - and three neutral and with very little mass - electron-neutrino (ν_e), muon-neutrino (ν_μ) and tau-neutrino (ν_τ). Again, as their name openly implies, they are grouped to form three pairs (because of some distinctive behaviour during the creation or decay processes).

The (e^-/ν_e) and (up/down) have the lightest mass and are all that is needed to build up the stable matter in the Universe. They make up what is called the first generation of matter.



Figure 2.5, Particle Physics today (from www.cern.ch).

However, they are not all that was needed to build up the Universe; high energy processes produce a large variety of short-lived particles which require the existence of "heavier" pairs, or heavier "generations" of matter. We have then (μ/ν_μ) and (charm/strange) which make up the second generation, while (τ/ν_τ) and (top/bottom) are the third generation.

Recent results at CERN and from astrophysics confirm that there can be no more generations of this type. All second and third generation particles are unstable and quickly decay into stable first generation particles. That's why first generation particles are the only ones we observe in our everyday world.

The standard model includes three types of forces acting among particles: strong, weak and electromagnetic. Gravity is not yet part of the framework.

Forces are communicated between particles by the exchange of special "force-carrying particles" called bosons, which carry discrete amounts of energy from one particle to another. Each force has its own characteristic bosons: the gluon (strong force), the photon (electromagnetic force), the W and Z bosons (weak force).

A big success of the Standard Model is the unification of the electromagnetic and the weak forces into the so-called electroweak force. The achievement is comparable to the unification of the electric and the magnetic forces into a single electromagnetic theory by J.C. Maxwell in the 19th century.

Nowadays, physicists are also trying to include the strong force in a unified scheme called the Grand Unified Theory (GUT). They even contemplate the possibility of including gravity, thus unifying all the forces of Nature in a single "super force". However, much experimental and theoretical work is needed before such a goal is achieved.

2.2.3 Unsolved mysteries and limits of standard model

The Standard Model is by now a well-tested physics theory, used to explain and exactly predict a vast variety of phenomena. High-precision experiments have repeatedly verified subtle predicted effects and it is currently the best description we have of the world of quarks and other particles. However, its present form cannot explain the whole story since there are important questions that the Standard Model cannot answer.

In fact, there are less "ordinary" forms of matter that exist which we can't see:

- **cosmic matter:**

It is the matter coming from the space and it is the result of the collision between energetic atomic nuclei (mainly protons) coming from outer space and the atoms at the top of the earth's atmosphere. Cosmic matter is made up of more components than the atoms: in addition we have some short-lived particles such as the muon, the muon-neutrino, the electron-neutrino and the strange quark.

- **high energy matter:**

It is the matter that we create in our laboratories because working with cosmic ray is difficult. Through the high energy matter we can study particles in a controlled condition using accelerators.

- **antimatter**

Matter and antimatter are perfect opposites; for each of the basic particles of matter, there exists an antiparticle, in which properties such as the electric charge are reversed. The electron, for example, has negative charge, whilst its antiparticle, called the positron, has positive charge. Similarly the positively charged proton has a negatively charged antiparticle, the antiproton. When matter and antimatter meet, they "annihilate" (mutually destroy), and their energy reappears as photons or other particle-antiparticle pairs. Scientists believe that when the Universe originated, about 13.7 billion years ago, equal amounts of matter and antimatter were created. In today's Universe though, there is no antimatter around (beside that produced in high energy collisions). All of it seems to have disappeared, and yet we have no evidence of a mechanism by which this could have happened, leaving scientists with one of the biggest puzzles yet to be solved.

Here we enumerate and describe four questions left unresolved by the Standard Model:

1. What is the origin of the mass of particles?
2. Can the electroweak and the strong forces be unified?
3. What is "Dark matter" made of?
4. Why are there three generations of matter and where did antimatter go?

1) The mass of particles

Particles have a wide range of masses starting from photons and gluons that are completely massless up to the top quark which weighs about the same as a nucleus of gold. Why there is such a range of masses is one of the remaining puzzles of particle physics. Indeed, how particles get a mass at all is not yet properly understood. In the Standard Model, particles gain a mass through the Higgs mechanism (named after theorist Peter Higgs). According to this theory, both matter particles and force carriers interact with a new particle, the Higgs boson. It is the strength of this interaction that gives rise to what we call mass: the stronger the interaction, the greater the mass. Experiments have yet to show whether this theory is correct. The search for the Higgs boson has already begun at the LEP collider at CERN, and this work will continue with CERN's next machine, the Large Hadron Collider (LHC).

2) Unifying the electroweak and the strong forces

One of the major breakthroughs in particle physics in the 1970's was the merging of electricity, magnetism, light and radioactivity - the development of a unified description of the electromagnetic and weak forces. Now theorists are attempting a broader grand unification, which will also include the strong force. Experiments show that the strong force becomes "weaker" in its effects as energies increase. This suggests that at very high energies, the strengths of the electromagnetic, weak and strong force are the same, and the forces are basically indistinguishable. Unfortunately, the energies involved are a thousand million times greater than particle accelerators can reach; they would have existed only in the very early Universe, 10^{-34} seconds after the Big Bang. But grand unified theories also have consequences at lower energies and can thus be tested with present day experiments. They require, for instance, a deep symmetry in the laws of nature, which in turn require the existence of special "superparticles". Some of these could be seen at the LHC accelerator.

3) Dark matter

Measurements in astronomy imply that up to 90% or more of the Universe is not visible (i.e. does not emit electromagnetic radiation). Scientists call this undetectable "stuff" dark matter. Its presence is felt through the gravitational effects on the matter we can see. Stars in galaxies, for example, appear to be moving much faster than they would if they were influenced only by the visible matter in the galaxy. The nature of dark matter and its role in the evolution of the universe are still unknown. Probably it is made of several components, among which are neutrinos, dust, cold gas, and special particles predicted by the grand unification theories but not yet seen, the so called "superparticles". Physicists hope to identify some of the elementary constituents of dark matter at the LHC.

4) Three generations of matter

All matter around us is built from only two types of quarks: "up" and "down", which form neutrons and protons. It also requires two types of leptons: the electron and the electron-neutrino. However, this pattern repeats itself in two heavier "generations", each with two quarks and two leptons. Why are there three generations and why is it that the only one that forms our world is not enough? We do not know. This puzzle though is linked to another, which is also part of the mysteries of the Universe: where did the antimatter go? Experiments in particle physics show that matter and antimatter are always created in equal quantities, indicating that this should also have happened at the Big Bang. If so, why did the antimatter not completely annihilate the matter, leaving only energy (photons) in the Universe? It seems instead that there was some small but significant asymmetry between matter and antimatter in our early universe. This asymmetry could come from an effect called CP-violation. Present understanding of this effect is inextricably tied up with the

existence of three generations. So far, CP-violation has been seen affecting particles that contain quarks of the second-generation (strange), but the LHC should readily produce particles containing the heavier, third-generation "bottom" quark. If the theory is right, such particles should also reveal the symmetry breaking effect of CP violation

These questions are why physicists are searching for "new physics beyond the Standard Model", that they hope will one day lead to a complete "theory of everything".

Chapter 3

Accelerator complex

Here I give a brief description of the accelerator complex at CERN. This is not meant to be exhaustive but should include enough detail to help the reader understand the context in which my work is developed.

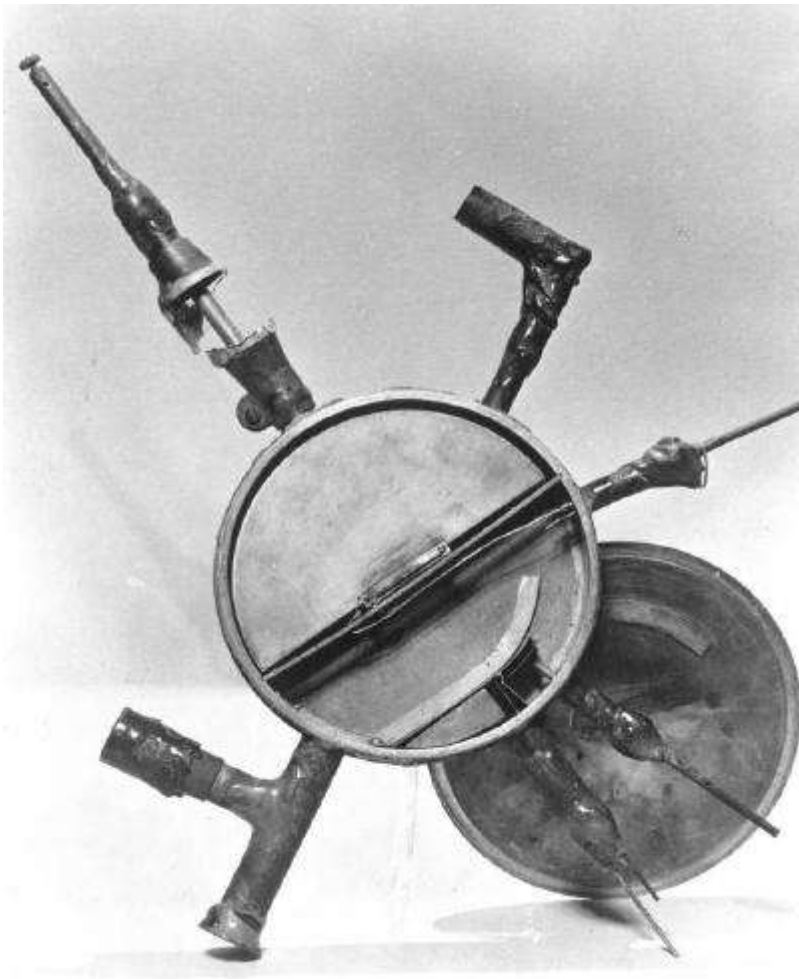


Figure 3.1, the first particle accelerator (cyclotron) developed by Ernest O. Lawrence in 1929 (from <http://science.howstuffworks.com/atom-smasher.htm>).

3.1 Accelerator as the ultimate microscope

CERN physicists investigate the constituents of matter at the subatomic level, where the typical distances are of the order of the femtometre (10^{-15}m) or smaller. Visible light consists of waves, which are characterised by their wavelength and this wavelength lies between 0.4 and 0.8 micrometers (0.000001 metres, or 10^{-6}m). This implies visible light cannot be used to investigate details smaller than about a micrometer. Early in the XXth century, it was discovered that moving particles of matter also behave like waves, whose wavelength would be smaller as their energy increases. Therefore, details smaller than a micron could, for instance, be examined using electrons, provided their energy is large enough. This is the principle of the electron microscope, which is used, among other things, in biology and metallography to look at details of cells or alloys.

The electron microscope is actually a small accelerator, which confers on charged particles - electrons - the energy it takes to make their wavelength small enough to view such details. [3]

To be able to investigate details a billion times smaller, we need to use particles that have energies a billion times higher. This means, oddly enough, that the smaller the details you want to look at, the larger the machine you will have to build. [3]

3.1.1 Brief description of how an accelerator works

An accelerator usually consists of a vacuum chamber surrounded by a long sequence of vacuum pumps, magnets, radio-frequency cavities, high voltage instruments and electronic circuits. Each of these pieces has its specific function. [3]

The vacuum chamber is a metal pipe where air is permanently pumped out to make sure the residual pressure is as low as possible.

Inside the pipe, particles are accelerated by electric fields.

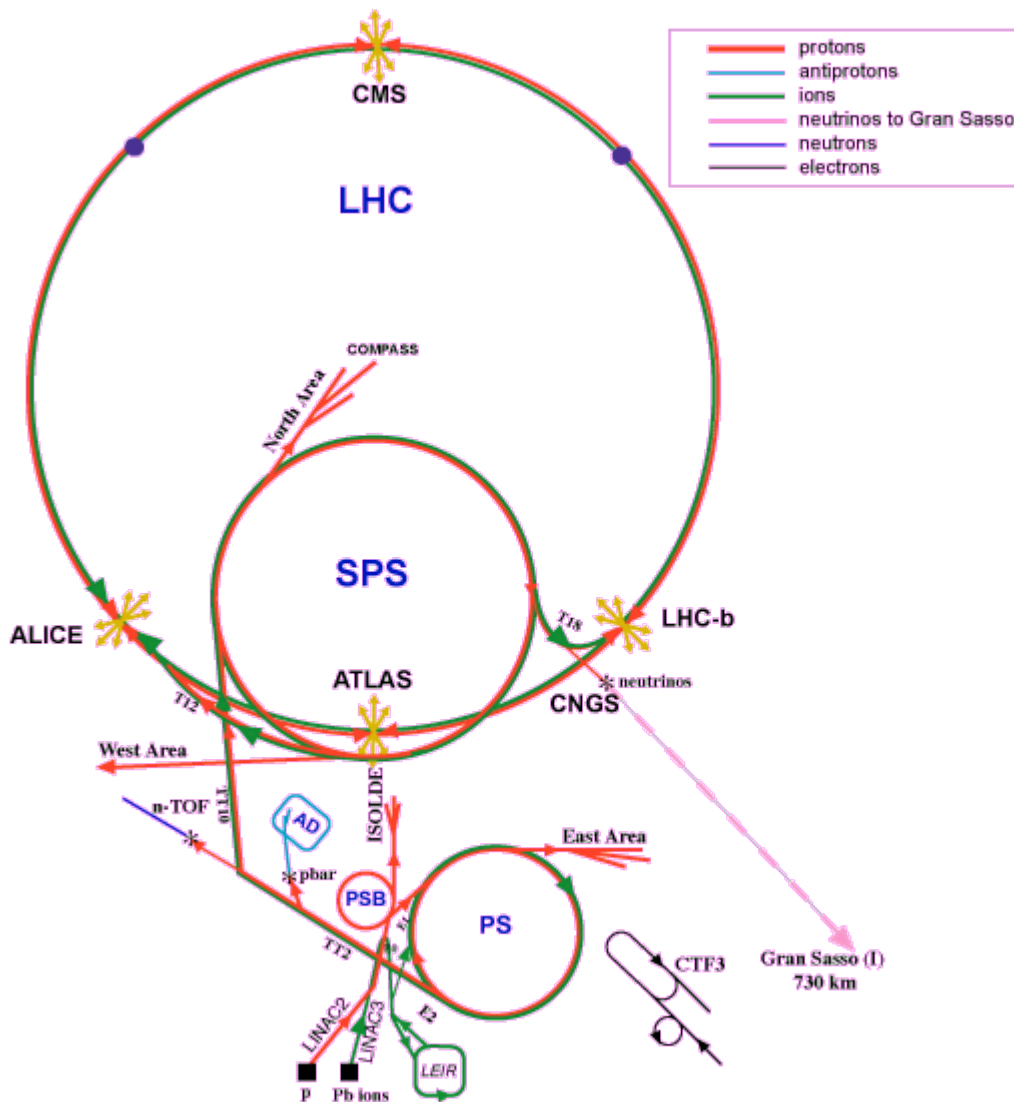
Powerful amplifiers provide intense radio waves that are fed into resonating structures, the Radio-Frequency (RF) cavities. Each time the particles traverse an RF cavity, some of the energy of the radio wave is transferred to them and they are accelerated. To make more effective use of a limited number of RF cavities, accelerator designers can force the particle beam to go through them many times, by curving its trajectory into a closed loop. That is why most accelerators will look roughly circular. [3]

Curving the beam's path is usually achieved by the magnetic field of dipole magnets. This is because the magnetic force exerted on charged particles is always perpendicular to their velocity - perfect for curving the trajectory. The higher the energy of a particle the stronger the field needs to be to bend it. This means that, as the maximum magnetic field is limited (to some 2 Tesla for conventional magnets, some 10 Tesla for superconducting ones), the more powerful a machine is the larger it needs to be. [3]

In addition to just curving the beam, it is also necessary to focus it because just like a beam of light, a particle beam diverges if let to its own. Focussing the beam allows its width and height to be constrained so that it stays inside the vacuum chamber. This is achieved by quadrupole magnets, which act on the beam of charged particles exactly the same way a lens would act on a beam of light. These are merely the basic ingredients needed to make an accelerator. [3]

3.1.2 The CERN accelerator complex

As mentioned earlier, the accelerator complex at CERN is a succession of machines. The next picture shows CERN accelerators:



LHC : Large Handron Collider
SPS: Super Proton Synchroton
AD: Antiproton Decelerator
ISOLDE: Isotope Separation On Line Device
PSB: Proton Synchroton Booster
PS: Proton Synchroton
LINAC: LINear ACcelerator
LEIR: Low Energy Ring
CNGS: Cern Neutrinos to Gran Sasso

Figure 3.1, accelerator complex (from www.cern.ch)

At the beginning we have LINAC 2 and LINAC 3 which are the first accelerators, they accelerate protons and heavy ions, respectively.

After that the beams (both proton and ion) are accelerated in the PS, then in the SPS and finally in the LHC once its construction is complete.

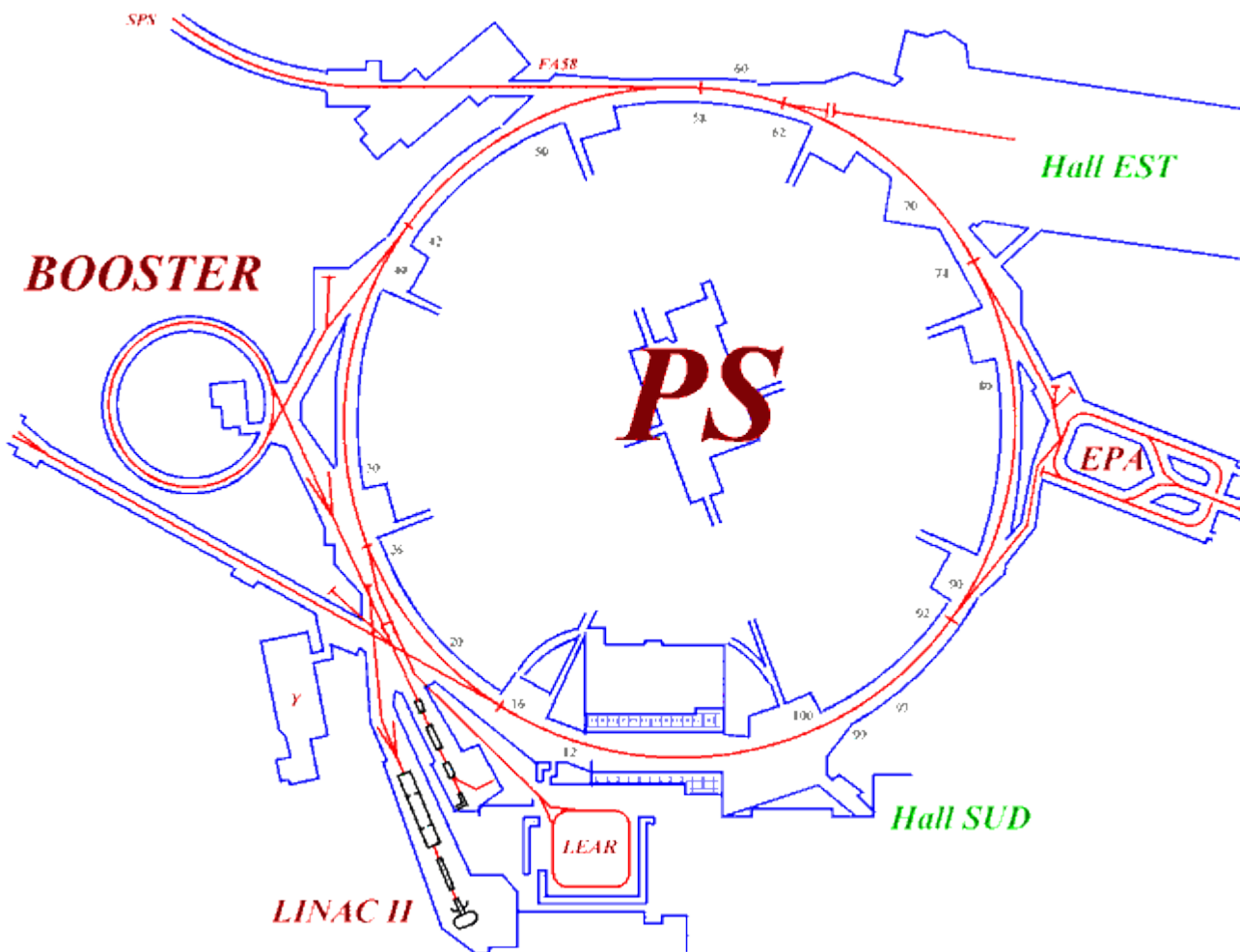


Figure 3.4, PS accelerator (from www.cern.ch).

The Large Hadron Collider (LHC) is a particle accelerator which will probe deeper into matter than ever before. Due to switch on in 2007, it will ultimately collide beams of protons at an energy of 14 TeV . Beams of lead nuclei will be also accelerated, smashing together with a collision energy of 1150 TeV.

A TeV is a unit of energy used in particle physics. 1 TeV is about the energy of motion of a flying mosquito. What makes the LHC so extraordinary is that it squeezes energy into a space about a million times smaller than a mosquito.

In addition, the LHC's injectors have their own experimental hall, where their beams are used for experiments at lower energies.

This experiments are:

- *ISOLDE*, On-Line Isotope Mass Separator, produces radioactive nuclei for a number of applications covering nuclear, atomic, molecular and solid state physics, but also biology and astrophysics. [4]

- *NTOF*, Neutron Time-Of-Flight, produces neutrons in a energy range covering more than eight orders of magnitude (between 1eV and some 250MeV) for experiments in nuclear physics. The time of arrival of the particle at the experiment is a measure of the energy.[4]
- *AD*, the Antiproton Decelerator, is an antimatter factory. Its aim is to produce antihydrogen atoms for particle physicists to study their spectroscopy and test for fundamental symmetries.[4]
- *CNGS*, CERN Neutrino beam to Gran Sasso, sends a high intensity beam of the very elusive neutrinos to a detector located near Rome, 800 km away from CERN.[4]

After a beam of energetic particles is produced in an accelerator, it can either be smashed into the atoms of a fixed target or be made to collide head on with a similar beam coming from the opposite direction. The particles are smashed together with the purpose of probing their structure and using the energy available in the collision to ‘create’ new particles. Lots of new particles are produced. Physicists use detectors to count, trace and characterize each of these particles and study the many possible interaction processes. Physicists need sensitive and specialized instruments, called particle detectors. These consist of many different pieces of equipment, each one able to recognize and measure a special set of particle proprieties, such as charge, mass and energy. [5]

3.2 Experiments at CERN

Experiments at CERN are divided into two main classes:

1) *Collider experiments:*

Collider experiments study the head-on collisions of two beams of particles traveling in opposite directions. In this way, no recoil energy is wasted, and all the energy is available for the production of new particles. In such events, the newly created particles radiate in all directions from the collision point, so the detectors are spherical or, more commonly, cylindrical. [6]

Five experiments have been approved for the LHC:

ATLAS
CMS
ALICE
LHCb
TOTEM



Figure 3.5, CMS Compact Muon Solenoid (from www.cern.ch).

2) Fixed target experiments

Fixed target experiments study what happens when a beam of particles smashes into the atoms of a target. In this configuration, most of the beam energy is used in the recoil of the target and only a small fraction is left to create new particles. In a fixed-target configuration, the particles produced generally fly forwards, so experiments usually have cone shaped detectors, placed downstream of the beam line. [6]

Currently, several fixed target experiments are running at the PS, SPS and AD accelerators.

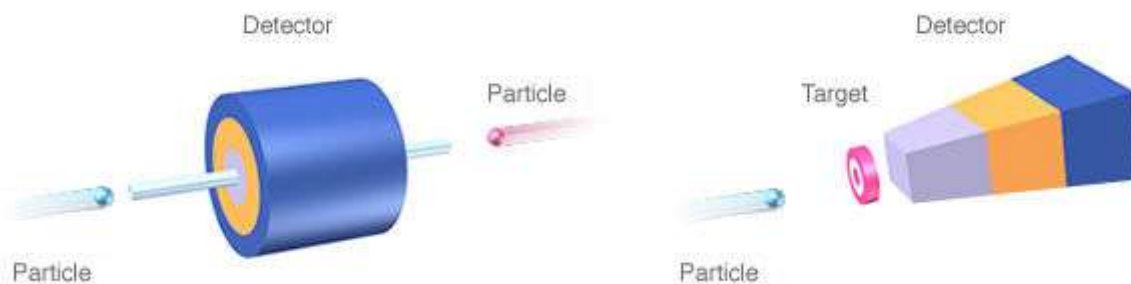


Figure 3.6, CERN detector: collider (on the left) and fixed target (on the right). (From www.cern.ch)

Chapter 4

LHC Control System

The LHC control system will be an extension of the infrastructure in use today for the Injector Chain and the Technical Services. Control projects are under way for additions and upgrades to the existing CERN accelerators.

The LHC control system architecture is largely based on standard components employed worldwide for the control of accelerators and used for years of efficient operations at the CERN injectors (PS, SPS) and of the LEP machine.

Particle accelerators use terminal devices that can be sensors, actuators or a combination of both. From a remote control room, operators are able to control and optimise the beam across the control system infrastructure, which consists of layers of hardware, software and communication.

In circular machines, there is a linear relationship between the particles' momentum p and the mean magnetic field that the accelerator has to provide to keep the circular orbit.

In figure 4.1 the vertical axis can thus be interpreted as either the magnetic field B or the beam momentum.

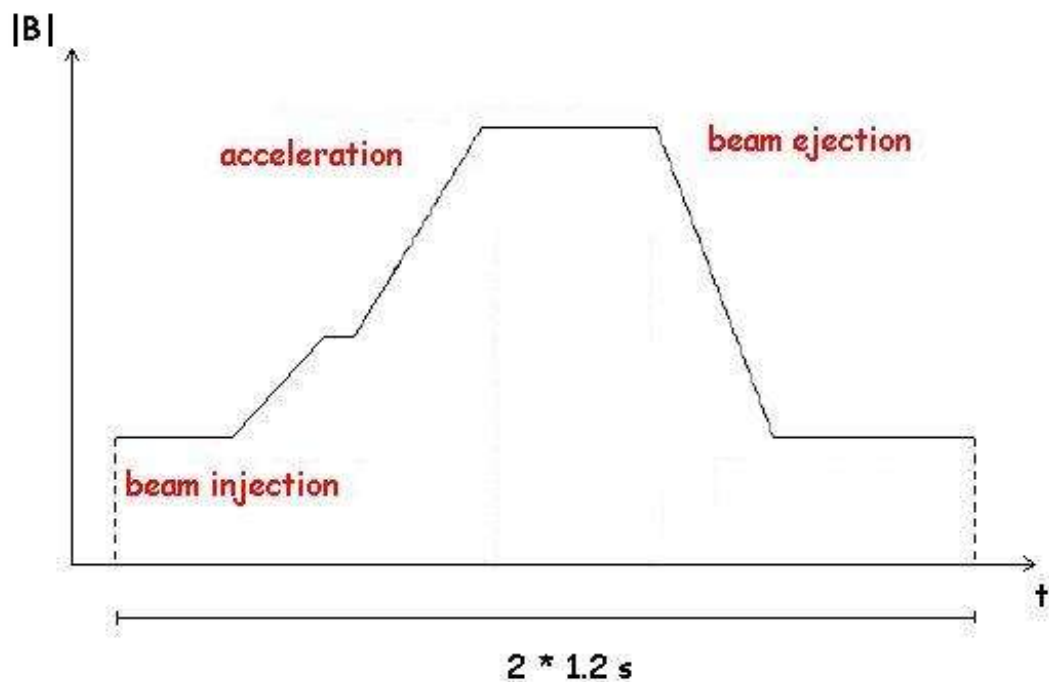


Figure 4.1 , magnetic field in a generic circular accelerator.

The relationship between momentum and energy is:

$$E = \sqrt{p^2 c^2 + m^2 c^4}$$

Basically E and p are almost the same at high energies for light particles, when $p \gg m$, such as electrons (in units where $c=1$) but they are quite different for heavier particles such as protons.

Particles with higher momentum describe circles of larger radius if the magnetic field is kept constant.

Unfortunately today's technology poses the upper limit on the magnetic field, that is why accelerators are also growing in radius to increase the momentum of the particles.

This is also the reason for the chain of accelerators with increasing radius and why the beam is injected in the next accelerator when the magnetic field reach a certain value.

We can identify three phases in the control of a beam:

- *beam injection*, when the beam is injected from the previous accelerator into the current one;
- *beam acceleration*, when the magnetic field is increased and the beam is accelerated;
- *beam ejection*, when the maximum momentum is reached and there is the need for an accelerator of bigger radius and the beam is ejected from the current accelerator to the next one.

To have the right magnetic field at the right time is a key issue for the control system, in order to accelerate a beam without losing it. The operation should be well synchronized and to understand synchronization issues we need the concepts of *cycle* and *telegrams* (discussed in par. 4.4).

4.1 Overall Architecture

The LHC control system has equipments communicating through the CERN Technical Network, a flat Gigabit Ethernet network using the TCP-IP protocol.

Applications controlling the LHC beams must handle a great variety of tasks such as visualization of data and significant computation, together with database and equipment access. These applications rely on several services such as security, transactions (to make sure that complex operations are performed atomically), and remote access or resource management. Those requirements dictate a move towards a modular and distributed architecture. Such architecture offers a clear separation between the GUI (the user interface), the control (the core business of the application), the model (the abstract model of the accelerator control) and the devices (the physics equipment) that are controlled.

An architecture made of three layers is being implemented:

- *Resource tier* (the equipment level):

the various actuators, sensors and measurement devices are interfaced to the control system through three different types of front-end computers :

- o *VME* computers dealing with high performance acquisitions and real-time processing; these employ a large variety of I/O modules.
- o *Programmable Logic Controllers* (PLCs) driving various sorts of industrial actuators and sensors for systems.
- o *PC based gateways* interfacing systems where a large quantity of identical equipment is controlled through fieldbuses. [7]

- *Middle tier* (Unix Server Host):

it is the heart of the control system, powerful UNIX servers host the operational files and run the LHC applications:

- o *Application servers* hosting the software required to operate the LHC beams and running the Supervisory Control and Data Acquisition (SCADA) systems.
- o *Data servers* containing the LHC layout and the controls configuration as well as all the machine settings needed to operate the machine or to diagnose machine behaviors.
- o *Central timing* which provides the cycling information of the whole complex of machines involved in the production of the LHC beam and the timestamp reference.

Processes running in these servers will communicate with the equipment access machines using various mechanisms and protocols such as the Common Object Request Broker Architecture (CORBA) and TCP-Modbus.[7]

- *Presentation tier* (the control room level):

At this level we have consoles running the Graphical User Interfaces (GUI) that allow machine operators to control and optimise the LHC beams and to supervise the state of the system. Dedicated fixed displays will also provide real-time summaries of key machine parameters. The consoles are responsible for the GUI applications and translate operator's actions into command invocations in the middle-tier. [7]

The middle-tier, through its centralized and shared processing power is in charge of accessing databases and accelerator devices. It is responsible for providing all services and for coordinating the client applications running on the operator consoles, it ensures the coherency of operator actions and hides details of resources and it enforces separation between presentation and application logic and provides shared services to many clients.

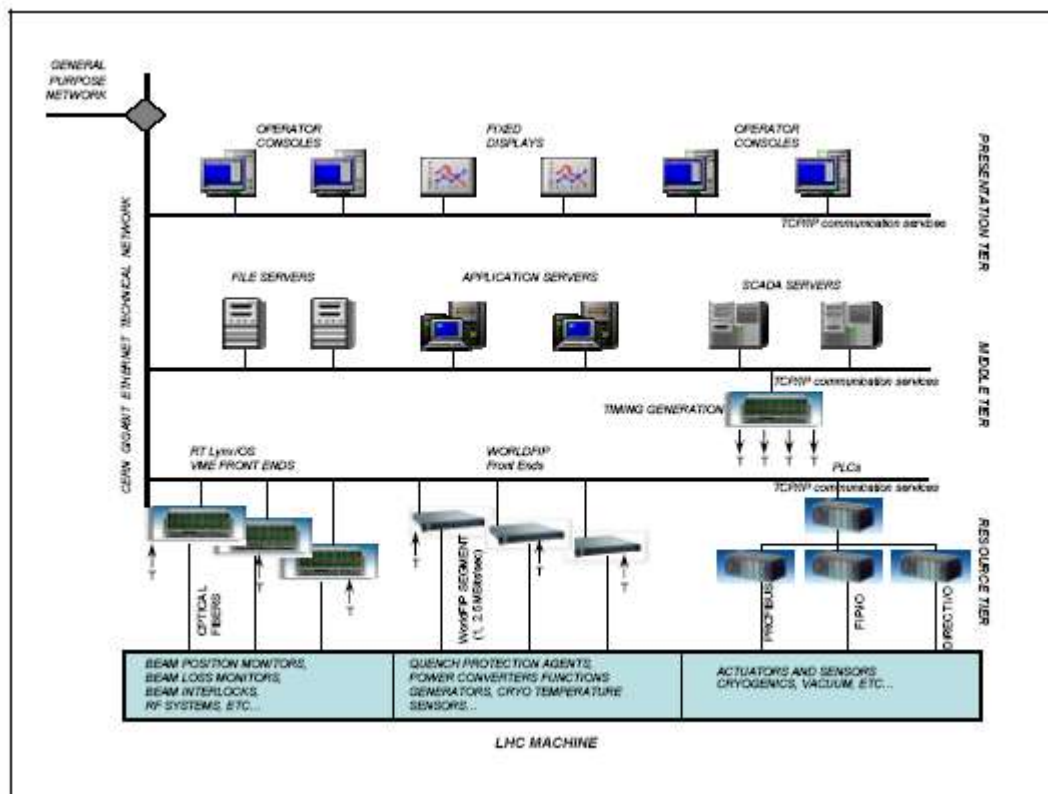


Figure 4.2, LHC architecture (from LHC Design Report vol.1 cap. 14).

To implement this new programming model, a platform supporting the middle-tier is needed. With such a platform, developers can write code for the accelerator controls components (such as

parameter management, settings generation, cycle handling, trim management) without writing system level services.

The platform of choice today is the J2EE which is an industrial standard defined by a set of specifications and APIs, implemented by many vendors. It is based on the Java programming language and on a component model known as Enterprise JavaBeans (EJB). Components are the key technology to write a modular object-oriented distributed application. EJB components are "Lego pieces" that implement a set of interfaces allowing them to run inside a so-called "EJB container". Developers write accelerator-specific code in the form of EJB components and the EJB container provides the necessary infrastructure to run them: remote accessibility, components management, concurrency support, fault tolerance, high availability, scalability, resource pooling, transaction management and security. [7]

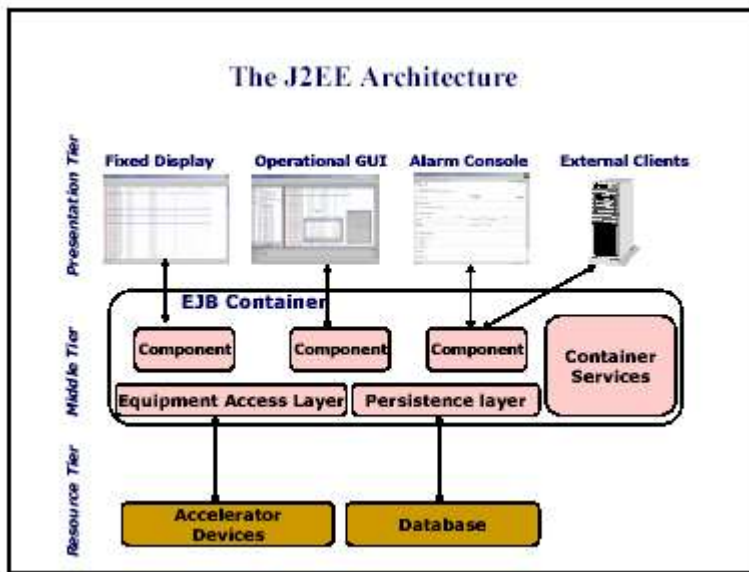


Figure 4.3, The J2EE Architecture. (From LHC Design Report vol.1 cap. 14)

4.2 Network

The control system for LHC relies on the CERN Technical Network. This network uses IP addresses of the form 172.18.xxx.xxx. This address prefix prevents any direct communication between the technical network and the Internet. The infrastructure is interconnected with the General Purpose Network so that communications from within the Organization are still possible; however, the technical network can be completely isolated if required. The technical network is a highly sub netted, routed network that is a high performance redundant backbone. This reduces parasitic traffic and keeps the overall structure maintainable. The basic distribution is based on a redundant optical fiber Gigabit Ethernet backbone. Two redundant routers, located in the CERN computer center and in the Prévessin control room, serve as core of the network. These two central routers are interconnected in a redundant way to a series of regional routers located in different buildings.

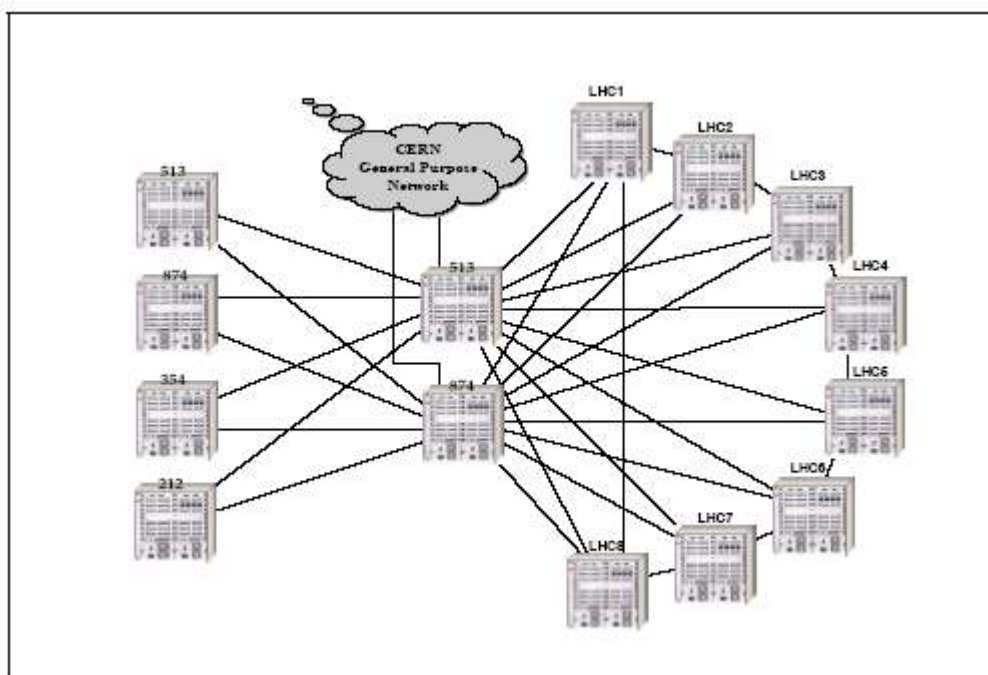


Figure 4.4, The CERN Technical network (from LHC Design Report vol.1 cap. 14).

Because the technical network can be completely isolated from the external world, independent dedicated redundant name and time servers have been implemented. These servers are installed at different locations and connected to different regional routers to ensure full redundancy of the system. It must be noted that because the technical network infrastructure is interconnected with the general purpose network, security break-ins can be attempted on the devices connected to this infrastructure. End nodes security survey and updates must not be forgotten. [7]

4.3 Equipment access

Let us see the different element that we have at the resource level.

4.3.1 The VME and PC Front End Computers

Most of the non-industrial accelerator hardware in the PS, SPS and LHC sites is already connected to the control system via VME or PC based Front End Computers (FEC). These systems run a real-time operating system called LynxOS from LynuxWorks or Red Hat Linux. There will be several hundred FECs distributed in the surface buildings of the LHC and in some cases in the underground areas. Wherever possible they will be diskless to increase reliability and will boot over the network. They will be connected to the remote reboot and terminal server systems so that they can be managed remotely by the operators and system administrators. A set of commercial or CERN made hardware interface boards is being standardized for the LHC, together with the necessary device drivers (e.g. timing generators, timing receivers, beam interlock controllers, digital acquisition boards, WFIP cards, digitisers, etc.). The type of hardware (VME or PC), as well as the Operating System (Linux or LynxOS), will be selected according to the performance needed and to the availability of the specific drivers. [7]

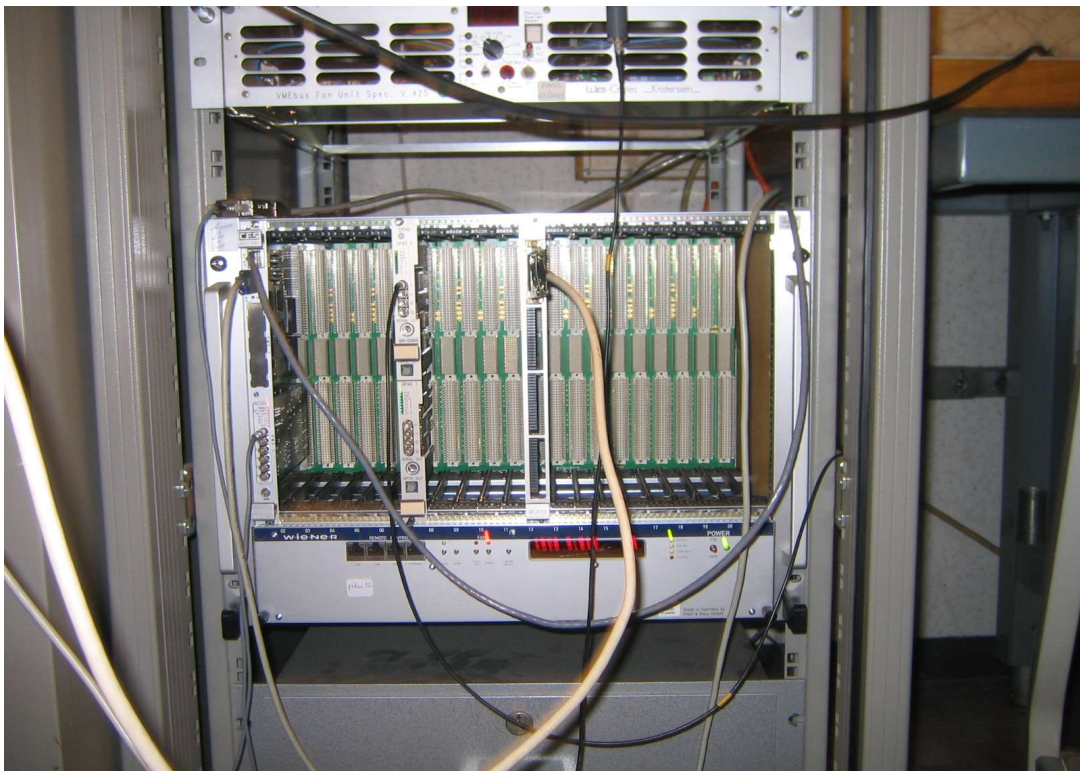


Figure 4.5, Psdsc12: an example of VME.

These FECs will execute the equipment access software which is part of the resource tier. This software accesses data of the hardware interface boards and provides it either via subscription or command/response to any software agent in the control system. A dedicated FEC software framework (Sec. 14.7.1) has been developed in order to simplify and standardize the software running in the FECs.

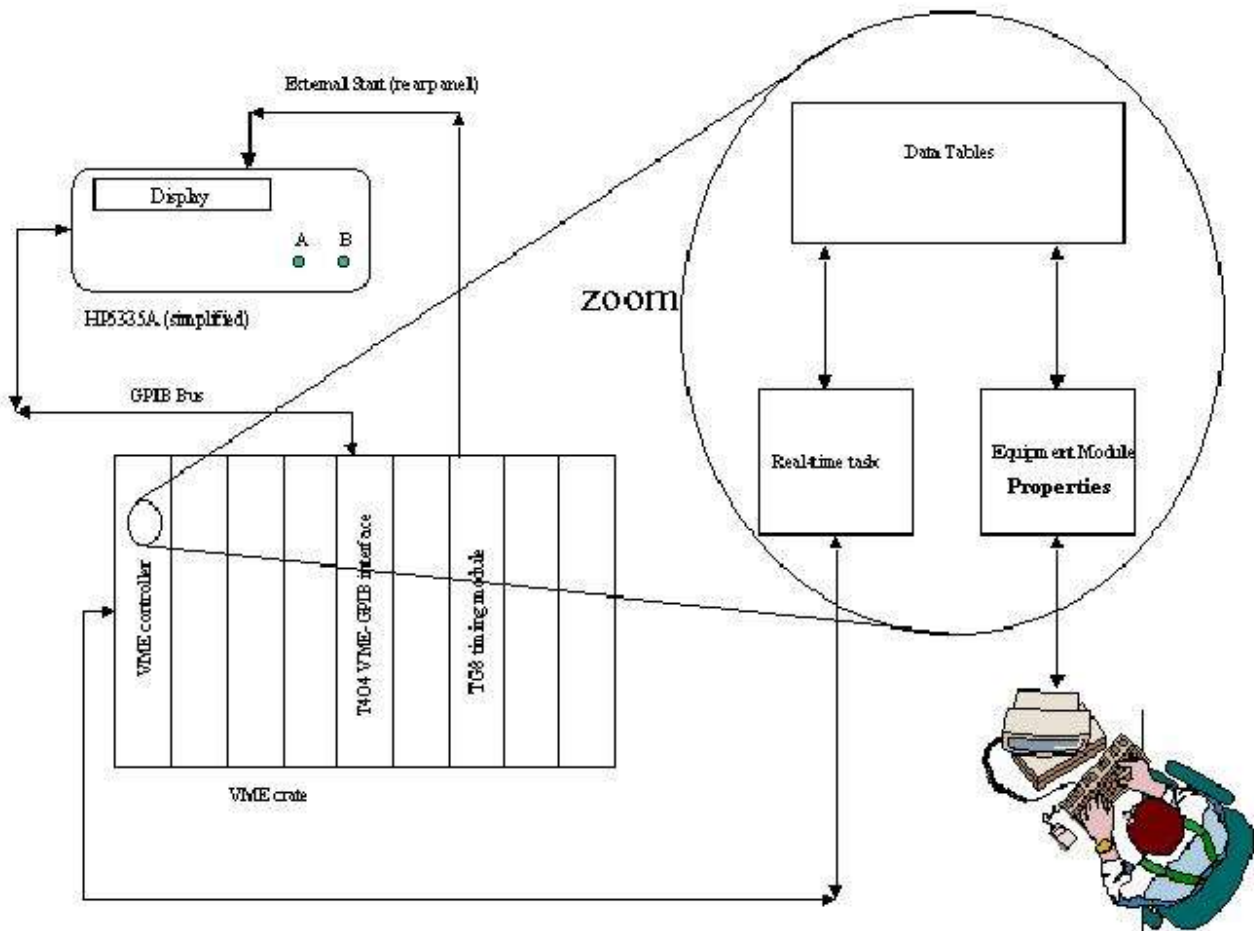


Figure 4.6, A VME crate (from PS/CO/Note 99-22 Tech.).

4.3.2 The PLCs

PLCs are increasingly used for controlling industrial equipment for LHC. This type of controller can be chosen when the process is not synchronised to accelerator timing and when the sampling period required is not smaller than 100 msec. PLCs offer ease of programming process logic via standard high-level languages, are generally part of a complete subsystem including supervision and in a few cases they are accessed through VME FECs.[7]

4.3.3 The Supported Fieldbuses

The two fieldbuses used in the LHC accelerator, namely WorldFIP and Profibus, are part of the three fieldbuses recommended and supported at CERN. These fieldbuses typically allow longer distance and more robust protocols than Ethernet. In addition they are the only means to connect equipment located in the LHC tunnel or other radioactive areas. [7]

4.3.4 Servers and operator consoles

The upper layers of the LHC control system will be executed on application servers that meet the requirements of the LHC applications software, and interfaced through operation consoles fixed displays. The servers will be used to run the LHC middle-tier software, to host operational programs and data files, and also to offer specific services (web services for operation, fixed displays, SCADA servers, Database servers, etc.).

The servers will run the Linux operating system. Emphasis will be put on the hardware reliability and availability issues by selecting multi-CPU architectures with redundant and hot swappable power supplies, discs and fans. Mirroring and RAID techniques will be used to ensure data integrity and error recovery. Fixed displays and operator consoles will both be based on desktop PCs with a large amount of memory. They will run GUI applications from their local disks and use one to three screens to display the data. At present the choice of the Operating System between Linux and Windows has not yet been made. More than 20 fixed displays and around 10 consoles will be needed in the control room to operate the LHC machine. Around 50 servers located nearby and in the LHC surface buildings will be used to deploy and run the operational LHC software. [7]

4.4 Machine timing and UTC (Central Beam and Cycle Management)

As we have already said at the beginning of the chapter, the LHC beam production involves a long chain of injectors which need to be tightly synchronized. A beam, from the controlling point of view, can be considered a collection of *cycles* (see below) in different accelerators that are synchronized to obtain an end product. A *cycle* is the representation of the magnetic field behaviour in a particular accelerator from the injection of beam to the ejection. In the figure 4.7 the typical cycles of PS and PBS are shown. Each accelerator has a particular type of cycle, it can be described by a particular set of values and different kind of events can trigger the beam treatment. The set of all the cycles used to accelerate a beam is called super-cycle.

The description of cycles in real time is carried over a special network through *telegrams* which carry information about the current and the next cycle, that are needed by real time applications and by some application programs.

Telegrams are currently made up of a set of 32-bit words, each one corresponding to a *group* (see below). Each telegram is made of a number of words ranging from 20 to 30 depending on the accelerator. The first 16 bits of each word are descriptive: 4 for the machine (accelerator) concerned by the word, 4 for the type of event and 8 bits for the group number. A group number encodes such information as particle type, end user, beam destination ... and a cycle number is identified by these group numbers. The remaining 16 bits are the value for the given group. For example, in a PS telegram, the particle type group could have a value of 1 for protons, 2 for antiprotons and so on. Telegrams are broadcast in every cycle but the words that make them up can come at any time and in any order and all the information carried is used to provide detailed information for sequencing real-time tasks running in the FECs and for setting up equipment. [8]

The groups that we can have in a telegram depend on the accelerator that we are considering. Some groups are present in telegrams of every accelerator, like USER (who owns the cycle), PARTY (the type of particle produced), DEST (the primary destination of the beam produced)... and other groups are peculiar to specific accelerators.

The different types of beam to be produced for LHC are only a subset of the beams that the injectors have to produce, in sequences of typically a few tens of seconds.

The composition of these sequences changes many times a day and the transition between different sequences has to be seamless.[7] To manage the settings of the machines and to synchronize precisely the beam transfer from one injector to the other up to the LHC, a Central Beam and Cycle Manager (CBCM) is required. Timing is probably one of the most difficult subjects in the PS controls system.

The CBCM will:

- Deliver General Machine Timing (GMT),
- Provide Beam Synchronous Timings (BST),
- Elaborate the *telegram* specific to each machine and broadcast it over the GMT networks.

To achieve extreme reliability, a CBCM is running in parallel on two different machines, with a selector module capable of seamlessly switching between them.[7] We can take as an example what happens when particles go from the Booster (PSB) to the PS. (Figure 4.7), the values on the vertical axis can be interpreted as the magnetic field or as the momentum.

The particles start being accelerated in the booster at time T1. When they reach their maximum momentum at T2, the PSB can start injecting particles into the PS. This is done in the small flattop between T2 and T3. Then the PS starts accelerating the particles until it reaches its top magnetic

field. Between T2 and T4 particles can be ejected from the PS to the SPS or to other experimental areas. At time T4, the ejection is finished and the magnetic field in the PS starts to decrease to start another cycle. Different cycles can be aimed at different targets and contain different particles at different energies. The length of the booster cycle is currently 1.2 seconds, while that of the PS can be any number of these 1.2 second basic periods. This means that after the booster has executed a cycle for the PS, it has time to work for other machines or physics complexes such as the ISOLDE experimental area, executing what we call parasitic cycles. [8]

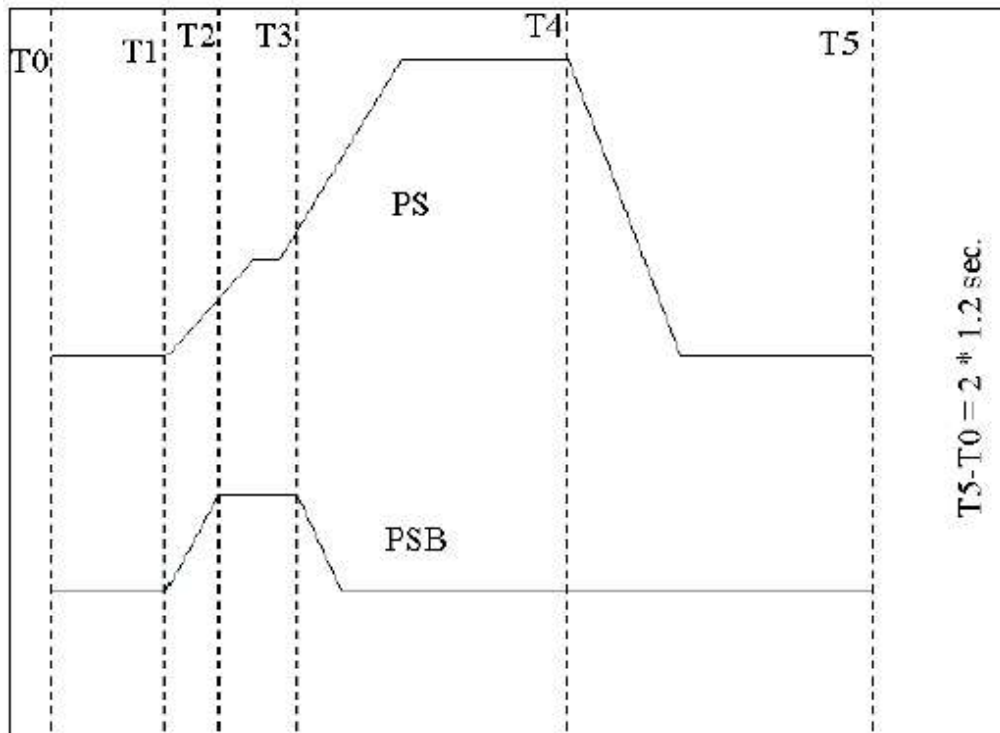


Figure 4.7, from PSB to PS (from PS/CO/Note 99-22 Tech.).

Now that we understand cycles, let's explore the concept of beam as PS timing specialists understand it. The need for a beam point of view arises when you deal with more than one accelerator. Particles must be transferred from one machine to another and some central intelligence (in our case the MTG -Main Time Generator-) must coordinate these transfers. The whole situation can be schematically represented as a directed graph with nodes and arcs linking them. Each accelerator is a node in that directed graph and the arc between two nodes represents the physical transfer line from one accelerator to another. Particles flow through arcs and spend some time in each node. [8]

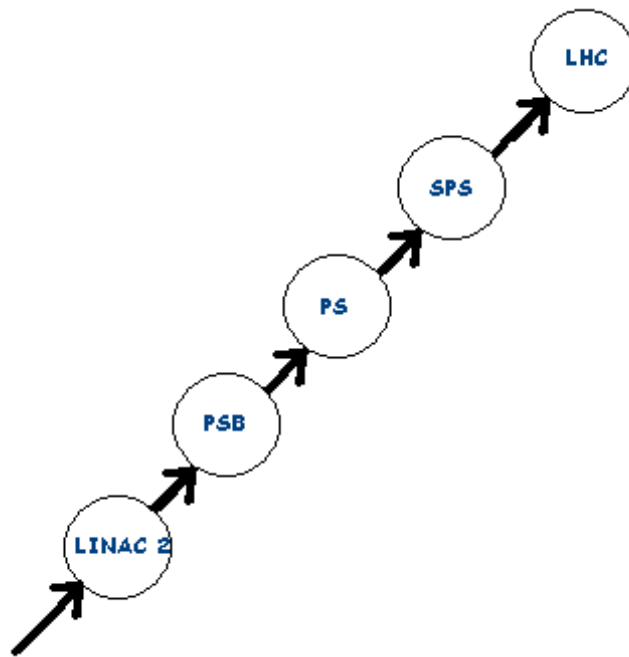


Figure 4.8, example of a graph to represent the proton beam.

In this context, a beam is a path through the directed graph, and the MTG deals with beams in this “PS controls” sense of the word. The `plsedit` program is used by machine operators to change the supercycle, that is, the sequence of cycles to be repeated over and over.[8] This program is able to receive input in graphical form and to translate it into what we call a Beam Coordination Diagram (BCD), which is just data that will allow the MTG to generate telegrams. [8]

In fact, things are a bit more complicated, a complete sequence without much detail would look like this:

- First of all, an operator “draws” the supercycles with `plsedit`.
- Another program then uses ORACLE-stored data to merge the BCD generated by `plsedit` with information for the AD, which is a loosely coupled machine. After doing this, it sends the merged BCD to the MTG’s main task, a real-time task running on the DSC where the MTG is sitting.
- With this information, the real-time task automatically generates a pseudo-assembly code program and downloads it to the MTG’s internal processor every 1.2 seconds.
- The MTG card then starts interpreting the downloaded code, which is just information on what events to distribute and when to send them.

These operations need an external timing reference, so that the whole timing system agrees on what 1.2 second intervals to take. The general reference is a 10 MHz clock sent by a satellite and received by a GPS module. [8] This 10 MHz clock is transformed into a 1 kHz clock and then again into the 1.2 second clock.

4.5 Data Management

The LHC Machine will be of unparalleled complexity in terms of number of components and in the manipulation of operational parameters. A large number of signals will be made available in the control room, including over 100,000 from the cryogenics, machine protection and beam monitoring systems. Over 1600 electrical circuits will power a wide variety of magnet systems. The quantity of data to be recorded for diagnostics after a “beam abort” has been estimated as a few gigabytes; during Hardware Commissioning it is anticipated that 1 terabyte of information will be accumulated. Operation of the machine will rely on stringent control of the hardware settings. [7] A complete beam operational history will be created in order to build a long term understanding of the accelerators operation.

4.5.1 Offline and Online Data Repositories

The LHC Reference Database is being populated during the machine construction in order to manage the essential information for integration studies, installation and hardware commissioning. This offline Database will be one of the sources of data that is required to configure the online LHC control room application software.[7]

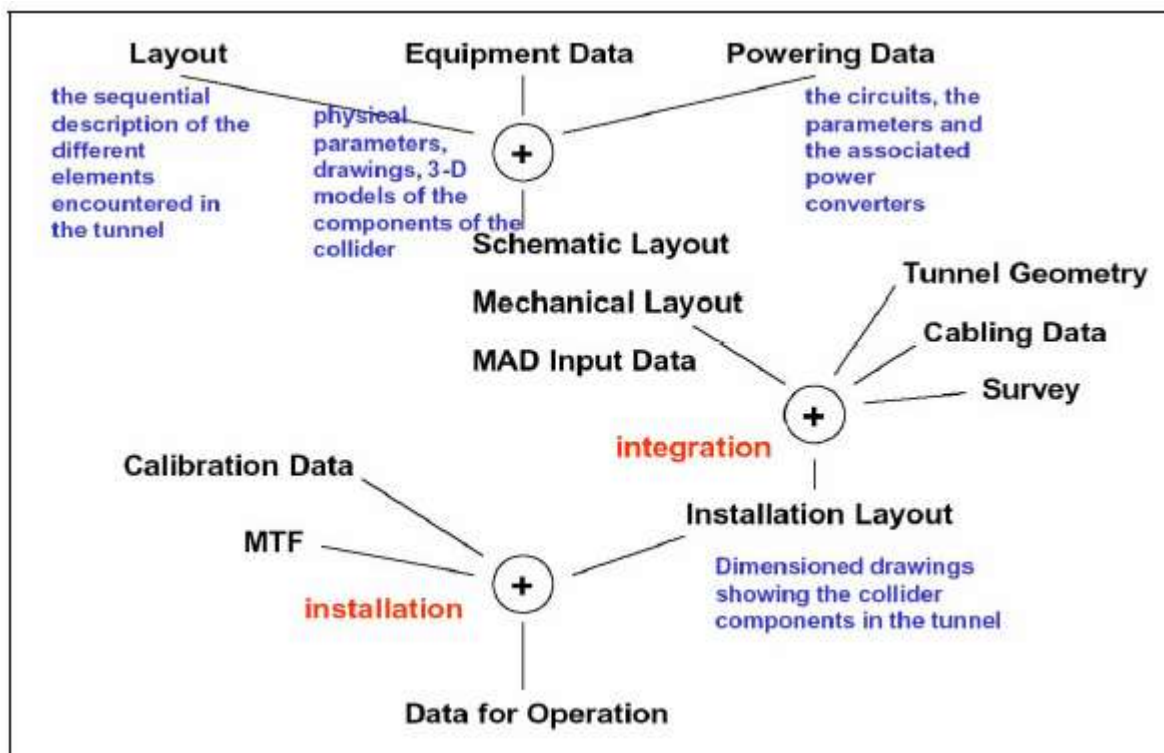


Figure 4.9, LHC Reference Database (from LHC Design Report vol.1 cap. 14).

While the Reference Database will be an offline source of information during machine operation, several online databases - alarm archives, periodic data logging, Post Mortem event archive and accelerators settings management - will be required to capture the operational history. We will talk about alarm, logging and post-mortem later.

4.5.2 Control System Configuration

The control system is composed of a large number of application programs, a middleware layer, VME and PC-based FECs with control software, industrial PLCs, fieldbuses, and hardware interface modules. These components have shared characteristics, associated with their type or class, and specific addresses and parameters associated with their instances.

All these data will be stored and described in an Oracle relational database, the Controls Configuration Database. [7]

This centrally managed configuration information permits the use of generic software on all levels.

A layer of Java and C++ interfaces will enable all control room software to be data-driven, by fetching at runtime the layout, interconnection, access methods, and parameters of remotely controlled LHC equipment. All FECs can be automatically configured and bootstrapped with files generated from the database. Thanks to a generic design, the database also holds the controls configuration of the injection accelerator chain of the PS and SPS complexes. This common source of data will make it possible to have uniform controls data management throughout the accelerator chain, as well as effective sharing of tools and methods that use this data. [7]

On top of the Configuration Database, a Web browser service will allow exploration of all the information about the underlying controls hardware and software. [7]

4.6 Communication and software frameworks

In the following sections I'm going to introduce the framework used in the front-end computers and the controls middleware.

4.6.1 FESA: FEC Software Framework

The software running in the LHC FECs will be developed using a specific framework, known as FESA. This framework is a complete environment for the equipment specialists to design, develop, test and deploy real-time control software for the FECs. This framework is also the new standard for the LHC injector chain. The recurrent components of the equipment control software have been formalized into a generic model. It defines the structure of the real-time part of the software which handles hardware and interrupts together with the structure of the task server that handles external requests. In order to enhance the productivity and the maintainability of the code, a clear separation between the generic and the equipment specific parts is enforced and automatic code generation is extensively used. As a result, the ratio between the generic/generated code and the developer specific code is usually around 10 to 1. [7] An object-oriented design and a C++ implementation have been used to support this approach.

FESA provides different tools to help the developers. First of all we have tools to define internal data structures, syntax of the external commands and binding between hardware events or external requests.

Also system-level services are provided for the operation of the FECs: handling of errors from the equipment software, monitoring of the processes, etc. [7] These dedicated services are made for preserving the real-time behavior of the FECs and they support remote control for adjusting parameters, such as the trace level of a task, or executing corrective action like restarting a process. Finally the developers and the users (equipment-specialists, operators, and maintenance team) are provided with a set of interactive tools, test programs and Java libraries which allow immediate testing of all the functionality of the software during its development and operation.

These facilities are all generated in an automatic manner, without any specific code required.

The FESA framework is built for LynxOS and Linux platforms and software tools are written in Java and are thus platform independent. No commercial software requiring a run-time license has been used. The FESA framework will be described later in chapter 10.

4.6.2 Controls Middleware

The Controls Middleware (CMW) is the ensemble of protocols, Application Programming Interfaces (API) and software frameworks, which allow seamless communication between the software entities of the control system. Two conceptual models are supported: the device access model and the messaging model. The device access model is mainly used in the communication between the resource and middle tiers while the messaging model is mainly used within the middle tier or between the middle tier and applications running in the presentation tier. [7]

Device Access Model

The typical use of this communication model is between Java applications running in the middle tier and CMW equipment servers running in FECs. These processes communicate together through the Remote Device Access (RDA) API which is available for both Java and C++ languages. [7]

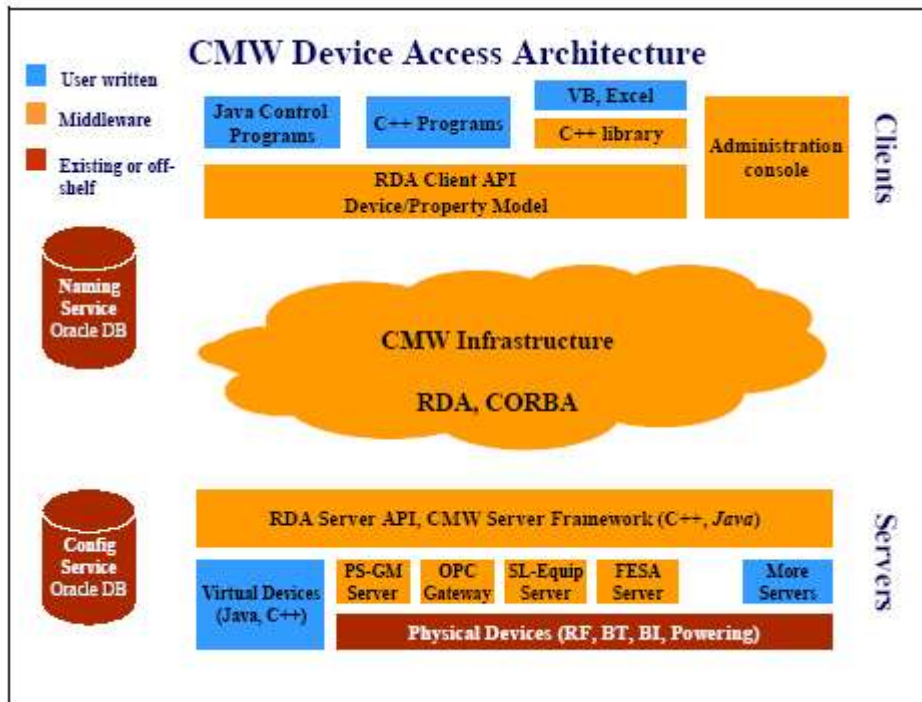


Figure 4.10, The Device Access Model (from LHC Design Report vol.1 Ch. 14).

Within the device access model all accelerator equipment can be accessed as “devices”, which are named entities within the control system. Conceptually each device has a number of properties that can be used to retrieve its state or to control it by setting a required value. We can have Get and Set operations: these can be either synchronous or asynchronous and it is also possible to monitor changes to the property via listeners (callbacks). It is often necessary to specify under which conditions the operation has to take place, for instance to which cycle of the machine the operation applies. These conditions are commonly referred to as context or filter. [7]

Messaging Model

Complementary to the device access model, the messaging model allows any software process to send and receive asynchronous messages in a loosely coupled way. Various application programs for accelerator controls naturally exchange data via asynchronous messages which are potentially of interest for multiple consumers. The LHC control system software relies on the Java Message Service (JMS) as the messaging solution for Java based controls application. The JMS specification adds a standard and vendor-independent API that enables the development of portable, message-based applications in the Java programming language. Moreover, JMS is a strategic technology for the Java 2 Platform Enterprise Edition (J2EE). The LHC Alarm Service is a typical example of a messaging system relying on the publish-and-subscribe paradigm: it subscribes to alarm notifications being published by several processes and, after processing, this redistributes the result to dedicated consoles and any other controls software component that subscribed to the appropriate alarm content hierarchy.

4.7 Control Room Software

Now I will give brief introduction of the presentation level: all the facilities needed by the users in the control room.

4.7.1 Software for LHC Beam Operation

The LHC control room applications will be based on a set of common control system services and components, in order to avoid duplication of effort and solutions. The operational software development process relies on common development tools, guidelines and procedures for the construction, testing, integration, deployment and change management.[7]

The LHC aims at injecting, accelerating and then colliding beams with well controlled beam parameters in an efficient, reliable and reproducible manner. This is a non-trivial task, since the small aperture, the high stored beam energy and the sensitivity of the machine to beam losses impose very tight accelerator physics constraints. The superconducting magnets will generate field errors that have large static and dynamic components. Thus, the destructive power of the LHC beams and the low tolerances to beam losses will place very stringent demands on the LHC control system. [7]

4.7.2 The Software Development Process

The current software development follows the Unified Software Development Process, a practical software process model followed by many industrial OO projects. Java is the programming language chosen for the implementation of the high-level services and control room applications, as it enables platform independent development. XML is playing an increasingly important role in the exchange of data. A suite of selected software tools for code optimization, quality insurance and testing is provided to the developers to guarantee the quality of the control room software. Software configuration management facilities are provided on top of the archive engine (RCS) to provide version management services, making it possible to trace and identify any component of an operational application, and to deliver consistent operational software releases. In addition tools for code building and distribution are available to release operational software components in a multi-versioned repository residing on dedicated file servers. Operational software, once released, is deployed locally to the operators' consoles using standard Java distributed deployment technology (e.g. Java Web Start), which guarantees automatic delivery of software updates without operator intervention. Local deployment ensures the speed and performance required by operations. [7]

4.7.3 Services for operations

We have different services offered for operation, such as the transmission of analogue signals, the alarms, the logging, the post mortem, the support for real-Time feedback loops and some other monitoring and diagnostic tools.

4.7.3.1 Analogue Signals Transmission

While most signals from the LHC will be digitized for internal usage by the respective accelerator systems there are a number of them which are required for visualization during tuning and measurement. These include some 200 signals, mainly in the 0 to 10 kHz range from the RF system and over 300 signals with a typical bandwidth of 50 MHz from the kicker systems for injection and beam extraction (see introduction of Ch 4). In the injector chain, the nAos system is successfully used to fulfill the need for coherent acquisition and display of analogue signals. The system's main

feature is the digitization of signals in acquisition crates as close as possible to their sources, therefore ensuring optimal fidelity. Whenever possible, in order to save costly oscilloscope modules, signals are multiplexed allowing some 100 signals to be serviced by one crate containing four two-channel oscilloscope modules. The triggers for all the oscilloscope modules are elaborated from the GMT distribution, ensuring a precise time-correlation between signals. The digitized signals are sent through Ethernet to control room consoles, where a GUI application allows operators to monitor them in four-channel “virtual oscilloscope” screens. The nAos system is currently being upgraded to more modern, commercial hardware and software platforms.[7]

The new version of the system is called Open Analogue Signals Information System (OASIS). It will implement all the above features, including arbitration of resource (i.e. oscilloscopes), allocation to clients and the additional functionality for the LHC requirements. This approach will ensure that signals from the injector chain may also be integrated into the LHC operation; the older, lower energy machines have been instrumented to support this system. As with all LHC systems OASIS must provide information to the Logging and Post Mortem systems.[7] This offers a very attractive possibility of flexibly connecting signals which should prove very useful in understanding malfunctioning of complex systems. Analogue signals will be monitored by Compact PCI-based acquisition systems; the total number of crates will depend on the amount of multiplexing achievable with regard to the number of clients. [7]



Figure 4.11, Snapshot of nAos GUI (from LHC Design Report vol.1 cap. 14).

4.7.3.2 Alarms

Operating the LHC will require many services to help both operators and equipment specialists to understand the many complex situations which will arise during maintenance or operation of the CERN accelerator complex. The detection, collection, management and distribution of information concerning abnormal situations ranging from severe alarm states to warning states, hereafter referred to as Fault States (FS), will require one global system. The system will accept information concerning any problem associated with the CERN accelerator complex and offer this information, in a structured way, to any interested party. The core part of this system will be the LHC Alarm Service (LASER). As a result of a detailed technology survey, a 3-tier architecture was chosen and is now being implemented according to the J2EE Specification. [7]

Fault State (FS) detection is performed in the resource tier by surveillance software written by equipment specialists, application programmers and operators. The LASER system offers a standard interface for these sources to push a FS, by means of either a C or Java API based on the messaging system. FS time stamping at the point of detection, using UTC time, down to the microsecond level if needed, will be crucial in correlating data. The middle-tier will collect the FS by subscribing to all FS sources. [7] A number of services and facilities will be offered at this level including:

- FS persistence;
- Logging of FS transitions using the LHC Logging facility;
- FS dependency and reduction management;
- FS browsing and interaction;
- Administration of the overall system;
- User and configuration management;
- Scaling, using clustering facilities;
- FS distribution according to a FS hierarchy, representing the interest of users.

Finally, the presentation tier will cater clients interested in those services. The major user of this interface will be the alarm console, which will be used by equipment groups and control centers to select, receive and display FS, and in order to access all LASER services. Important configuration facilities will allow the alarm console to be personalized. [7]

4.7.3.3 Logging

The process of capturing and recording information on the operation of the LHC is generally referred to as “LHC Logging”. Logging of information in a centralized storage medium is an operational requirement and has proven to be essential in previous projects. The information of interest that needs to be logged is various and heterogeneous, but usually concerns values of parameters that evolve over time, for example cryogenic temperatures, vacuum pressures, magnetic field measurements, bunch intensities, or beam profiles. The total number of logging variables will be in the order of 10^5 - 10^6 with logging frequencies up to 0.1 Hz. The main purpose of LHC logging is to improve the machine performance and that of the individual subsystems. Therefore the recorded data will be stored and kept accessible over consecutive years, in order to permit comparisons of historical data, off-line analysis to search for data correlations, and compilation of operational statistics. For each of the logged records of the data variables, the date-time value is a key attribute, and therefore correct “time stamping” is vital for the exactness of the data. For time stamping, the usage of UTC is endorsed throughout the LHC Logging system, with microsecond precision where applicable. In order to enable effective exploitation of the logged data by users (equipment engineers, control room operators, machine physicists and CERN managers), the LHC Logging system will employ web enabled interfaces to visualize the information. This data is also used by operational data system such as Alarms and Post-Mortem. The functional specification and architectural design of the LHC Logging system have been documented. The central storage medium will be a dedicated Oracle database instance. Input of the logging data and its description will be done through formally defined XML files, which may be generated from Java, C or C++ programs. [7]

4.7.3.4 Post Mortem

Many tens of thousands of signals will be available for the operators and engineers to interpret the circumstances of Beam and Power Aborts. The Quench Protection System alone will provide around 80000 signals for Alarms, Logging and Post Mortem purposes. Recovery from a perturbation to the nominal magnet cycling will require a minimum of 2 hours (7 TeV back to 7

TeV) which precludes learning by trial and error as practiced with non-superconducting machines. [7] The LHC Post Mortem System is required as a suitable diagnostic tool. The purpose of this system is:

- To ensure comprehensive monitoring of machine protection systems.
- To improve the operational efficiency of the LHC by:
 - Providing diagnostics of power and beam aborts - the aim is to quickly identify the origin of failures in order to initiate appropriate actions and restore operation,
 - Building long term understanding of accelerator operation,
 - Providing diagnostics for beam losses resulting from equipment malfunction.
- To explain the mechanism if damage occurs.

To achieve these aims the post mortem data must be complete and coherent across systems. Whilst it is intended to make full use of diagnostics from Alarms and Logging these systems will not provide all the facilities required for the understanding of quenches and beam losses in the LHC. In particular the underlying hardware must capture transient data pertaining to conditions prevailing before, during and after such events. These transients may be relatively slow when they are provoked by quenches; indeed the Logging system will gather some pertinent information, such as cryogenic temperatures. In order to satisfy the essential aims of being complete and coherent across all LHC systems the following essential ingredients are required.

Every LHC equipment and diagnostics system must implement a circular Post Mortem buffer of appropriate depth holding the latest data (for example 1000 turns for beam instrumentation). This data must be time stamped using a common clock with a precision related to the corresponding process. The Post Mortem buffers must be frozen by the Post Mortem timing event or by self-triggering in the case of the protection systems. Data must be self describing so that it can be managed by the event builder and analyzed by generic tools. [7]

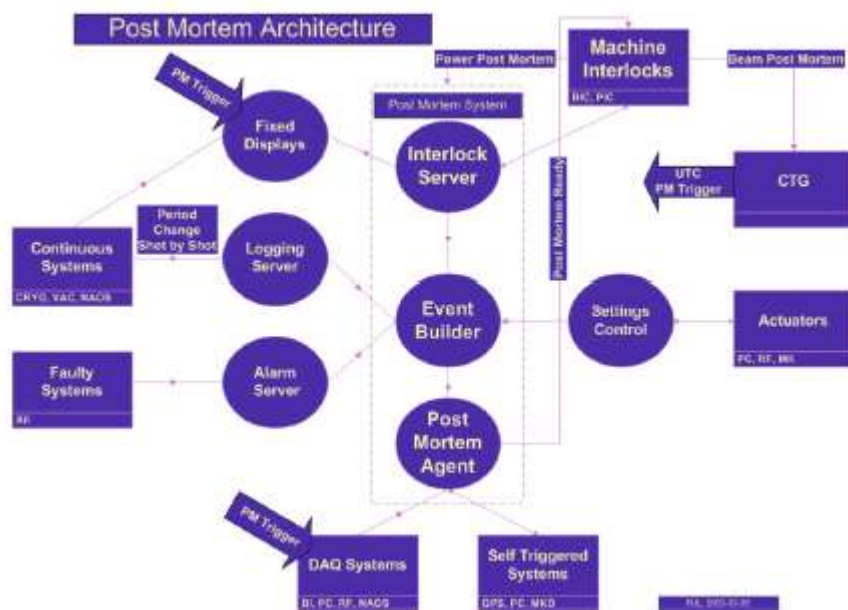


Figure 4.12, *The LHC Post Mortem Architecture* (from LHC Design Report vol.1 cap. 14).

The Post Mortem events will be large - several gigabytes - and therefore analysis must be automatic in order to generate digested information for operators. After analysis the information must be stored in order to build up the longer term history and understanding of the machine. The most relevant data will be stored for the lifetime of the LHC. [7]

4.7.3.5 Support for Real-Time Feedback Loops

To operate the LHC efficiently with high intensity beams, a number of key beam parameters must be stabilized using digital real-time feedbacks and fast feed-forward from reference magnet measurements. It is anticipated that an 80% reduction of the persistent current decay effects can be achieved with feed forward but that real-time feedbacks will be required for orbit, tune and chromaticity. The expected sampling frequencies of the feedbacks lie in the range of 10 to 100 Hz. For chromaticity the main difficulty lies presently in the possibility to measure this parameter in a non-destructive way at a reasonably high frequency. It has been shown that a properly designed feedback loop can control orbit distortions during injection and at the beginning of the ramp. For example, a proportional–integral control loop that samples at 10 Hz and that has a total time delay of 100 ms can reduce orbit errors at a frequency of 0.1 Hz with a gain of 8. A considerably better performance can be obtained when the sampling frequency is increased and/or when the total time delay is reduced. From that point of view, robust operation of feedbacks mainly concerns networks and FECs. A global orbit feedback in particular will involve more than 100 FECs of the beam instrumentation and power converters. Despite the large size of the LHC ring and the distribution of the equipment over the machine, the network performance must be stable and guarantee delays in the range of a few milliseconds for data sent between any points of the LHC. Large variations of time delays due to the computation, control or communication must be avoided, if necessary by allocating dedicated resources to feedback operation. The LHC FECs will provide good real-time support based on the LynxOS operating system. Networking topology and resources will be organized to meet the service requirements. At present no specific infrastructure for feedback is foreseen. [7]

4.7.3.6 Control System Monitoring and Diagnostic Tools

The monitoring tools used by accelerator operators, experimental areas operators, technical services operators and system administrators allow operators to detect, understand and repair faults or remotely reboot the computers. The current system monitors a few hundred UNIX, Linux and LynxOS computers, 365 days a year, 24 hours a day. In order to cope with the growing and evolving controls environment as well as to conform to the new application software standards a new design of the monitoring tools is currently underway. A new surveillance program will inject alarms directly into the LASER system. The new operator interface will be based on the same technology as the LHC Alarm Console and will allow the operator to view the state of all the computers being monitored, either good or bad. The operator can then select one of them to perform more detailed checks or restart a failing program or even reboot the computer. The LASER archive will store all these monitoring alarms. So, after a major event for example a UPS failure on an LHC site, the operators and system administrators will be able to check at what time computers stopped functioning due to loss of power and when the control system was restored. As part of the upgrade, the surveillance will be extended to Sun Workstations, Windows Servers and PLCs in order to cover all the main components of the LHC Control System. [7]

Chapter 5

Introduction to the GFAS



This chapter deals with the main topic of my work, which is the function generator known as GFAS (Générateur de Fonction Analogique Simple - Simple Analogue Function Generator).

In the accelerator complex (see Ch 3) there is the need to define several functions, used as reference value in the control activities.

The functions depend both on time and on the different cycle specified by the telegram (see par 4.4).

5.1 Assignment

The development of the GFAS (Simple Analogue Function Generator) equipment module is based on the previous one which was realized through the previous GM framework. (PS/CO/Note 93-108).

The FESA facilities made it possible to realize an easier and user friendly interface, thanks to the new features which were not in the GM framework.

The GFAS board has two independent channels and each of them can produce a function according to some parameters given by the user.

Figure 5.1, GFAS board.

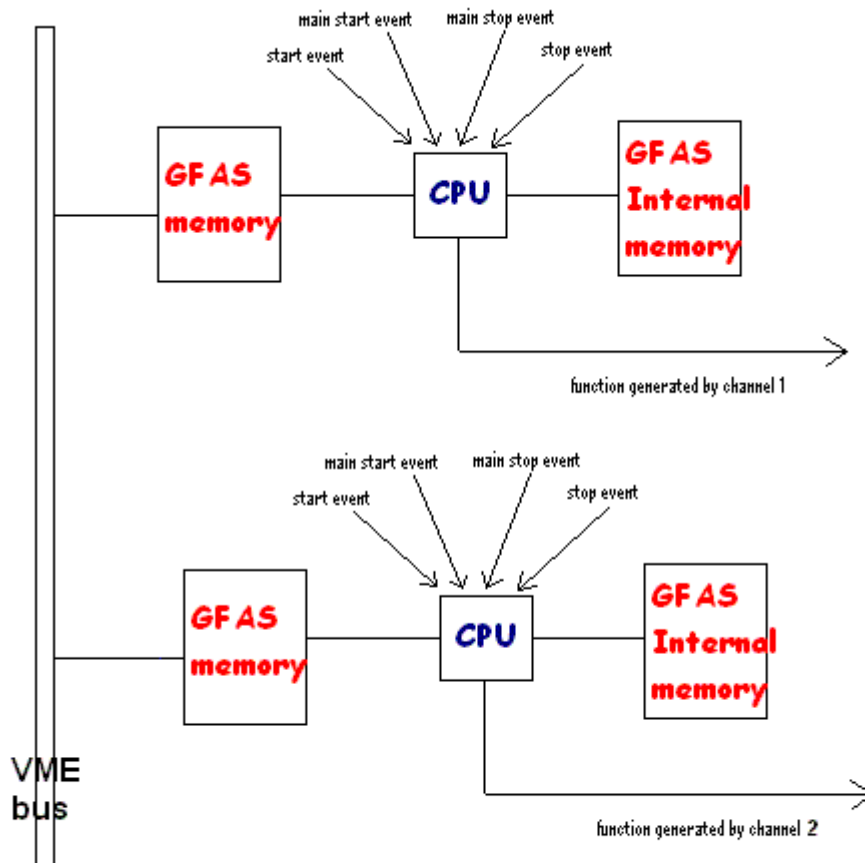


Figure 5.1, GFAS main organization.

The functions to be generated are piecewise linearly approximated and are loaded as tables into the part of the GFAS memory which is accessible via the VME ("GFAS memory"). It is possible to load up to 32 functions for each channel and each function is made of up to 510 vectors. The CPU of the GFAS reads these functions, converts them and puts them in converted form into the internal working memory which is not accessible from the VME ("Internal GFAS memory"). According to the electrical input needed by the equipment that uses the GFAS, we have to choose the right connection type among the four possible for GFAS:

- Bipolar DAC,
- Unipolar DAC,
- Bipolar digital,
- Unipolar digital.

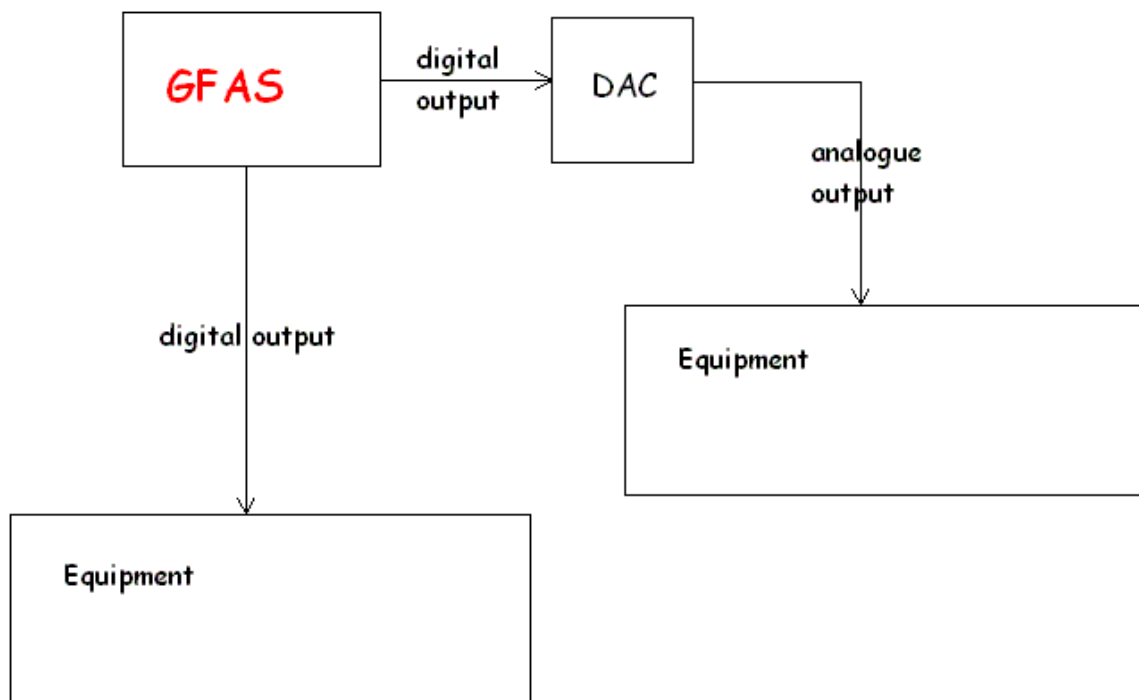


Figure 4.2, Different way to connect the equipment.

Some types of equipment need digital GFAS output and other types need the analogue output (obtained adding a DAC - Digital Analogue Converter - between the GFAS and the equipment). Also we have to know if we are using as physical values just the positive range (unipolar, e.g. voltage between 0 and 10V) or both positive and negative (bipolar, e.g. voltage between -10V and 10V).

The GFAS can work in eight different modes:

- normal,
- DAC,
- low jitter,
- normal recurrent,
- low jitter recurrent,
- test1,
- test2,
- no function generation.

The difference between 'normal' and 'low jitter' is related to the timing of the GFAS output: it is an abrupt and unwanted variation of the delay needed to send out the output value. The GFAS CPU needs ca. 48 μ s before it sends it out. In the 'normal mode' the jitter for this time is 2.8 μ s in the worst case and 2.1 μ s in the average case and with the 'low jitter mode' we have the worst case of 1 μ s and the average one of 0.6 μ s. If the second channel of the GFAS is not operating the two worst case values are reduced to 2.6 μ s for normal mode and 0.8 μ s for low jitter mode.

We can choose the recurrent mode if we have a periodic function and we want to repeat more than once the set of vectors which are loaded. We can specify the number of times, which we want to

repeat a particular set, by the value of recurrence; we have the same value for all the functions of the same channel.

Using the DAC mode, the GFAS a fixed value (DAC value) is put repeatedly onto the output. Escaping from this state is possible only by changing the mode.

For Test 1 the generated function is a positive and negative triangle spanning the full range of the DAC. The duration of the two triangles is 20ms.

For Test 2 we have a sequence of software start and stop events to test them (we will introduce them later).

5.2 Requirements

Now I am going to describe in detail the function representations, the management of functions, their different states and other details concerning the GFAS module.

5.2.1 Representation of a function

We can have different representations of the same function, let us describe them.

5.2.1.1 Basic Function

A function is represented as a sequence of *vectors* and a vector describes a linear ramp or flat that has to be generated in an interval of time.

According to this basic representation a vector is made of a pair made of the final instant of the interval and the corresponding value of the function in the appropriate unit (e.g. Volt, Ampere, Hertz ...). Then a function can be specified by the number of vectors, the initial value and the sequence of vectors. There is a vector with a particular meaning, it is the internal stop vector (we will talk about it later in par. 5.2.4.) and it is the one with 0 as time value.

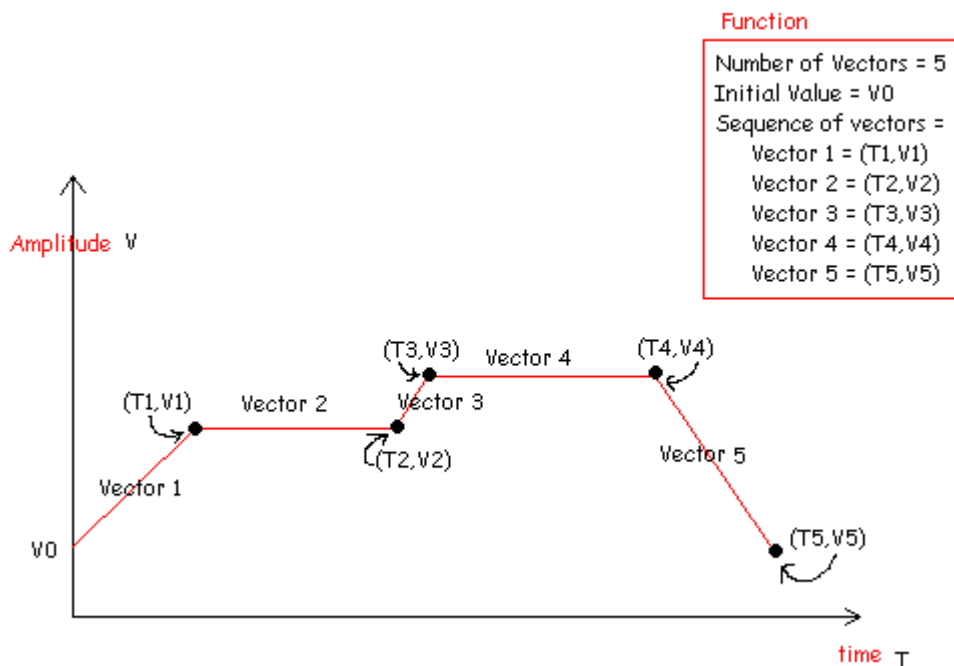


Figure 5.3, Basic Function.

5.2.1.2 Standard Functions: Constant, Cosine and Saw-tooth Function

There are some well known functions that can be defined by some particular parameters easily. These functions are:

- Constant Function (the amplitude is enough to describe it),
- Cosine Function (the amplitude and the frequency are enough to describe it),
- Saw-Tooth Function (the amplitude and the frequency are enough to describe it).

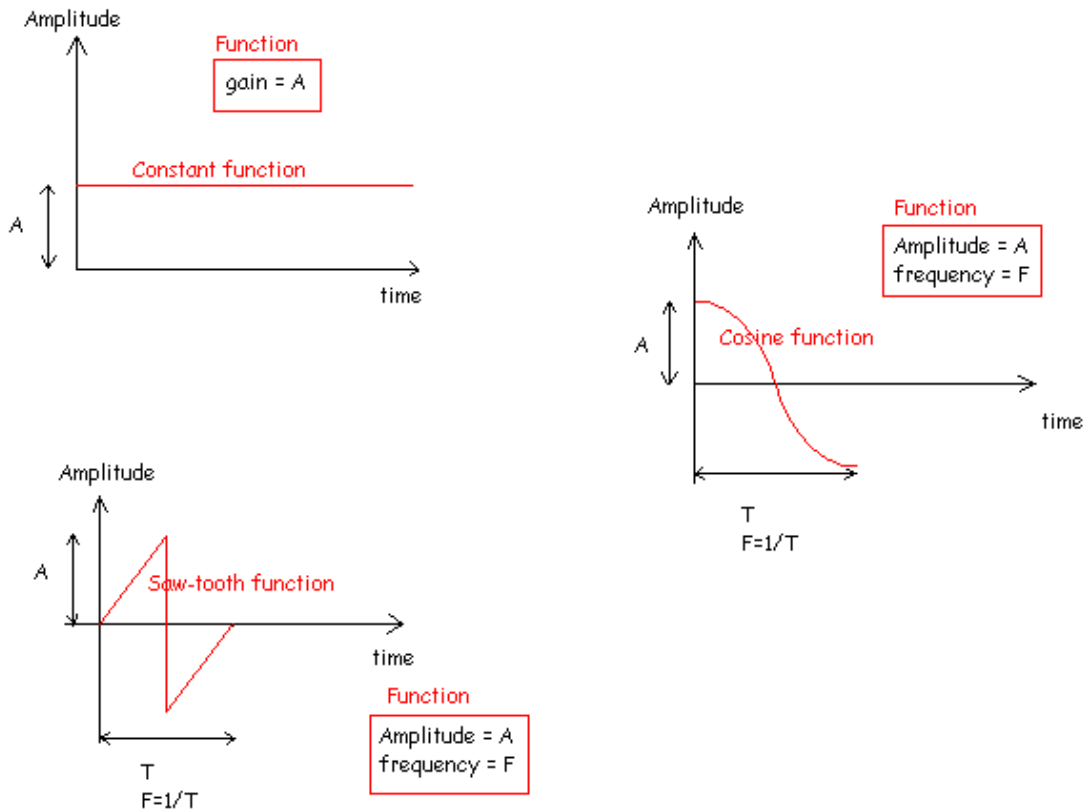


Figure 5.4, Standard Functions

5.2.1.3 The hardware representation

The hardware representation is the description of the functions used into the GFAS memory. Both Basic functions and Standard function are stored using this representation. The time interval of every vector is expanded as a sequence of sub-intervals called steps. Each step is made of one or more basic steps of $5\mu\text{s}$.

Each vector is seen as one or more triples, where a triple is made of:

- Number of step N_i
- Duration of step (as a multiple of $5\mu\text{s}$ basic step) S_i
- The difference in amplitude from the previous vector, $V_i - V_{i-1}$

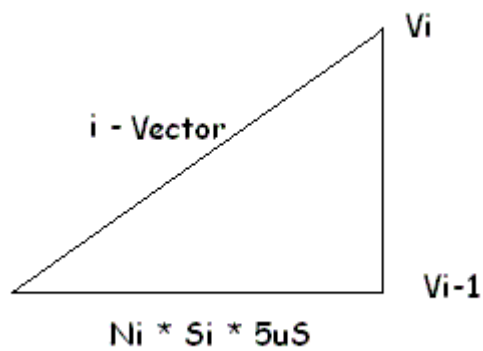


Figure 5.5, Hardware representation of a vector.

The following picture shows a microscopic view of the timing of the analogue output voltage of a DAC generating a trapezoidal function with 3 vectors.

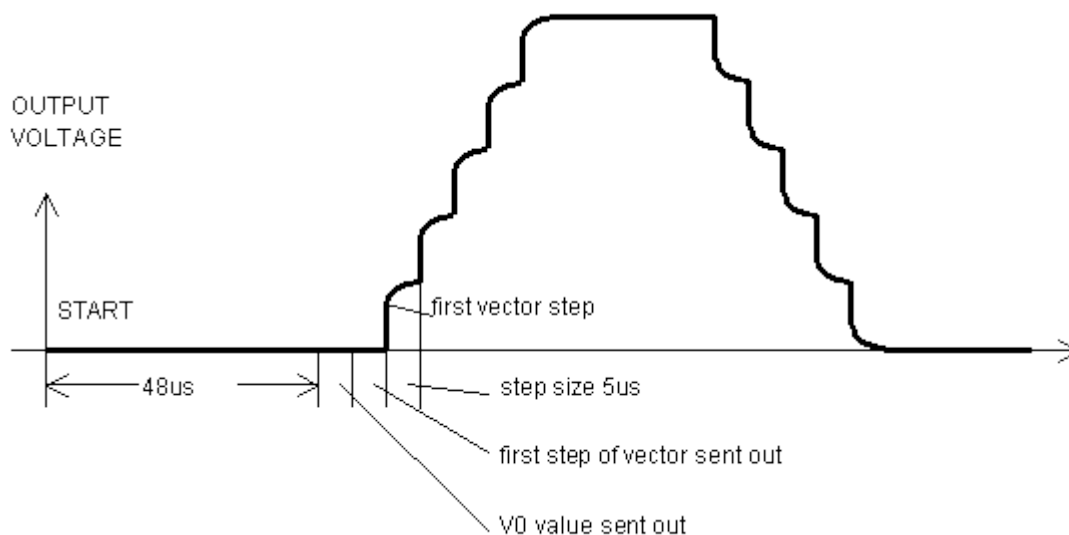


Figure 5.6, Microscopic view of a function. .(from PS/CO/Note 93-108)

Sometimes a triple is not enough to describe a vector because of a representation problem. In fact in the GFAS memory we have 16 bit to represent the value of $V_i - V_{i-1}$, if this value is larger than the maximum value then we have to split the vector to a set of sub-vectors and associate a triple to each sub-vector in the same way as we do with common vectors. Let's see an example in the following figure. The difference between V_i and V_{i-1} is too big then we use the sub-vector 1 with the difference between V_i' and V_{i-1} and the sub-vector 2 with the difference between V_i and V_i' .

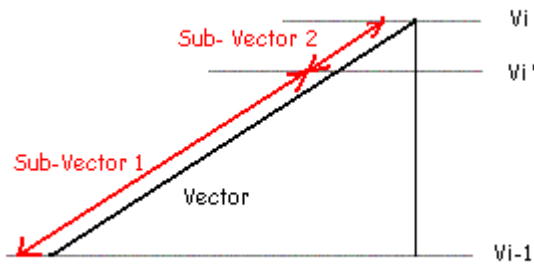


Figure 5.7, Vector and sub-vector.

Then to define a function according with the hardware representation the following items are enough:

- number of triples in the hardware representation
- number of vectors in the physical representation
- Origin value of the function
- The set of triples.

The internal stop vector is represented by the triple (1,-1, next vector). We will talk about it later in par. 5.2.4.

5.2.1.4 The internal representation

There is also another representation of a function, it is the one loaded into the Internal GFAS memory and it is completely transparent to the users.

5.2.1.5 Virtual functions

The possibility of working with a virtual GFAS is a concept added later in the GFAS environment. Up to 5 virtual GFAS can be associated to one physical GFAS.

A virtual GFAS does not have its own hardware but it uses the hardware of the physical GFAS it is associated with. The function, which is loaded into the physical GFAS, is the one made by adding all enabled virtual functions of the associated virtual GFAS' for the corresponding function index. Let us see an example illustrated by the following figure. We have two virtual functions, the first is made by a vector and the second by four. As we see, the composite function is obtained by an adding operation between the virtual functions and the result consists of 5 vectors. The end of the 5th vector is the end of the function corresponding to a STOP, and therefore at generation the output value falls back to its initial value.

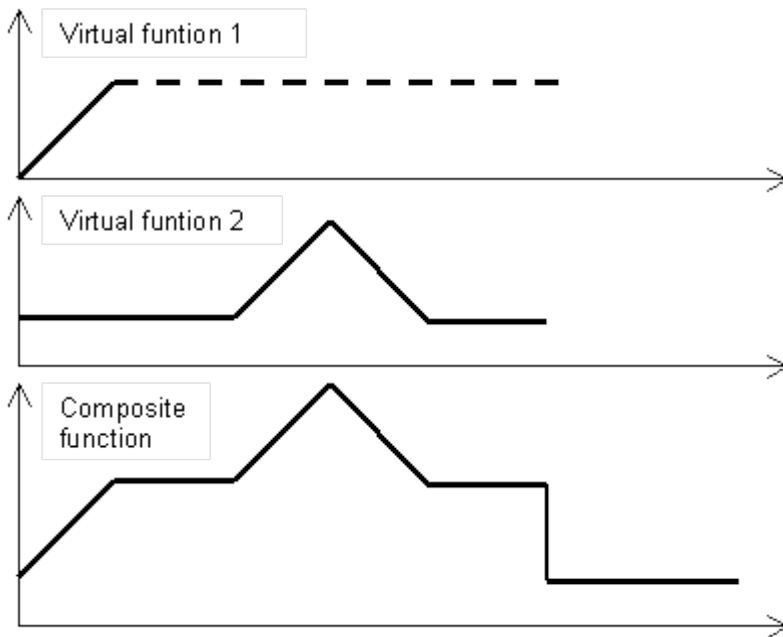


Figure 5.8, virtual GFAS (from PS/CO/Note 93-108) .

The number of vectors of the composite function can be as large as the sum of the vectors of all virtual functions (510 vectors).

If n vectors have the same time points, the total number of vectors is reduced by $n-1$.

One has to take care that the total number of vectors does not exceed 510 and that the amplitude stays within the maximum values. The physical and all associated virtual GFAS have the same scaling factor and the same type of connection. (We will talk about this parameter later).

The composite function is calculated and loaded into the physical GFAS every time a virtual function is changed, enabled or disabled. If all virtual functions are disabled, a function with a zero vector is loaded into the physical GFAS.

The physical GFAS behaves in all aspects like a normal GFAS, i.e. its function can be disabled, enabled or loaded. Normally, but not necessarily, its manipulation via the GFAS editor is inhibited. (We will see it later). A virtual GFAS cannot access the hardware, all operations that try to do that are considered not valid and they raise an exception.

5.2.2 Function management

In the GFAS memory we have a table to load the hardware representation of up to 32 functions. Each function is distinguished by a function-index, an integer between 0 and 31, corresponding to its position in memory. A real time activity chooses one function from this pool according to the accelerator needs. The information brought by the corresponding telegram is enough to select one of these functions.

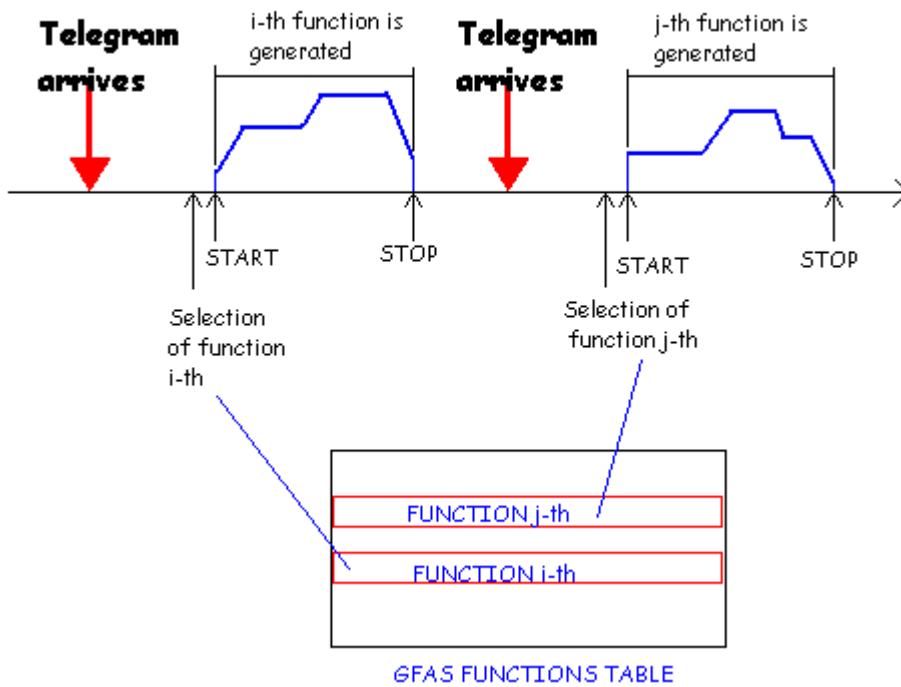


Figure 5. 9, Relationship between telegrams and function generation.

5.2.3 Possible states of a function

Now let's describe all the states in which a function can be.

A function can be:

- *Absent*, the number of vectors is 0.
- *Loaded*, we have one or more vectors loaded to define a function in the GFAS memory.
- *To be converted*, the function is loaded and it's waiting to be converted according to the internal function representation.
- *Converted*, it's converted according to the internal function representation and loaded in the Internal GFAS memory.
- *Accepted*, the conversion into the internal representation is correct and the function has been accepted into the Internal GFAS memory without errors.
- *Selected*, it's the function chosen to be generated.
- *Enabled*, the user decided that it is possible to generate it.
- *Disabled*, the user decided that it is not possible to generate it.
- *To be generated*, the selected function is the one generated as output of GFAS.

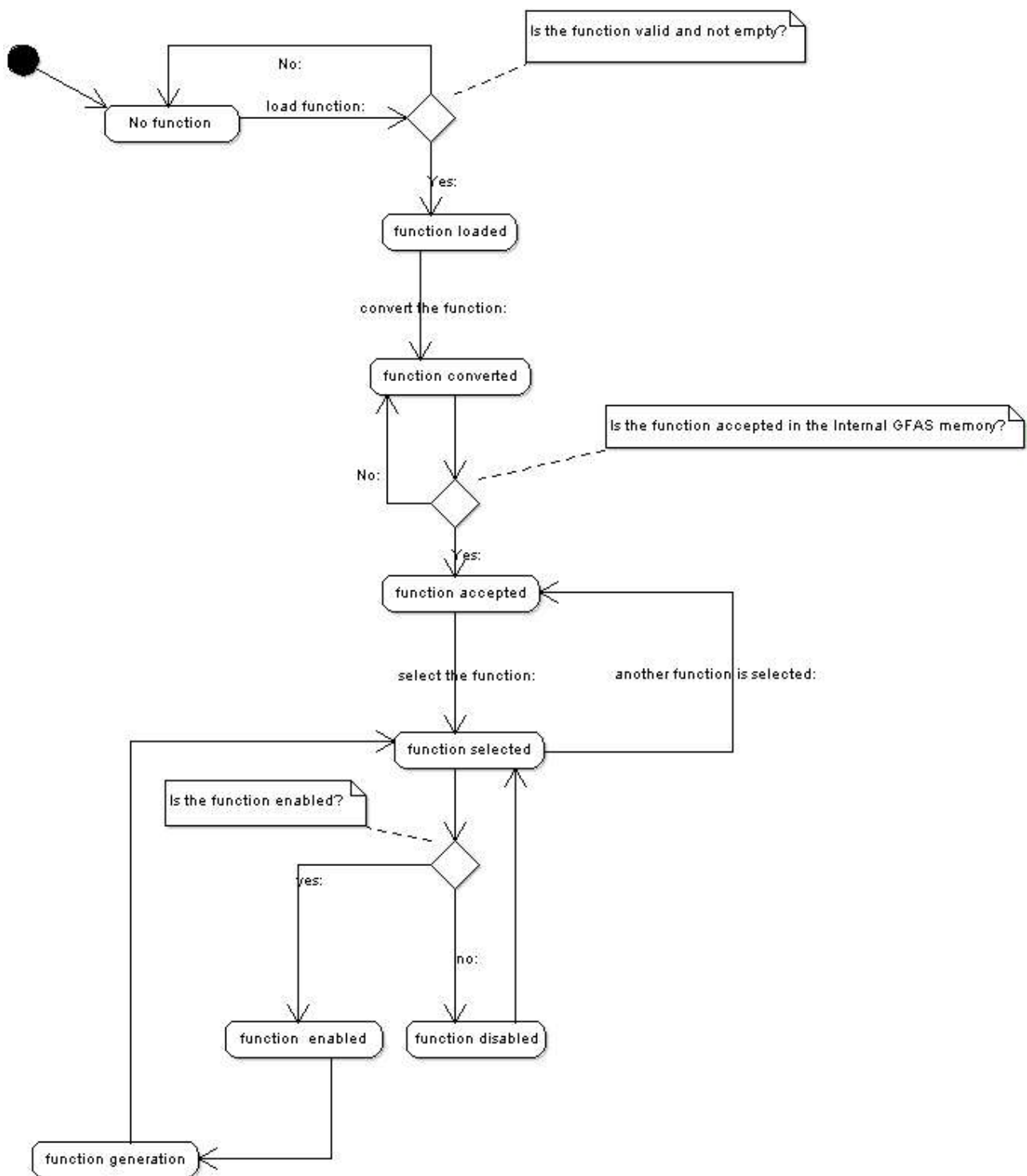


Figure 5.10, function state diagram

5.2.4 Function generation

Let's now analyse what could happen during the function generation and the influences of different events that can happen.

	Main start events	Main stop events	Start events	Stop events
Hardware events	external	external	external	external
Software events	external	internal	external	internal

The main start is the event that occurs when the generation of a function starts from its first vector and the main stop it is when the function finishes its generation. The simple start event (or start event) makes the function generation start from the last vector used and the simple stop event is used to pause the function until the next simple start event.

The events could be generated by hardware or software. For software events we have external ones (we use a function to trigger them) or internal ones (the event is coded in the representation of the function). The main stop events and the stop events are coded in the same way (see par. 5.2.1) but the main stop events can be coded only in the last vector of the function.

Let's use a picture to explain better the generation of a function.

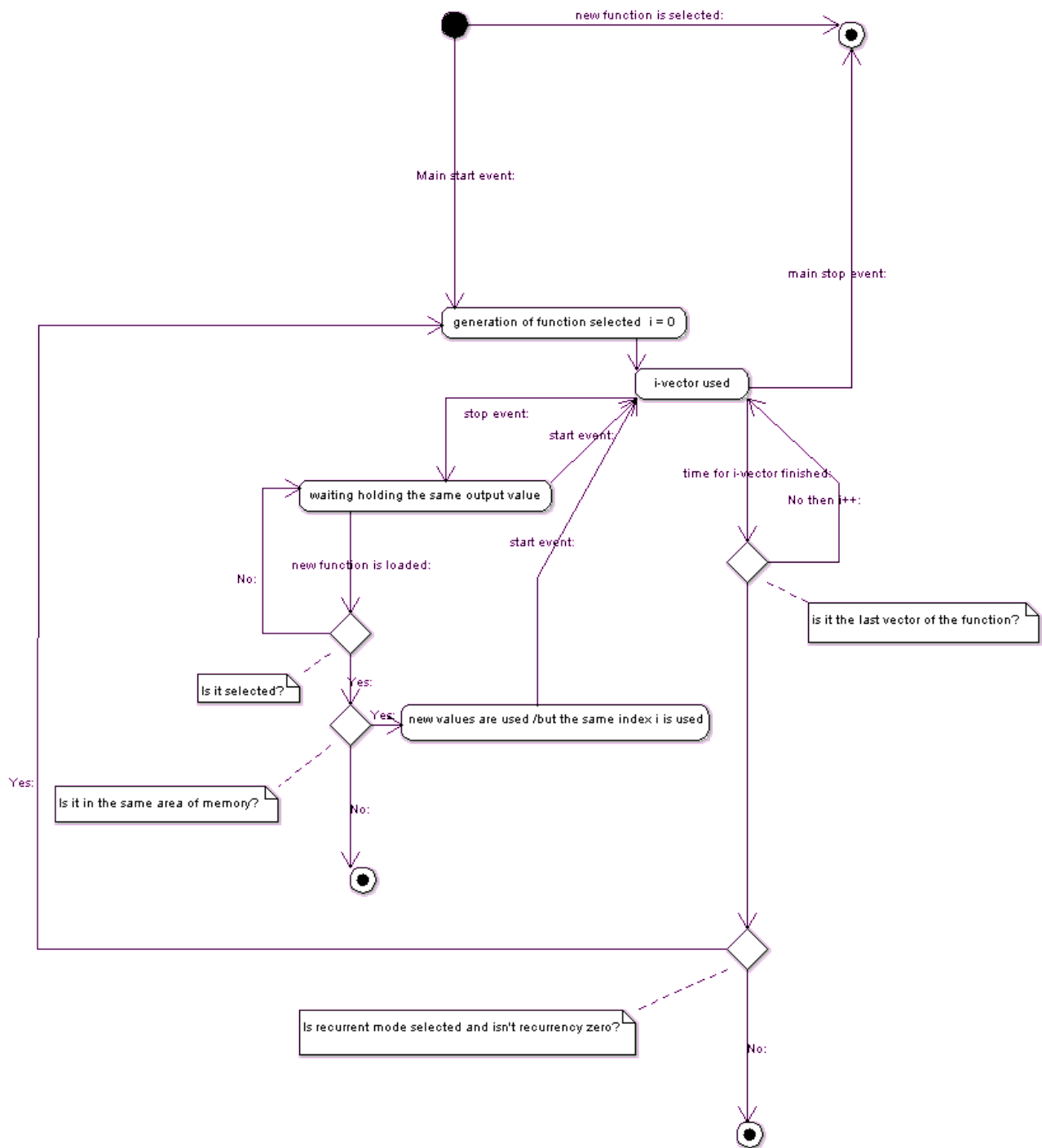


Figure 5.11, Function generation state diagram.

We have a few possible situations. Let us see some examples showing how the analogue output changes (we use a DAC to convert the GFAS digital output to the analogue one).

If during a function generation, an Event Stop is given to the GFAS, function generation is suspended and reactivated at the next Event Start, see figure 5.12. Software Event Start and Hardware Event Start behave exactly in the same way. At the Event Stop, the DAC keeps its actual value until the Event Start at which moment generation of the interrupted vector is resumed. No vector is sent to the DAC. The first vector is sent to the DAC ca. 32µs after the Event Start. [11]

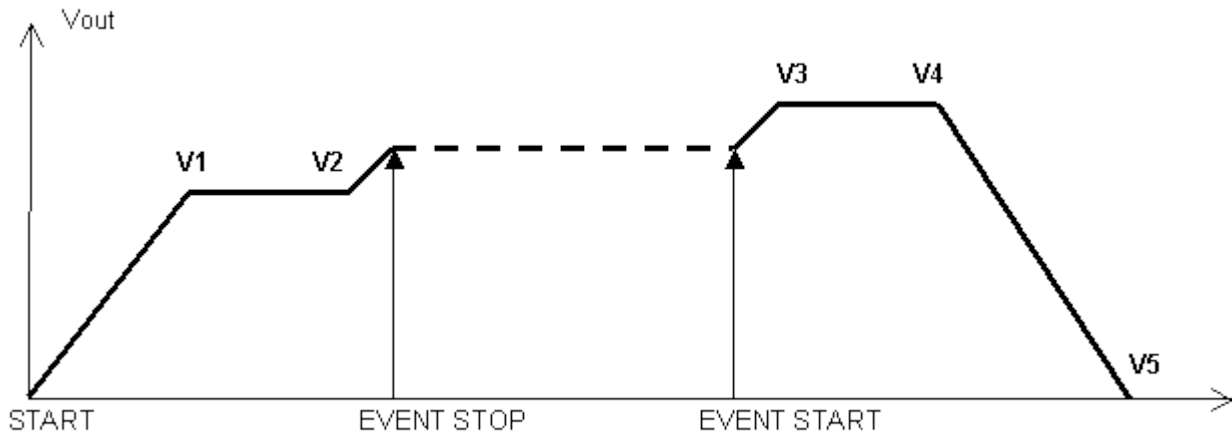


Figure 5.12, Event start and event stop. (From PS/CO/Note 93-108)

After an Event Stop, another function can be loaded into the same function area. The new function can be completely different from the previous one and also the number of vectors can be different. Figure 5.13 shows an example, F_1 is the original function, F_2 is the new function loaded after an Event Stop. Loading means in the following also converting (i.e. loading into the Internal GFAS memory). [11]

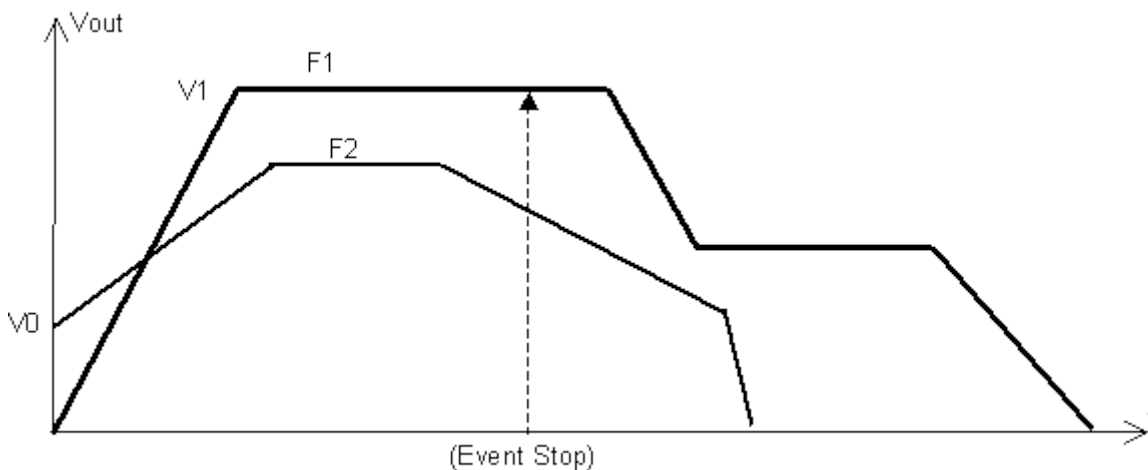


Figure 5.13, Event start and event stop with a new function loading in the meanwhile (from PS/CO/Note 93-108).

Loading of the new function with the same function number, but not selecting it, does not change the DAC output. After an Event Start, the vector during which the function has been stopped before is continued, and only then the new function is taken into account, see figure 5.14. [11]

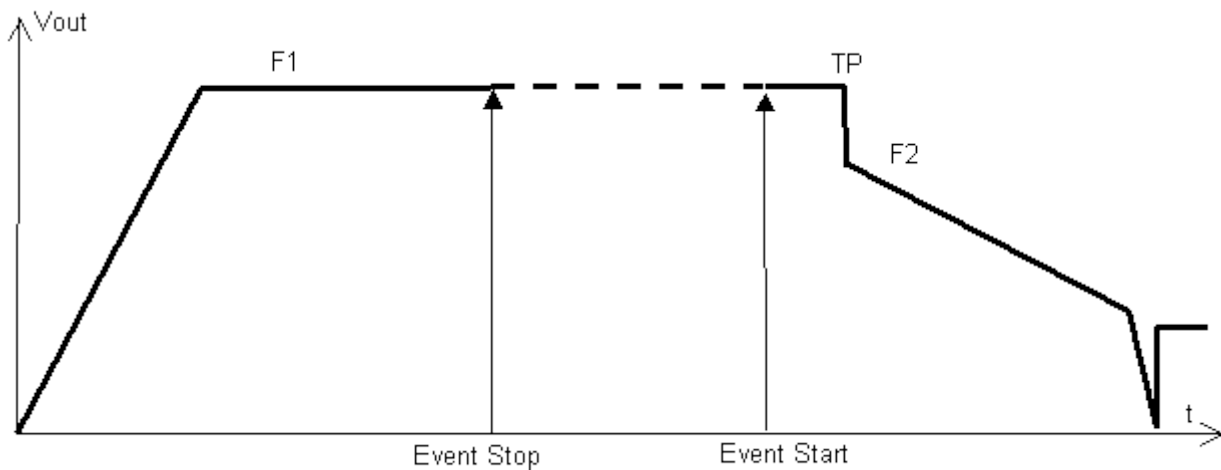


Figure 5.14, Function after Event stop and event start (from PS/CO/Note 93-108).

After the 2nd vector of the first function is finished (at the Transition Point TP), the start of the 3rd vector of the second function is reached within one step. The value of the slope parameter has no influence on that slope. Note that after the stop the output jumps to the initial value of the secondly loaded function. [11]

Instead of an Event Stop, an Internal Event Stop could have been built into F_1 . Suppose the 2nd vector had been programmed as $(0, V_1)$, then the function generated would look very similar to the one in figure 5.16 except that F_2 is immediately entered at vector 3 when the Event Start arrives, see figure 5.16 (vector 2 is considered as already worked through by the Internal Stop). Figures 5.14 and 5.15 show the difference between these two Event Stops. [11]

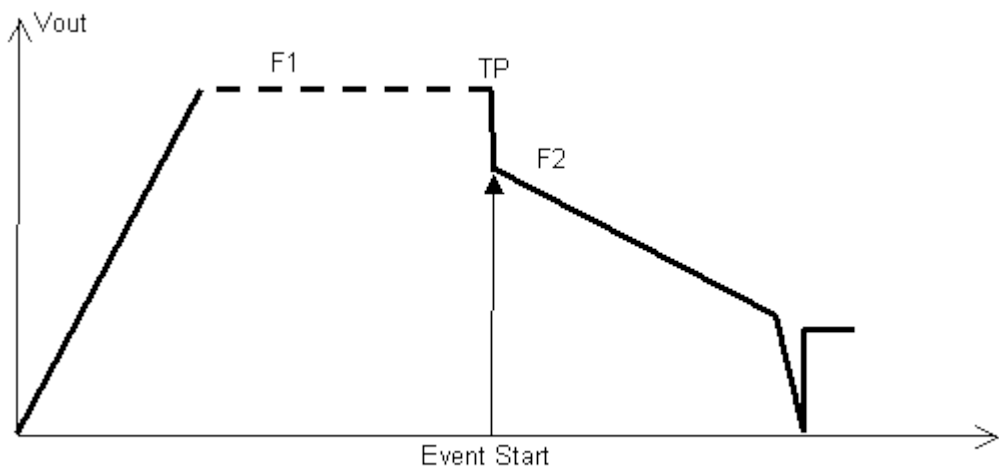


Figure 5.15, Function with internal event stop and event start (from PS/CO/Note 93-108).

The picture changes, if the second function is not only loaded but also selected. Selecting the second function brings the DAC output with a slope to the value of the vector of the second function whose vector number is the same as the one containing the Event Stop, see figure 5.16. [11]

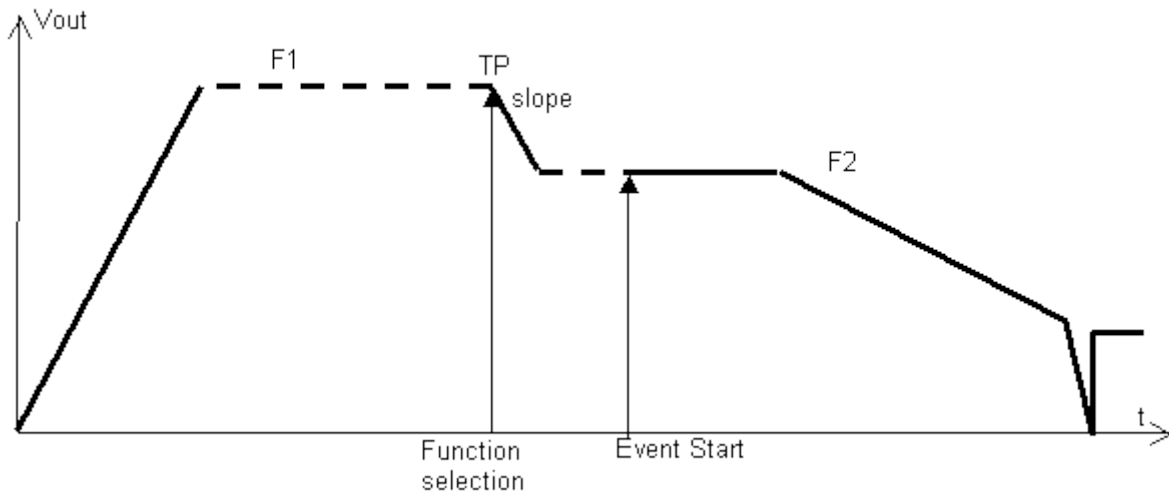


Figure 5.16, Function with internal event stop and event start, F2 is selected, (from PS/CO/Note 93-108).

The slope depends on the slope parameter (sl), that can be written and read. We will talk about all GFAS parameters later. The slope parameter is always positive, the sign is chosen automatically according to the function. For the function shown in figure 5.16 we have the following rise time and fall time (for an operational range of [10V,-10V], ΔF in Volt and Δt in ms):

$$t(\text{rise}) = (42.5 + 6 \cdot \text{sl}) \cdot \Delta F$$

$$t(\text{fall}) = (33.7 + 6 \cdot \text{sl}) \cdot \Delta F.$$

The generated function changes again if, after the internal stop, the new function F₂ is loaded under a different function number and selected. As before, an Internal (or External) Event Stop leaves the output at the reached amplitude V₁. Loading F₂ into a different function area does not change the output. Only if this new function is selected, its initial value V₀ is reached with a slope, i.e. the selection is the trigger. The next Event Start then generates the function from its beginning on, see figure 5.17. The slope is calculated as before. [11]

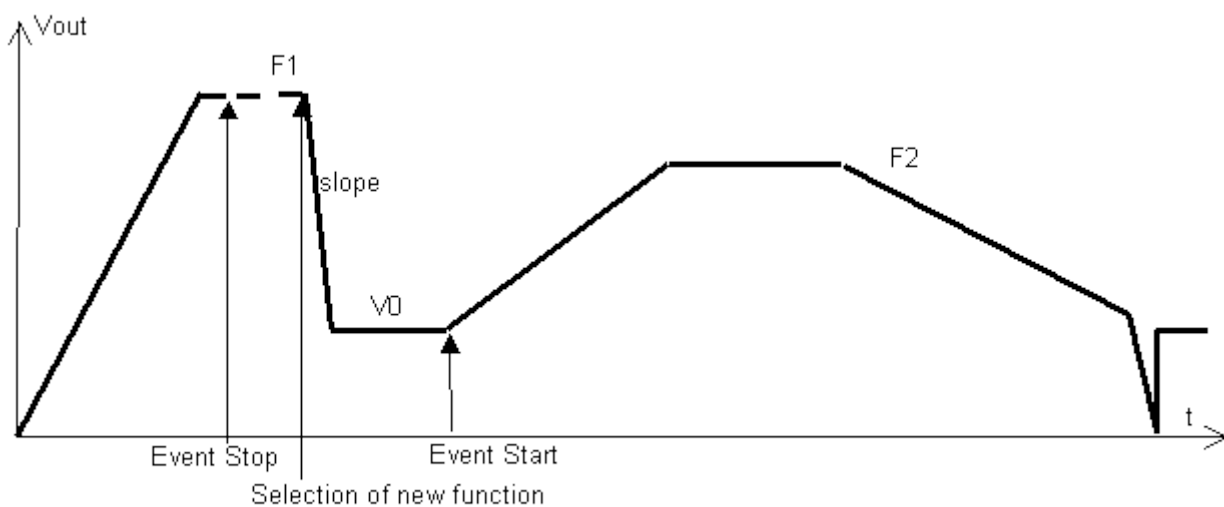


Figure 5.17, New function loaded under a different function number (from PS/CO/Note 93-108).

As we have already said in par.5.2.1, a vector can be defined to contain an Internal Event Stop. The triple characterizing a normal vector (N_i , S_i , V_i) is then replaced by a triple characterizing an Internal Event Stop which is N_i (dummy), -1, V_7 (start value of next vector. Number of steps N_i is not important, however, it must not be 0, one normally chooses 1.[11]

After reaching an Internal Event Stop, the function generation is suspended as for an External Event Stop. The function generation is resumed after an Event Start. In figure 5.18, the function is made of 3 pieces: f_1 , f_2 and f_3 is shown, where f_1 and f_2 are terminated by an Internal Event Stop. The start value of the next piece is reached in one step after the Event Start. The internal stop vectors are V_3 and V_7 . [11]

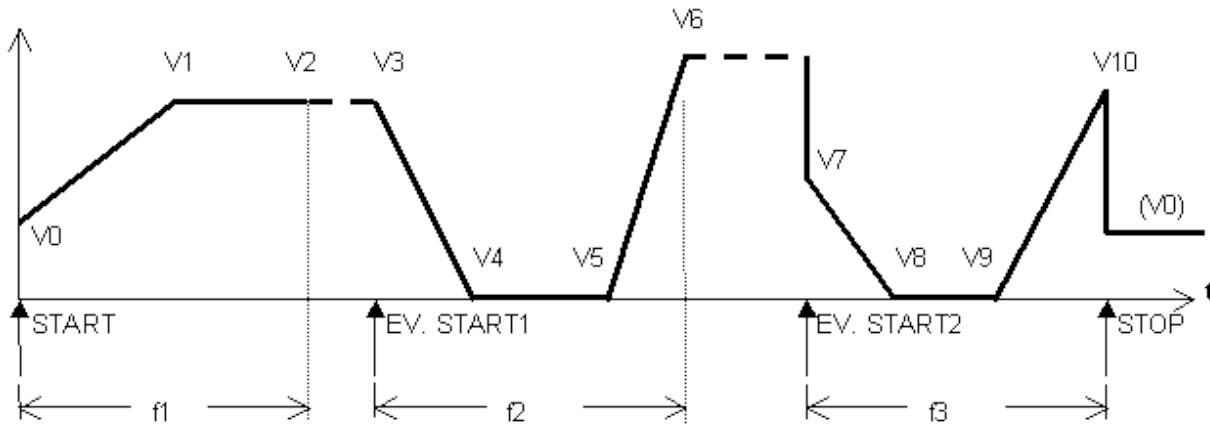


Figure 5.18, Function with two stop event (from PS/CO/Note 93-108).

Reloading a function after an internal stop is also permitted, the old function is overwritten and the generation is resumed at the same vector number as it would be if the old function would not have been replaced. If the new function contains the same number of vectors, with the same internal stops, loading and selecting the new function brings the output to the equivalent internal stop. This facility is used to adjust the amplitudes of the flat tops after an internal stop and the value of the new flat top is reached with a slope as explained in the previous chapter. [11]

Figure 5.19 shows the exact timing for the start of the f_3 part of the function.

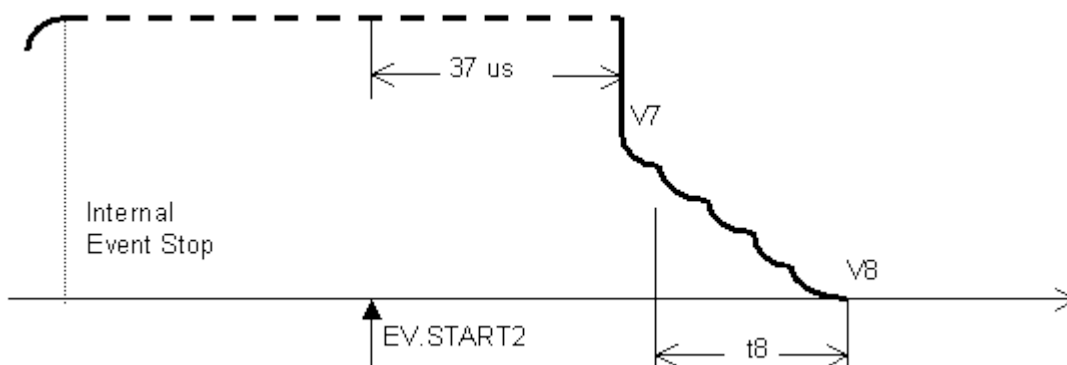


Figure 5.19, Microscopic view of stop event (from PS/CO/Note 93-108) .

The Internal Event Stop vector V_7 (after the vector V_6) is given by the triple:

- 1 (dummy),
- -1 (internal stop),
- V_7 (start value for next vector, i. e. vector number).

The Internal Event Stop vector containing the start value of the next piece of function occupies exactly the space of one vector. So, the vector count is similar to that of a normal function. The start value of the next vector is generated by the DAC ca. $37\mu\text{s}$ after the Event Start (the first data word is sent after $32\mu\text{s}$, this value is converted after one step of $5\mu\text{s}$). These values go down to $22\mu\text{s}$ and $17\mu\text{s}$ respectively if the new function is loaded into another function area.[11]

Concerning other important timing values of the function of figure 5.18 we have also:

- time of the generation of V_0 $53\mu\text{s}$ after the first start (value sent out after $48\mu\text{s}$),
- time of External Stop, it takes $32\mu\text{s}$ to bring the function to V_0 (value sent out after $27\mu\text{s}$),
- time of Internal Stop (after ending of the last vector), it takes in $26\mu\text{s}$ to bring the function to V_0 .

Start and Event Start pulse should not arrive at the same time. If this happens, normally Event Start is recognized.

5.2.5 Recurrent Mode

In recurrent mode, the set of vectors that defines a function is repeated more than one time. The number of times is given as a parameter (reccurrency) and if it is negative it means infinite times. The example in figure 5.20 shows a burst of 3 functions, i.e. recurrent cycles = 2 (function repeated 2 times). In recurrent mode, the CPU needs ca. $38\mu\text{s}$ to send out the V_0 value. The CPU needs one step size ($5\mu\text{s}$) to start function repetition, i.e. the V_0 value of the next function repetition is sent out $5\mu\text{s}$ after the end of the previous function and then the time of the function generation is not long as three function basic period but longer because every time to start the repetition of the basic period there is a delay of $5\mu\text{s}$. In our example it is $10\mu\text{s}$ longer because we have two repetitions. The jitter of the start of the burst is the same as in single function generation.

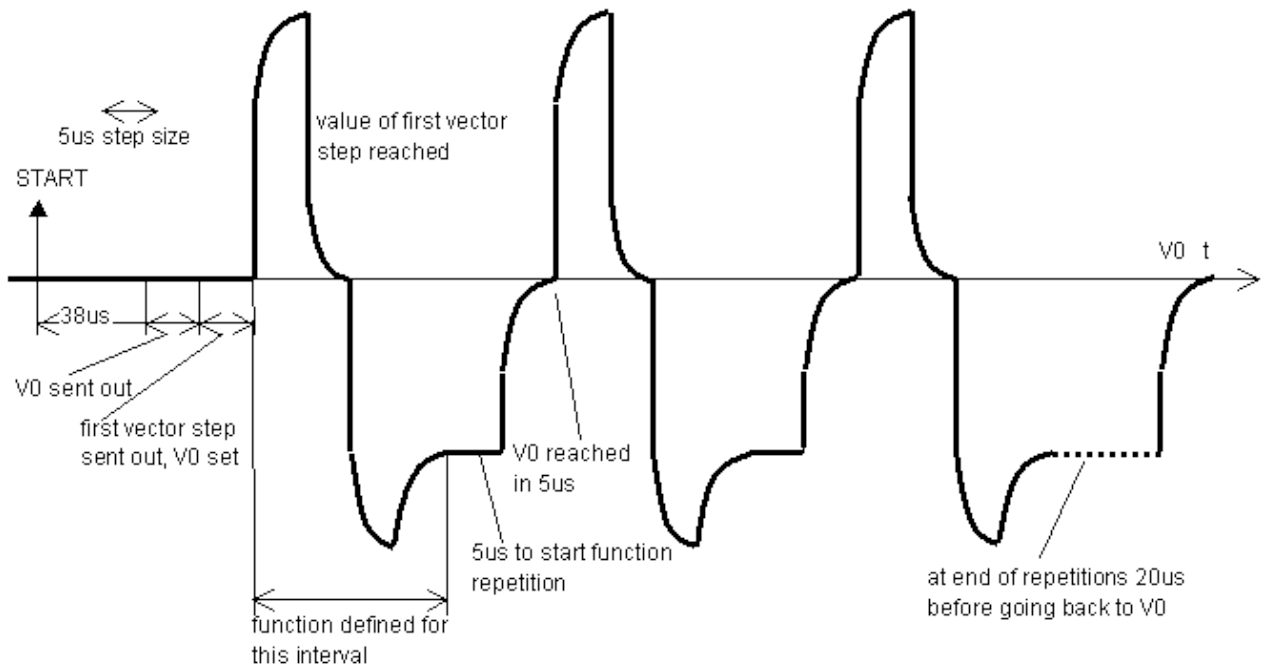


Figure 5.20, Recurrent mode (from PS/CO/Note 93-108).

5.2.6 Declaration of the GFAS

As all the hardware devices, as soon as a GFAS board is plugged into a VME crate, it is declared into a database with all the information needed to define it.

That information depends on the particular VME crate and also on the equipment device that will use the GFAS output.

For both channels of each GFAS board we have an instance of the GFAS software module and to define a particular instance of a GFAS the following information is needed:

- the *address* of the GFAS channel specified by the triple:
 - Type (type of address, the one used by GFAS which is *IocGFAS*),
 - Lun (number of the slot on the VME crate in which the board is plugged) ,
 - Channel (number 0 or number 1).
- the *connection type* : how the output is used by the equipment which is going to use the GFAS.
- the *unit* of the physical function produced, we can have Volt for voltage, Hertz for frequency, Ampere for current...
- *maximum* and *minimum* value of the range of admitted physical values.
- the *scaling factor* , defined as output of equipment in physical unit per input where the input can be according the digital representation or the physical one (e.g. in Volt).
- name of a *master* device if we are working with double or triple multiplexing (see par 6.2) and a *multiplexing extra condition* (0 if we don't use this feature).

– name of the *sine slave* (0 if we don't use this feature), if we are going to work with a cosine function on a channel and we want (at the same time) to control a sine function on the other channel. The only difference between the two functions produced is a difference on 90 degrees in the phase. We will see it later.

– the *virtual number* if we use virtual functions.

Technically, a virtual GFAS is recognised by having the element number of its associated physical GFAS as virtual number.

Physical GFAS has -1 as virtual number and GFAS's, which are not using this feature, have 0 .

– the *shaping code*, which is a code to say if it is possible manipulate the delay or/and the maximum absolute value of the function (Ref value). (See figure 21)

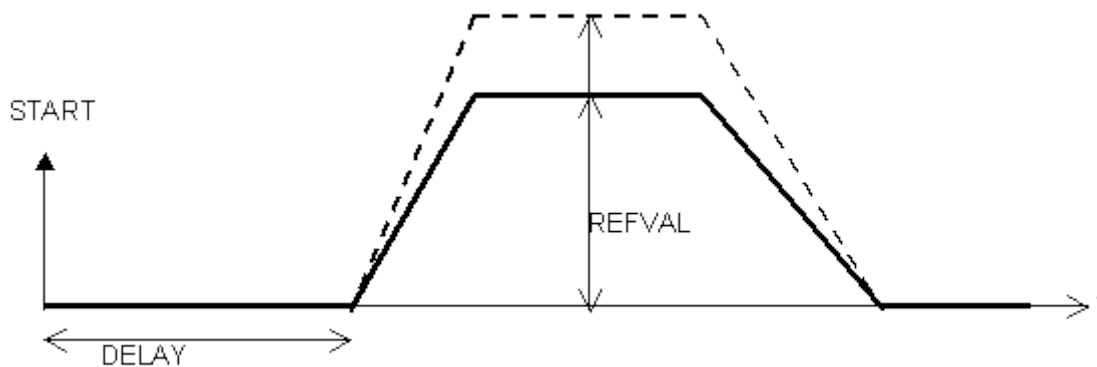


Figure 5.21, Delay and ref value (from PS/CO/Note 93-108) .

Chapter 6

GFAS USERS

Now let us look at the GFAS equipment module from the user's point of view.

We can distinguish three different types of user in regards to the server activity and one type of user for real time activities:

➤ Server users

- common user,
- expert user,
- guru user.

➤ Real time user.

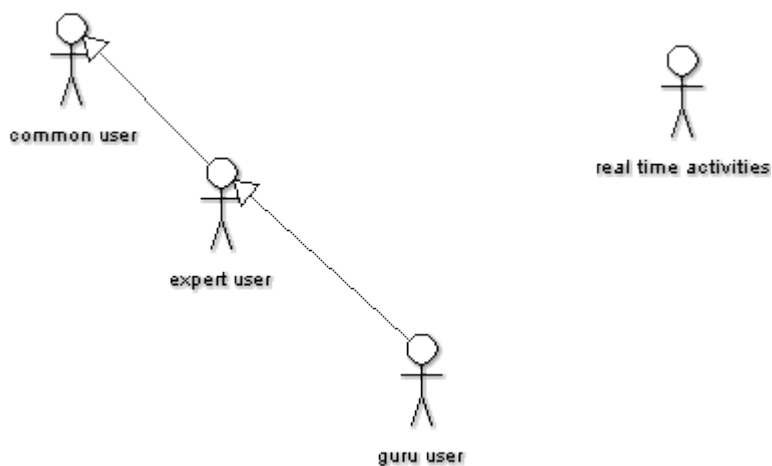


Figure 6.1, GFAS users.

6.1 Server users

First of all let us analyse the *Server users*:

The **common user** works with the physical representation of the function.

He may:

- specify the Mode in which the GFAS has to work (or he can use the default Mode),
- load, upgrade or delete the functions using the physical representation or specifying particular parameters such as frequency or gain, if he is working with constant, saw-tooth or cosine function,
- enable or disable the functions loaded,
- send hardware main start events, hardware start events, hardware main stop events and hardware stop events,
- read the physical value of the DAC value,
- read the date when the last function has been loaded,
- specify the value of recurrence if the recurrent mode is selected,
- specify the slope value that is the slope trough which the origin value of a newly loaded function is reached from the previous one. It's always positive and the correct sign is chosen automatically.

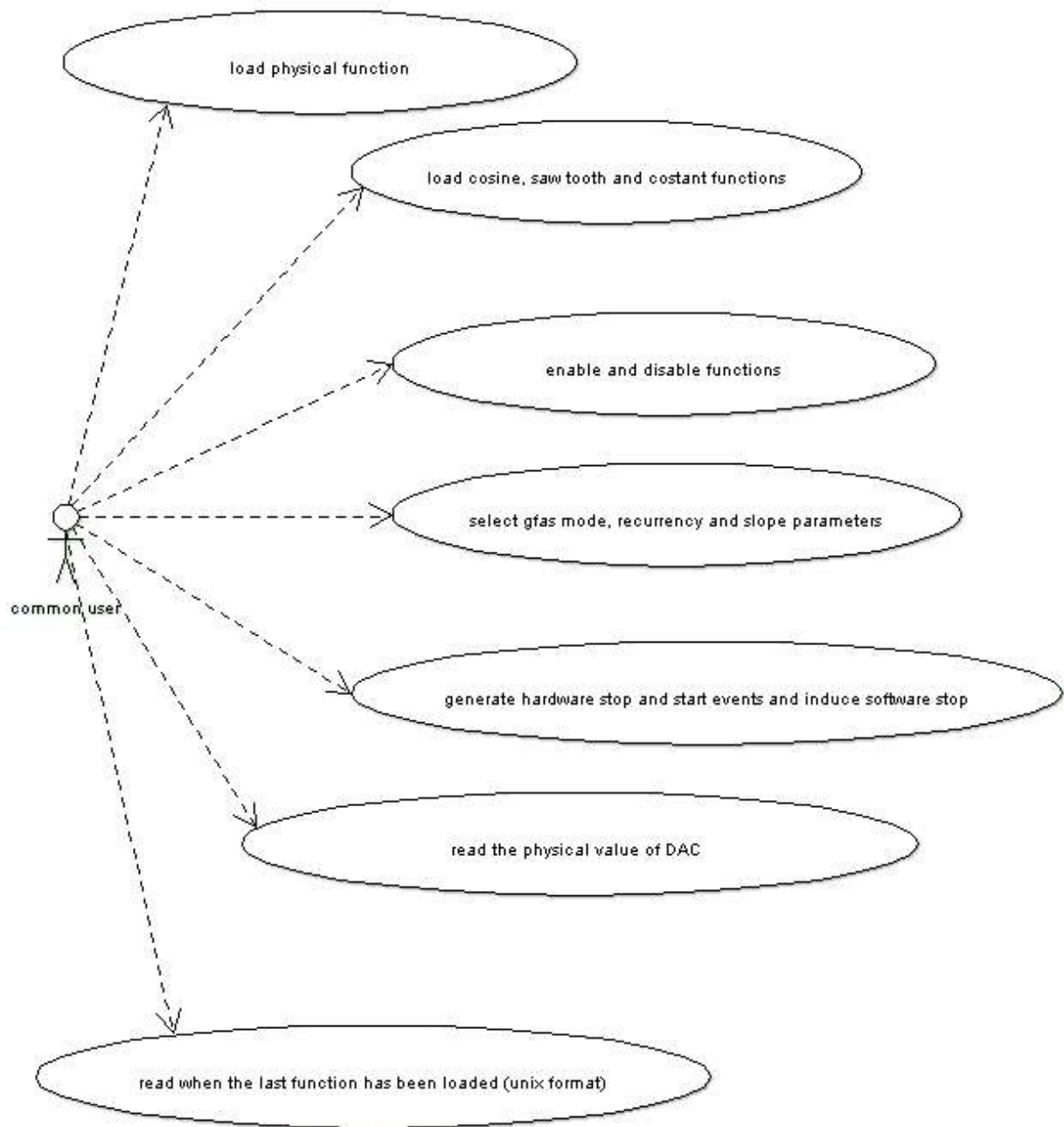


Figure 6.2, Common user's activity.

Concerning the **expert user** we have some more actions possible for him because he is supposed to have a deeper knowledge of the device, he can:

- work with a hardware representation of the function and set some flag to ask for the conversion of the function to the Internal representation by a hardware operation,
- select a function and induce its generation,
- make a copy of a function and set some flag to ask for the conversion of the function to the Internal representation by a hardware operation,
- trigger software main start events, software start events, software main stop events and software stop events,
- reset GFAS memory,

- read the digital representation of the DAC value,
- read the Status word from the GFAS, which is a word that can describe the status of the device.

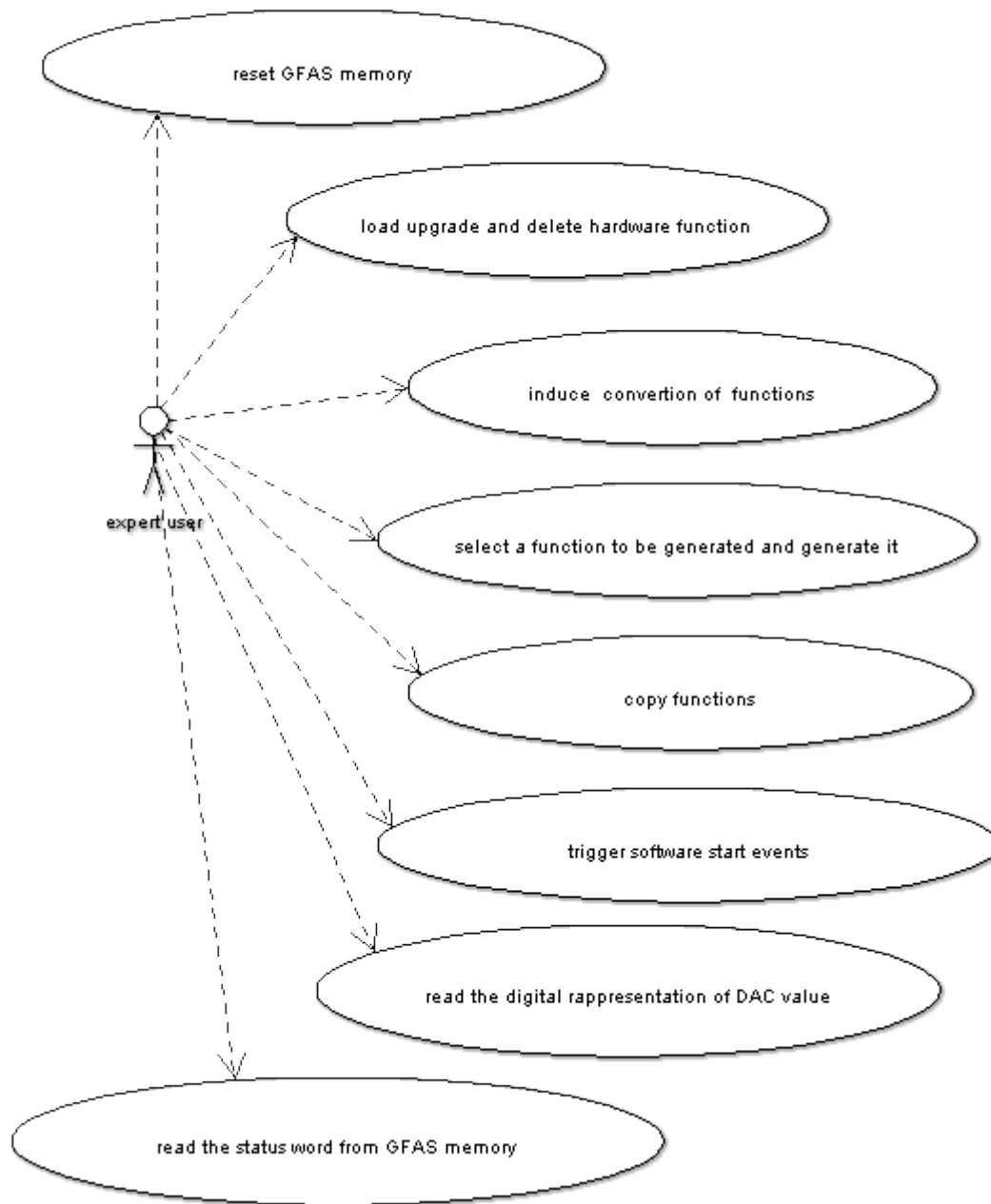


Figure 6.3, Expert user's activity.

Finally we have the **guru user** who has a deeper knowledge of the GFAS memory, and can set all the parameters manually and to control the board going straight to the hardware words. To analyse what these users can do, we need to introduce how GFAS memory is organised, and then we will explain it later in a more detailed way.

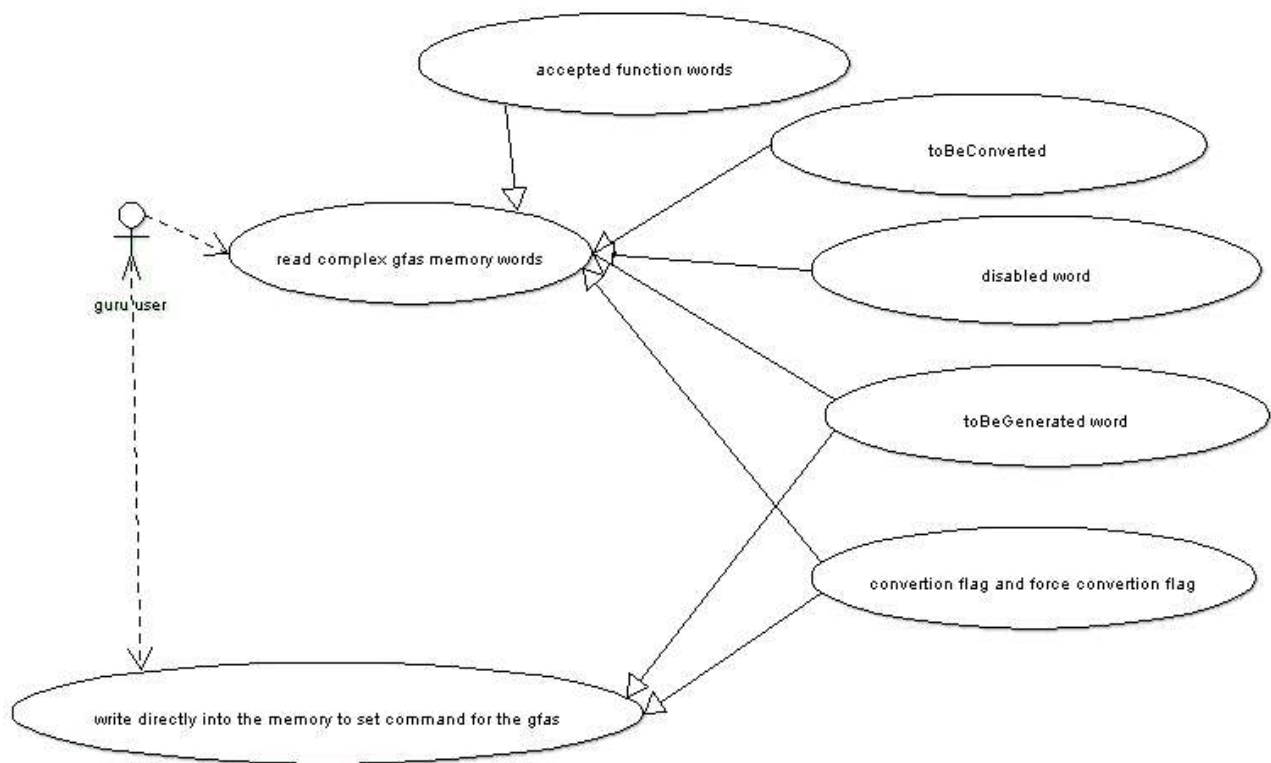


Figure 6.4, Guru user's activity

6.2 Real-time user

Concerning the *real time activity* we have two different operations.

- the acquisition operation :

Getting information about the function generation.

- the controlling operation :

Using information in the telegram to choose a function in the group of functions loaded by the Server users.

Only in the case of double and triple multiplexing there could be the necessity for loading a function in memory and converting it to the internal representation by a hardware operation, before the generation (We will analyse this better in the next paragraph).

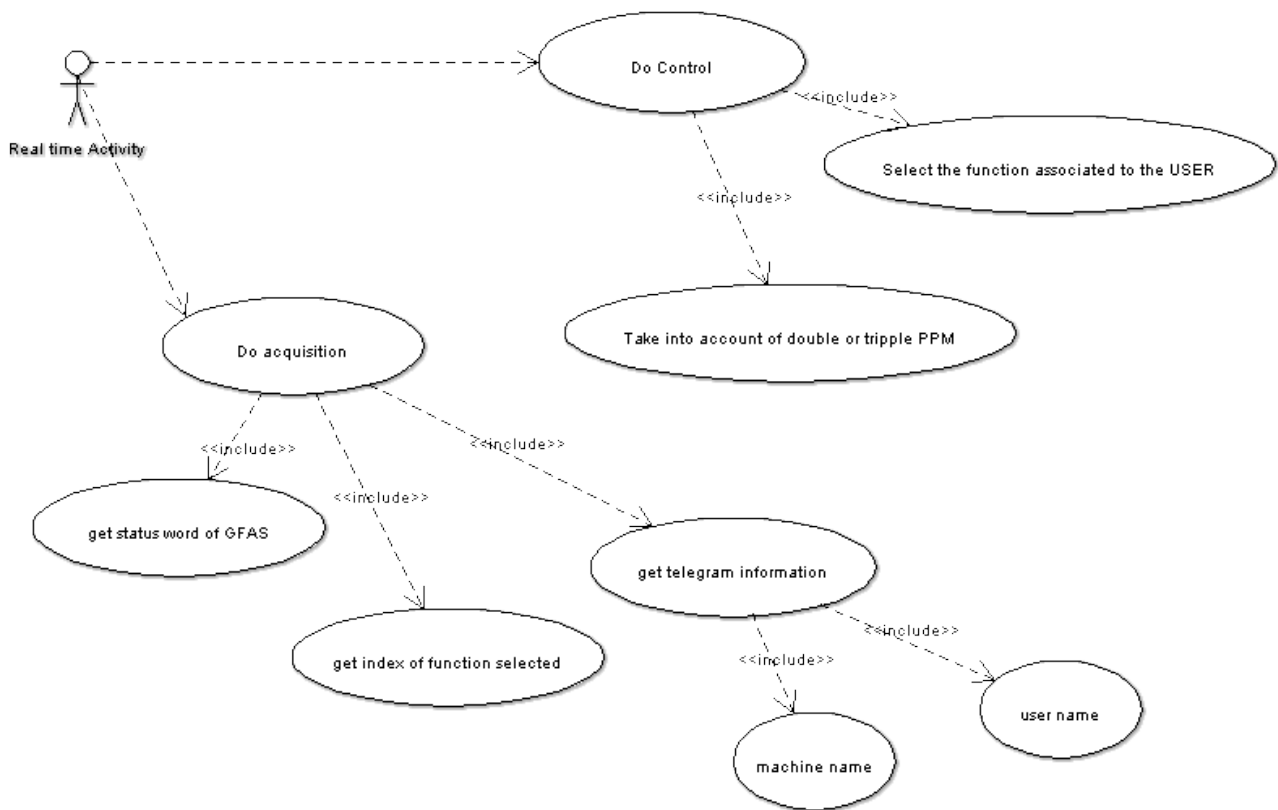


Figure 6.5, Real time user.

6.3 Real time context

We have different real time behaviours depending on the type of context which could be non-multiplexed, multiplexed, double multiplexed or triple multiplexed. As we have seen in par. 4.4, telegrams are used to provide information about the sequencing of real time activity and in each telegram we have a set of values for all the possible groups (USER, DESTINATION...). In a non-multiplexed context the behaviour of the equipment is always the same even if we have different information brought by the telegram. In a multiplexed context we choose one of the possible group (usually USER) and for different values we can see different behaviours. In a double or triple multiplexed context we use more than one of these groups to define different behaviours.

Let us analyse the situation for the GFAS module.

In a non-multiplexed context we have just one function (the function index is always 0) loaded in memory, and it is the only one that can be selected by the real time activity. It is generated if it is not disabled by Server users.

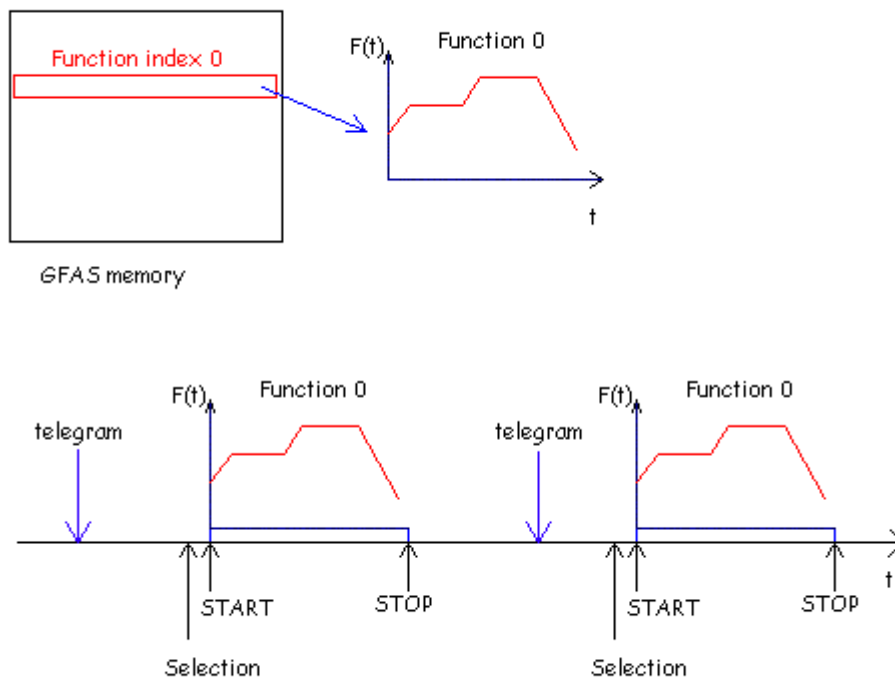


Figure 6.6, *None-Multiplexed context.*

Otherwise, in a multiplexed context, we have a function for each different cycle. All functions are loaded in parallel in the memory and the duty of the real time activity is to select one of them according to the information in the telegram: If the i -th cycle is specified in the telegram, the i -th function is selected. To distinguish the different cycles we use the group USER. Of course it is possible to write, overwrite or read any function at any time.

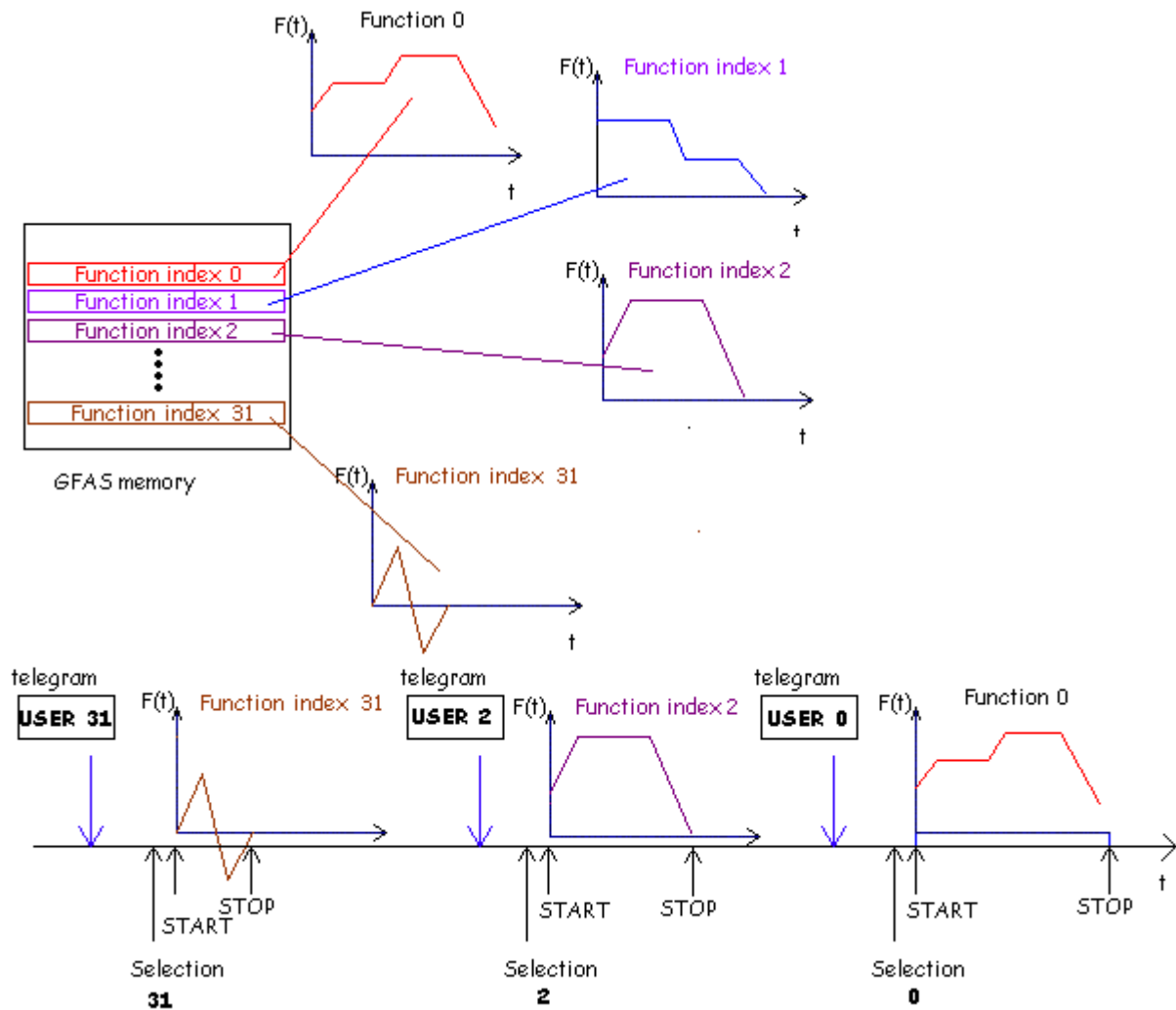


Figure 6.7, *Multiplexed context.*

Things are different if we use the triple or double multiplexed context feature. In fact the function changes not only depending on one of the attributes of the cycle (e.g. USER) but also on an option indicated by another group in the telegram. (E.g. some functions can depend on the DESTINATION and not just on the USER. See the following figures).

For the triple multiplexed context we have up to 3 different sets of functions per GFAS. This aim is to distinguish between the GFAS master and the two possible GFAS slaves associated with it (this concept is NOT related to the slave channel already seen).

The behaviour of the GFAS master is the same as that of a device that does not use the double or triple multiplexed-context. When we work with a GFAS slave, an extra condition is checked. If it is verified, the function of the slave is loaded into real memory of the master, is converted according to the internal representation, and selected to be generated. The extra condition to be verified contains the name of the machine, the group that we want to consider, and two possible values that it can assume. In the master GFAS we have a default set of functions (one for each cycle) and other two sets loaded in the two GFAS Slaves. In the double multiplexed context we have the same situation except that we have one slave instead of two.

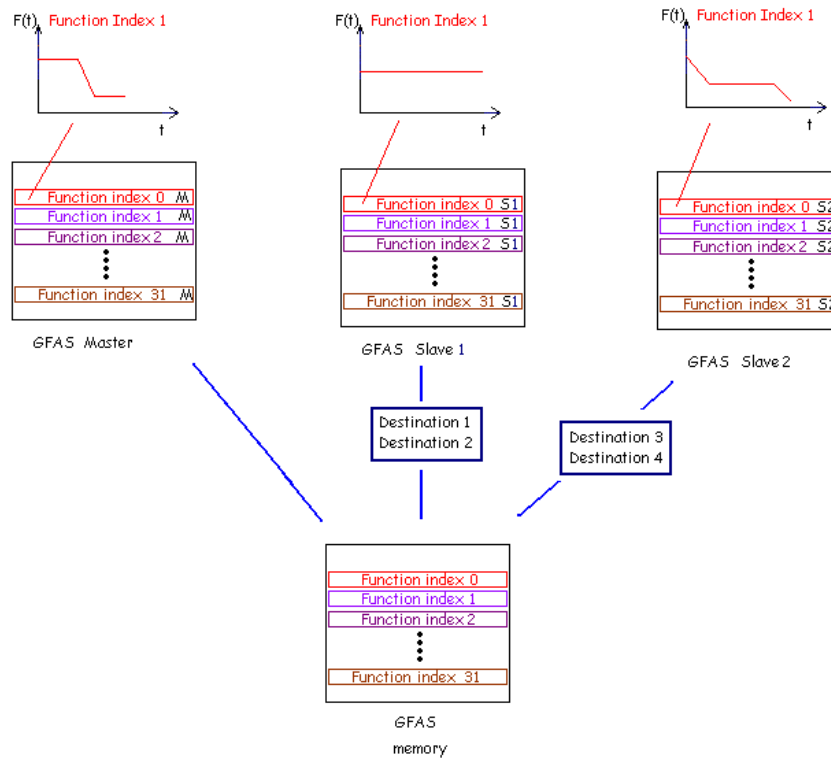


Figure 6.8, a Triple Multiplexed context.

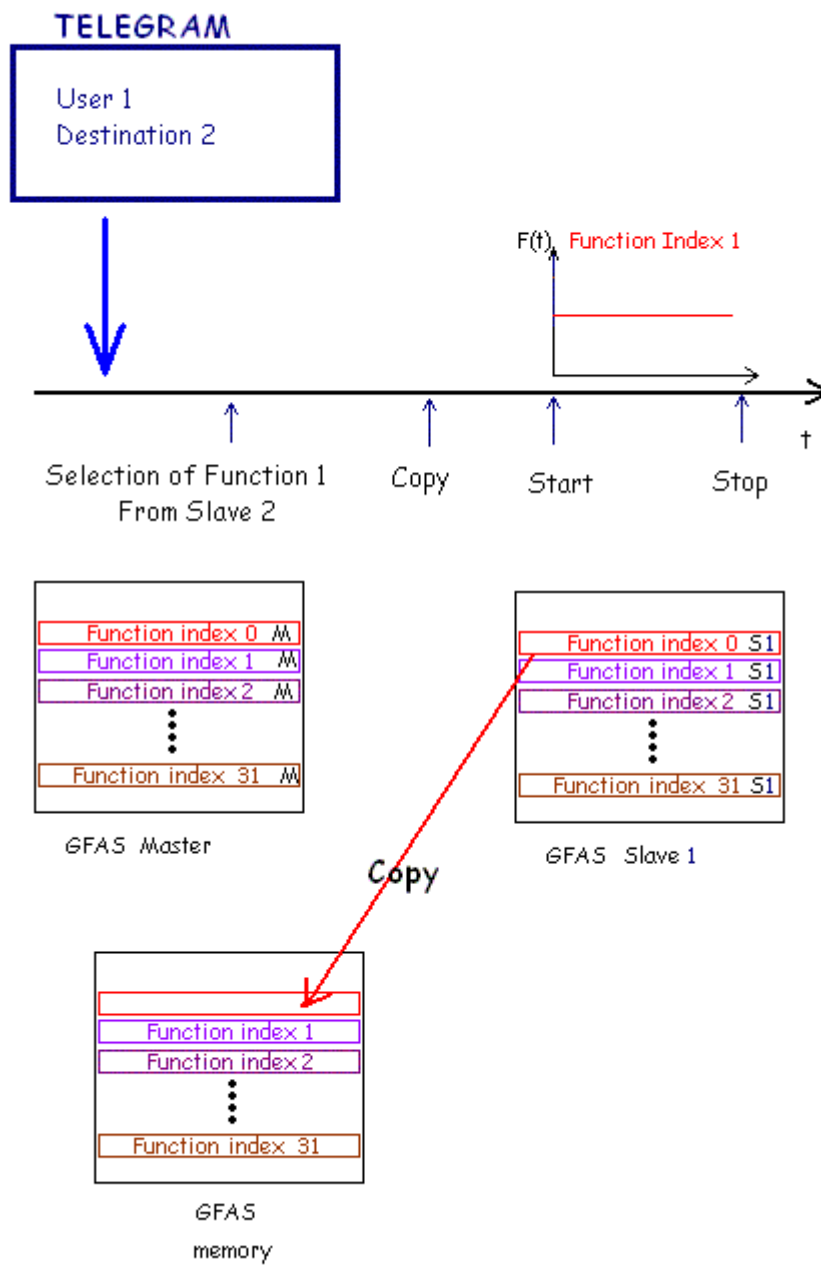


Figure 6.8b, Triple Multiplexed context.

Chapter 7

Hardware specification

This chapter deals with the inputs and outputs available for the GFAS hardware module and also gives a complete description of the GFAS memory, the memory accessible via the VME.

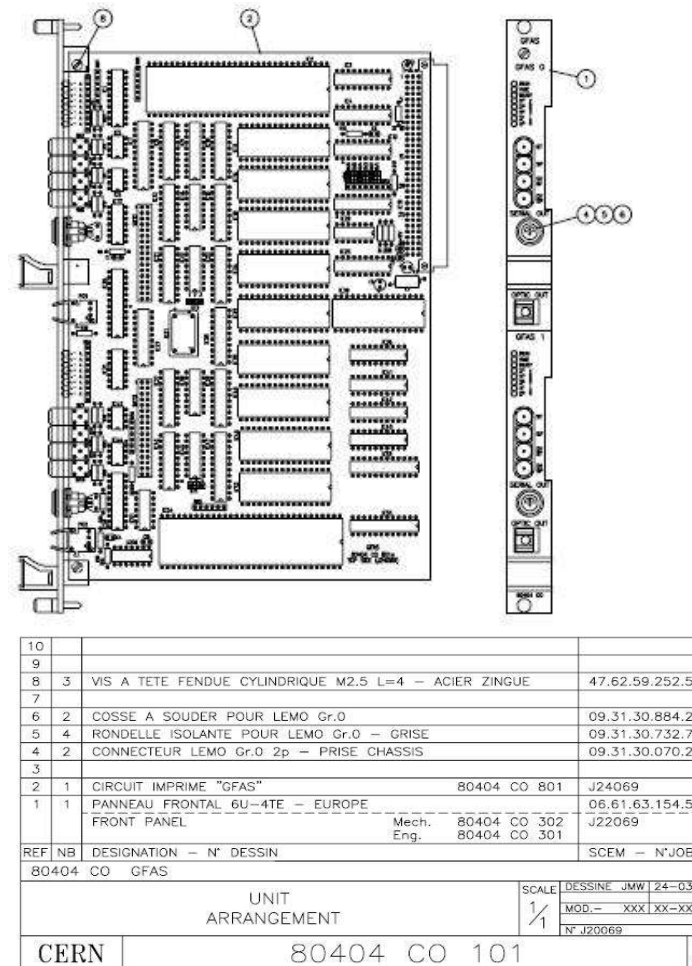


Figure 6.1, GFAS board (edms CERN document).

7.1 GFAS input/output

For each channel we have some LEDsto show the user the status of the device.

- RUN LED: is on when the GFAS has finished its hardware initialisation and its CPU is running.
- VME LED: is on when the CPU is looping on the VME memory, reading the command mailbox or converting a function.
- BUSY LED: is on when the CPU is generating a function.
- 5 LEDs associated to 2^4 , 2^3 , 2^2 , 2^1 2^0 : to show the index of the function which has been selected. The function number 0 will not light a LED.

We also have four switches for each channel:

- ST: for hardware main start event
- SP: for hardware main stop event
- EV ST: for hardware start event
- EV SP: for hardware stop event

There are also a serial output and an optical output for each channel.

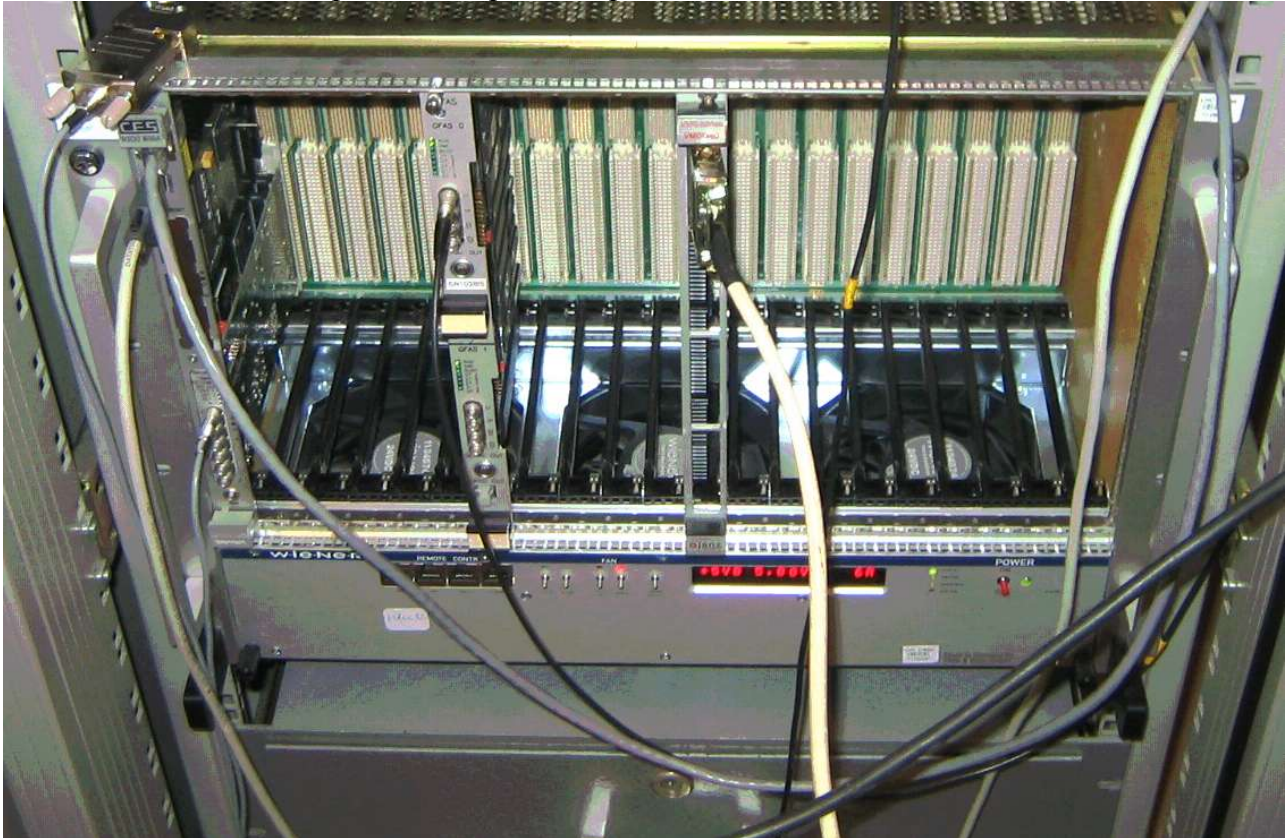


Figure 7.2, GFAS board plugged in Psdsc12 VME.

7.2 GFAS memory organization

Now let's analyse how the GFAS memory is organized.
We can distinguish three main parts:

- *Function table* where up to 32 functions can be stored.
- *Command Mailbox* where hardware operations on the GFAS can be set and triggered.
- *Status Mailbox* where the status of the GFAS is described by hardware operations.

7.2.1 Function table

We can store up to 32 functions (from address 0x0000 to 0x17FFE). Every function is 1536 words (3072 byte) and it is described by the following representation:

- | | |
|--|--------------------------|
| – number of triples of the hardware representation | (unsigned short 2 byte), |
| – number of vectors of the physical representation | (unsigned short 2 byte), |
| – origin amplitude of V(0) | (unsigned short 2 byte). |

Then for each vector V(i) (i=1.....510) (510 vectors) we have:

- | | |
|--|--------------------------|
| – number of steps from V(i -1) amplitude value to the V(i) one | (unsigned short 2 byte), |
| – step size | (signed short 2 byte), |
| – amplitude value V(i) | (unsigned short 2 byte). |

7.2.2 Command Mail Box

After the memory reserved to load the functions we have 4 bytes always 0x0000 at the offset 0xBFA and two byte at the offset 0xBF0 which are not taken into account. At the offset 0x18000 we have the words used to set hardware command for the GFAS:

- FUNCTION TO BE CONVERTED (16-31) is a vector of bits; each bit stands for one of the sixteen functions with index between 16 and 32. (0x18000)
The value of the i-th bit is 1 if the i-th function must be converted according to GFAS internal-memory representation, and is '0' otherwise.
- FUNCTION TO BE CONVERTED (0-15) is a vector of bits; each bit stands for one of the sixteen functions with index between 0 and 15. (0x18002)
The value of the i-th bit is 1 if the i-th function must be converted according to GFAS internal-memory representation, and is '0' otherwise.
- FUNCTION TO BE GENERATED is an unsigned short int (2 bytes) with the index of the function that is chosen to be generated by the module. (0x18004)
- MODE is an unsigned short used to select one of the different modes of GFAS:
normal (0), DAC (1), LowJitter (2), normal recurrent(3), low jitter recurrent(4), test1 (5), test2 (6), no function generation (7) . (0x18006)
- SLOPE is an unsigned short with the value of the slope (0x18008)
- DAC VALUE is an unsigned short with the value of the DAC (see DAC mode in par. 5.1) (0x1800A)
- RECURRENT CYCLE is a signed short with the number of times that every function should be repeated. If the value is -1, it indicates infinite repetitions. (0x1800C)
- FUNCTION TO BE DISABLED (16-31) is a vector of bits; each bit stands for one of the sixteen functions with index between 16 and 32. (0x1800E).

The i-th bit is 1 if the i-th function must be disabled otherwise it is 0.

- FUNCTION TO BE DISABLED (0 -15) is a vector of bit, each bit stands for one of the sixteen functions with index between 0 and 15. (0x18010).

The i-th bit is 1 if the i-function must be disabled otherwise it is 0.

After some bytes that are not used there are:

- FORCE CONVERSION is a flag of 2 bytes. It must be 1 if we want to trigger the forced conversion (not just the common conversion) of the functions of selected in 'function to be converted' words. (0x1A000)
- CONVERT FUNCTION is a flag of 2 bytes. It must be 1 if we want to trigger the conversion of the functions of selected in 'function to be converted' words. (0x1A002)
- SELECT FUNCTION is a flag of 2 bytes. It must be 1 if we want to select the function specified in the 'to be generated' words. (0x1A004)
- SOFT MAIN START is a flag of 2 bytes. It must be 1 if we want to cause a software start. (0x1A006)
- SOFT EVENT START is a flag of 2 bytes. It must be 1 if we want to cause a software event. (0x1A008)

7.2.3 Status Mail Box

After some bytes, which are not used, there are the bytes used to describe the GFAS hardware status:

- FUNCTION ACCEPTED (16-31) is a vector of bits; each bit stands for one of the sixteen functions with index between 16 and 32. (0x1C000).
The i-bit is 1 if the i-function is accepted; otherwise it is '0'.
- FUNCTION ACCEPTED (0-15) is a vector of bits; each bit stands for one of the sixteen functions with index between 0 and 15. (0x1C002). The i-th bit is 1 if the i-th function is accepted, otherwise it is '0'.
- STATUS WORD is a vector of 16 bits with the description of the status of the GFAS:
 - bit 0...4 number of the selected function
 - bit 5 CPU BUSY the CPU is generating the function (1) or not (0)
 - bit 6 VME READ the CPU is looping reading the command for doing function conversion (1) or not (0).
 - bit 7 CPU RUNNING the GFAS finished the hardware initialisation and the CPU is running (1) or not (0).
 - bit 8 FUNCTION ABORT (1) or not (0)
 - bit 9 FUNCTION PAUSED (1) or not (0)
 - bit 10 FUNCTION INVALID (1) or not (0)
 - bit 11..15 MODE SELECTED: normal (0), DAC (1), Low Jitter (2), normal recurrent (3), low jitter recurrent(4), test1 (5), test2 (6), no function generation (7) (0x1C004)
- FUNCTION DISABLED (16-31) is a vector of bits; each bit stands for one of the sixteen functions with index between 16 and 32. (0x1C000). The i-th bit is 1 if the i-th function is disabled otherwise it is 0.
- FUNCTION DISABLED (0-15) is a vector of bits; each bit stands for one of the sixteen functions with index between 0 and 15. (0x1C002). The i-th bit is 1 if the i-th function is disabled otherwise it is 0.

Chapter 8

FESA Framework

At this point we describe the FESA framework, this description is not meant to be exhaustive about FESA but just to give the basic notions to understand the structure of the code that I produced for the GFAS.

8.1 Framework Overview

Equipment software provides a stable and homogeneous functional abstraction on top of accelerator equipment whose hardware implementation is heterogeneous and evolves over time. The different types of software equipment used in the accelerator chain can work in different accelerators and as consequence in different *timing domains*. Each accelerator has its particular kind of telegrams and events that can trigger its real time activity: this context is referred to as timing domain. The accelerator complex provides beams for several users, making it a shared resource that relies on a time-multiplexing scheme orchestrated by the central timing system: the basic period of the accelerator is split as a set of successive time-slots during which the settings of a specific cycle (see par 3.4) stay valid.

The framework approach aims at defining a software package providing a partial and generic solution to equipment-software, and which can be refined when applied to specific equipment. The purpose of the FESA framework is to encapsulate aspects of equipment-software development as a reusable software package that can be tailored, or customized, on a case by case basis. The generic software package contains a set of base classes that encapsulate the essentials or key concepts of front-end software. In this case, customization reduces to deriving concrete classes from the base classes and implementing them to suit specific needs.

As shown by the use-case diagram below, a crucial part of the control infrastructure is located at the junction of two worlds:

- the communication with the control-room's computers to handle operator requests,
- the satisfaction of hardware and real-time constraints.

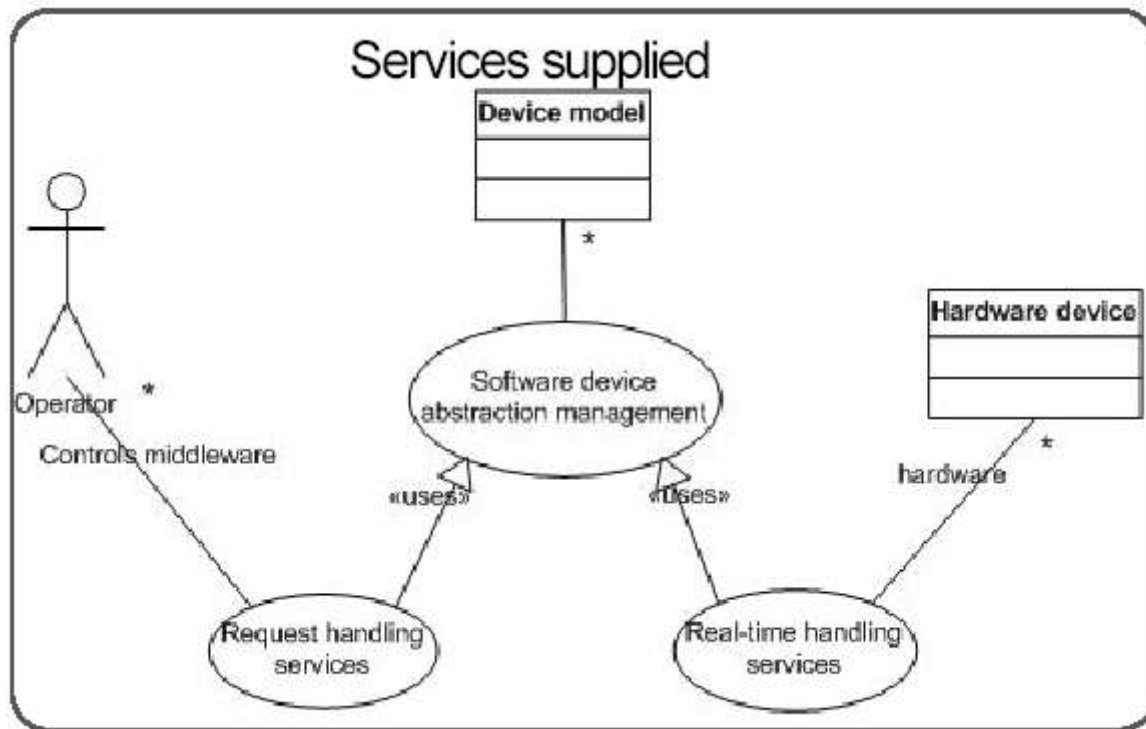


Figure 8.1, general user-case diagram (from FESA Essential).

Their nature is different: one is an on-demand service and the other has tight to real-time constraints. Obviously, request handling must run at a lower level of priority and shall not be able to pre-empt and disturb the real-time task. In order to decouple the two parts, the equipment software includes a software abstraction of the device and thanks to this abstraction, an operator does not directly see the hardware device, but rather accesses it through its proxy. [9]

The software equivalent of an underlying hardware device is a data-holder that contains attributes which can be settings, acquisitions, or dynamic state-variables, and whose values at any given time provide an accurate snapshot of the underlying hardware device. [9]

An equipment-software's core activity is to ensure that both the software abstraction and its underlying hardware device continuously reflect each other's state at runtime.

Ensuring such a real-time correspondence involves information flowing in both directions:

- from the device model and down to the hardware (controls flow);
- from the hardware and up to the device model (acquisitions flow).

Such transfers are usually synchronized by the accelerator's central timing system which orchestrates machine activity. After what we have seen we can say that equipment software is made by three parts: a server component implementing request-handling function, a real-time task implementing real-time handling and a memory segment embedding the software model of the device. This structure is depicted by the next diagram.

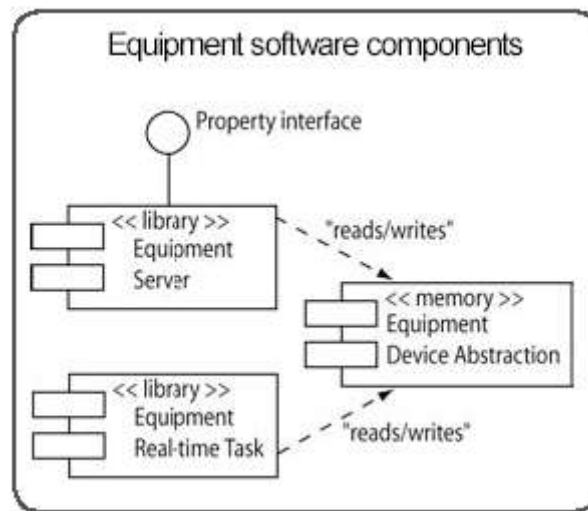


Figure 8.2, Equipment software components (from FESA Essential).

Development of new equipment software always involves defining the device model and coding the request-handling and real-time handling. For performance reasons, C++ is the programming language of choice for developing real-time equipment software targeted at LynxOS (a real-time version of the Linux operating system). In spite of the overwhelming diversity of accelerator equipments, development of equipment software exhibits some routine work, thereby suggesting that some form of code reuse is achievable. To achieve this code reuse, the need of defining a framework is felt and, thanks to it, we have a partial model that can be tailored and customized, on a case by case basis in order to suit the specific needs of the equipment specialist. In the following figure we can see the main structure of the FESA framework.

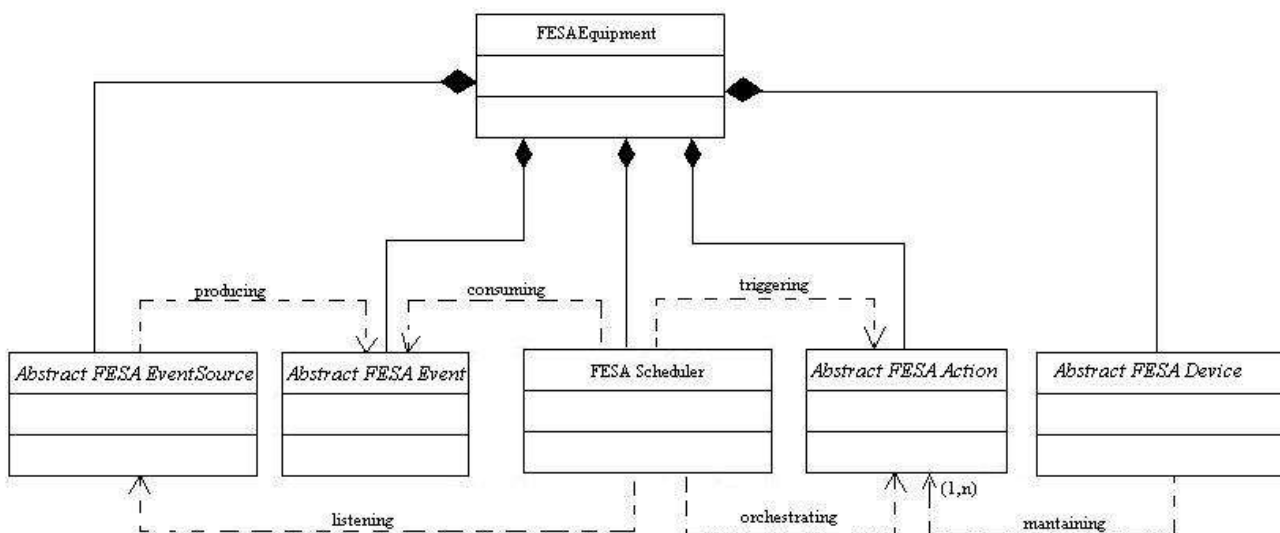


Figure 8.3, Framework Overview (from FESA Essential).

The foremost classes can be considered:

- the *event-source* which represents the core of any equipment software activity and produces all the events that are used to trigger the right action at the right time,
- the *FesaEvent* which wraps hardware interrupt, machine events or client requests as objects [20],
- the *FesaSchedule* which is the go-between for events and actions. It listens to the event-sources and whenever an event occurs, it triggers an appropriate action according to some pre-defined logic. [9]
- the *FesaAction* which is the basic work unit carried out by the equipment software, and is where the equipment specific functionality is coded. [20] Actions can be further categorized as being either real-time actions, that is to say actions that deal directly with the hardware, or they can be server-actions that serve and fulfill operator requests from the control room.
- the *FesaDevice*, it represents the set of variables needed to reflect the state of the underlying hardware device, these variables may belong to the settings, acquisitions or state-variables categories.

In figure 8.4 we have the description of these core classes.

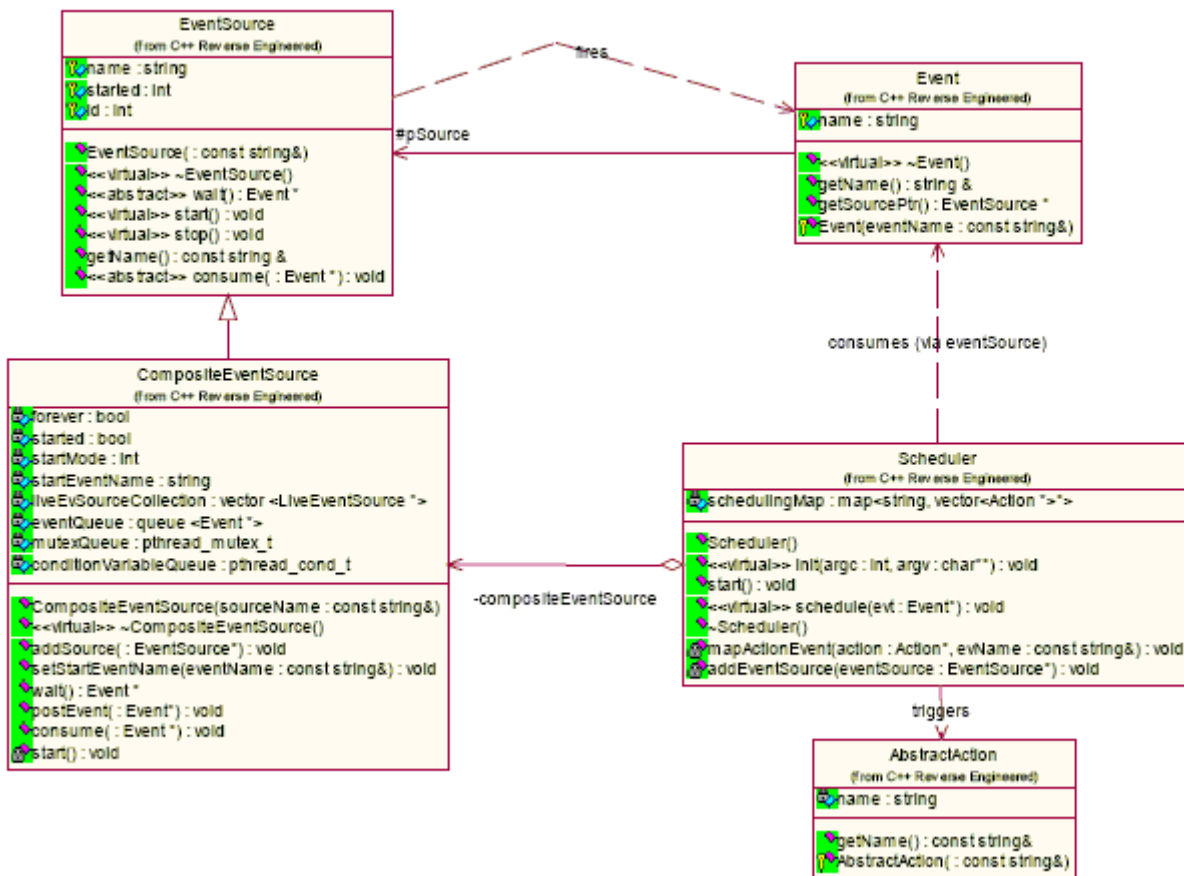


Figure 8.4, Detailed framework Overview (from FESA C++ design).

Only the Abstract classes (shown in figure 8.5 as grey boxes) must be implemented by real classes for the particular equipment to tailor the framework package for a specific equipment class and to do that the equipment specialist needs to describe the design of the particular equipment (see par. 7.3).

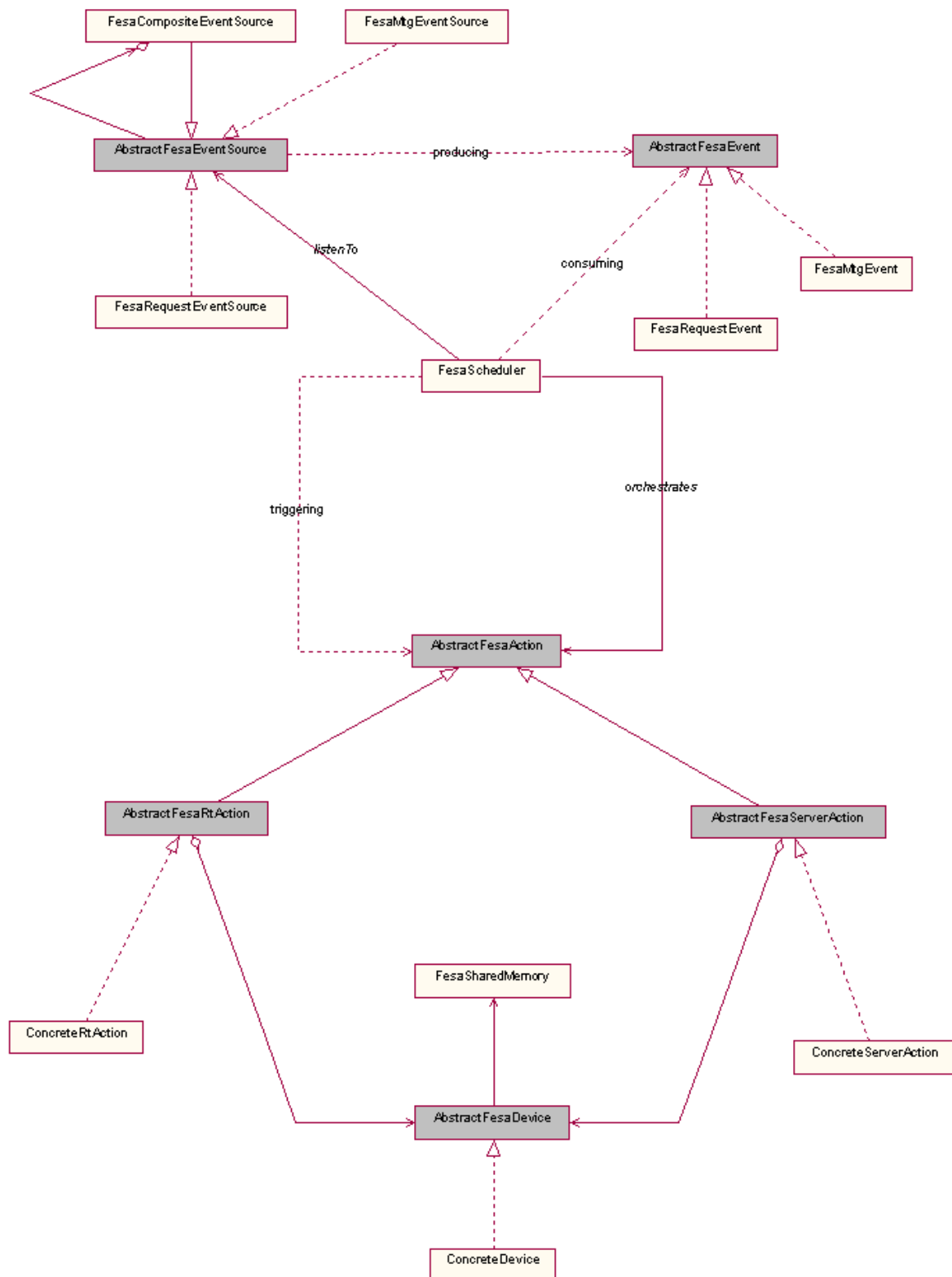


Figure 8.5, Main FESA classes (from CERN Front-End Software Architecture for accelerator controls).

Now let us analyse the different packages of the framework.

8.2 Framework packages

The FESA framework is organized in thirteen different packages:

- 1) Cmw,
- 2) Core,
- 3) Interface,
- 4) Rt,
- 5) Datastore,
- 6) Persistency,
- 7) Synchronization,
- 8) Exception,
- 9) Utility,
- 10) Logging,
- 11) Recorder,
- 12) Sorting,
- 13) PLC.

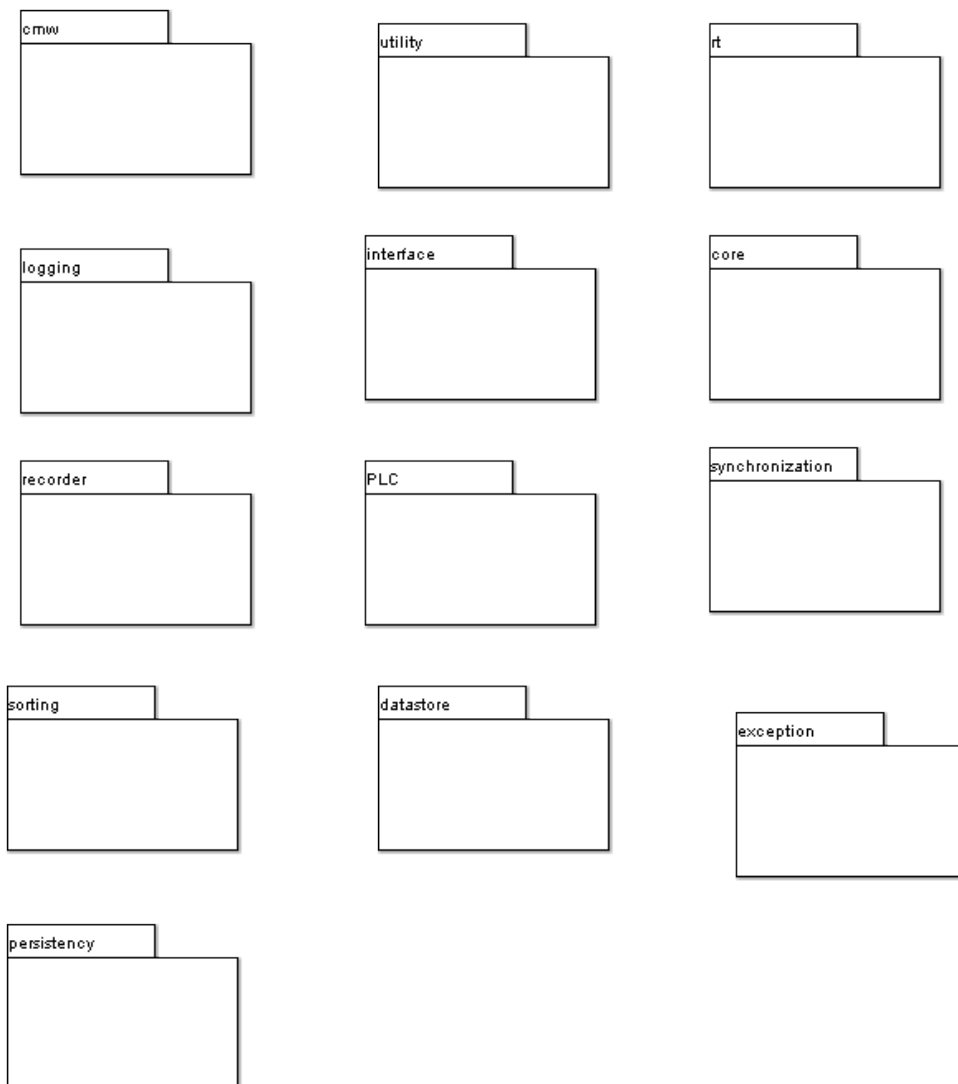


Figure 8.6, Fesa Framework packages.

8.2.1 Cmw

In this package we have the FESA classes which handle the client server relationship using the Control Middleware, that I have already introduced in par. 3.6.2.

We have an abstract middleware which is currently implemented by the FESA MW using the rda.

The rda is an external package which realize the remote device access with CORBA (the Common Object Request Broker Architecture) technology. The CORBA architecture is completely transparent for FESA environment then another technology can be used in the future without legacy problems.

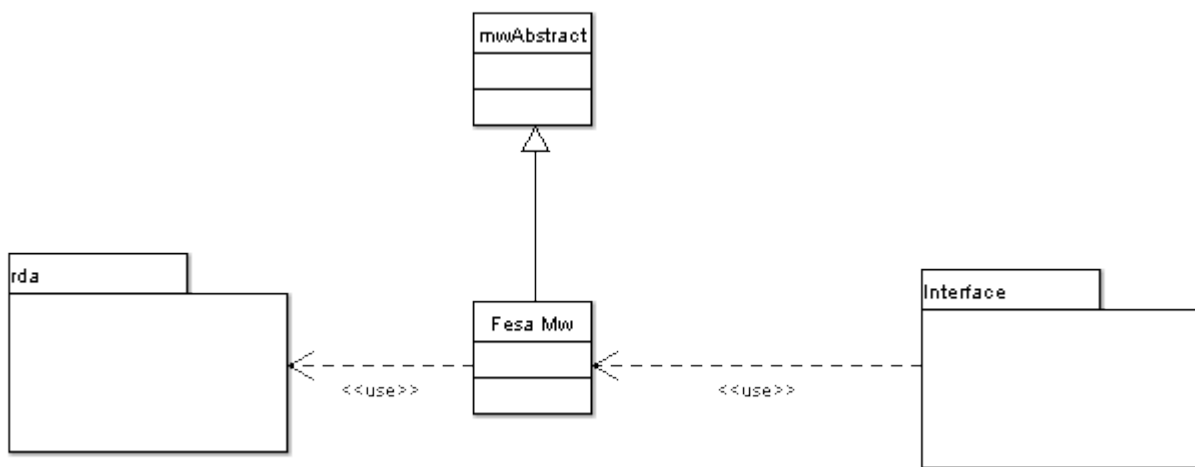


Figure 8.7, FESA middleware.

8.2.2 Core

An important implemented feature, which deals with the relationship between the server and the real-time part, is notification. A client may subscribe a property for a particular device and cycle, then he will not receive the data right away but only when the real time action, associated to it, finishes its execution. As shown in the next sequence diagram, at the end of the execution of the real-time action, a message is sent to the Equipment interface saying that the real-time has finished the execution. Then the Equipment interface, in an asynchronous way from the user point of use, notifies that to the middleware server which can give the client the updated snapshot of the device.

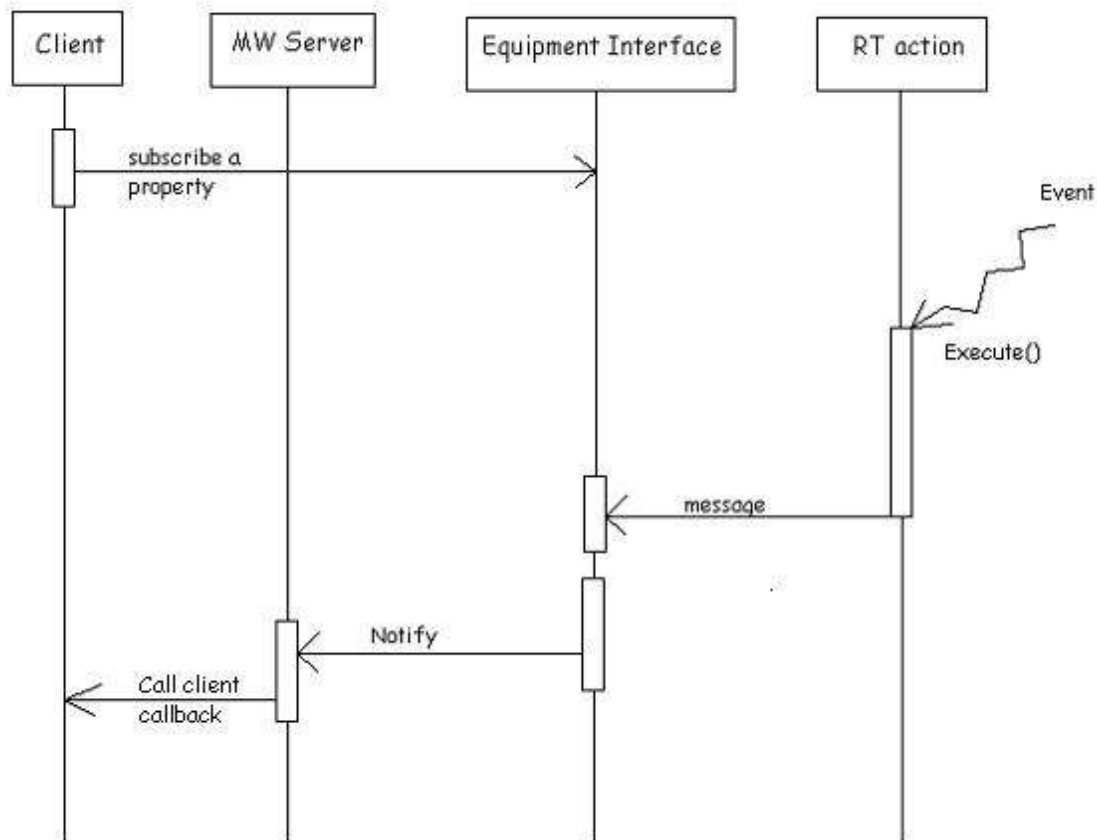


Figure 8.8, Notification.

Notifier and its children LocalNotifier and MQNotifier are used to handle all the operations related to notification, the first notifier is used if real time and server are executed in the same process as two different threads and the second one is used if we have two distinct processes instead of threads. We will talk about it in details in par 8.3.4.

In the Core package we have also the definition of the AbstractEvent and of the AbstractEquipmentClass. The AbstractEvent has the operations to retrieve an event from the AbstractEventSource. The AbstractEquipmentClass is the building block of the representation of the Equipment; it declares all the common methods and attributes that define a generic equipment module. UserEventQueueConsumer and UserEventQueueProducer are declared here and are used by UserEventsSource (see par. 8.2.4).

In this package we have two particular .h files :

- Fesa.h which is used to encapsulate all the include directives concerning all the .h files needed to use FESA framework.
- FesaDefs.h which is used to encapsulate all the define directive needed to use FESA Framework.

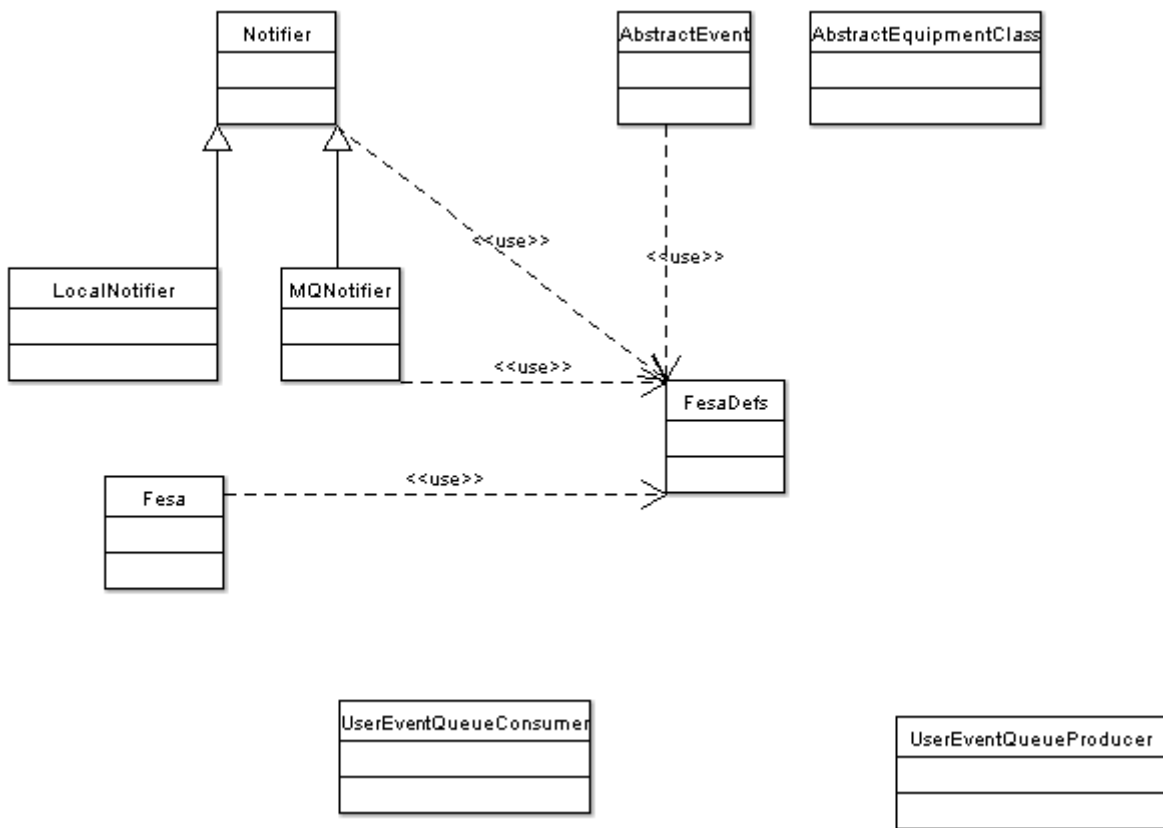


Figure 8.9, Classes of the Core package.

8.2.3 Interface

The interface defines the set of services published to the outside (clients from the control-room or middle-tier software layer). Designing equipment's interface involves listing so-called «properties» that can be remotely accessed through the controls middleware.

Properties are public methods of an equipment class that a client accesses in read (get) or write (set) mode. In both operations the remote invocation of the property causes some server-side processing: getting a property causes the property data to be computed by the server before being transmitted to the requester and setting a property triggers some server-side computation on the input parameter.

We can define some simple properties which just get or set some data fields of the equipment or complex properties which call non trivial server action. In the first class of properties the default get/set Server Actions are automatically provided; on the other hand, for complex properties, the code of this action must be written by who is defining a FESA Equipment model.

In the following pictures, we have in blue the class diagrams of the classes that the equipment specialist must implement.

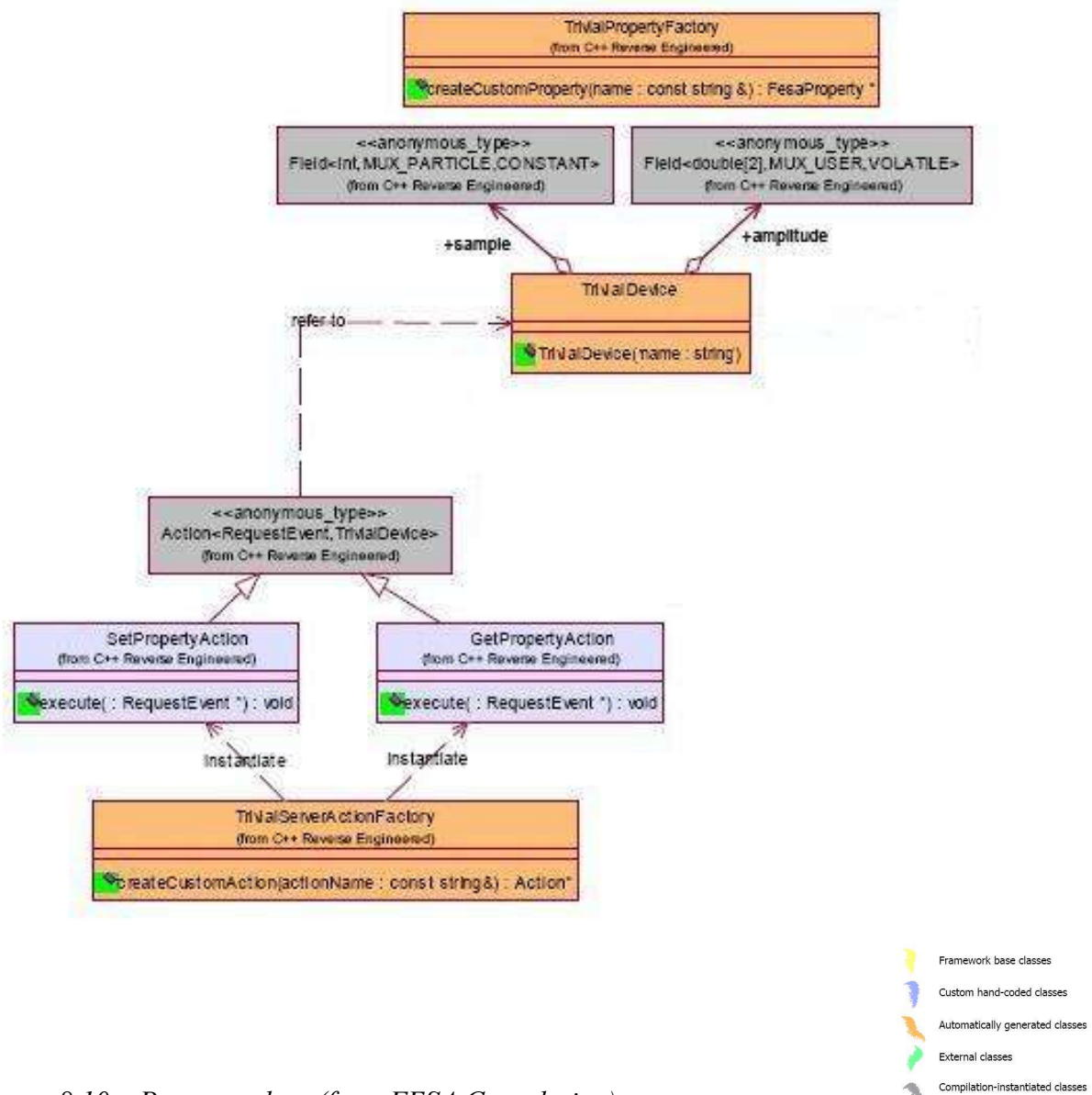


Figure 8.10 , Property class (from FESA C++ design).

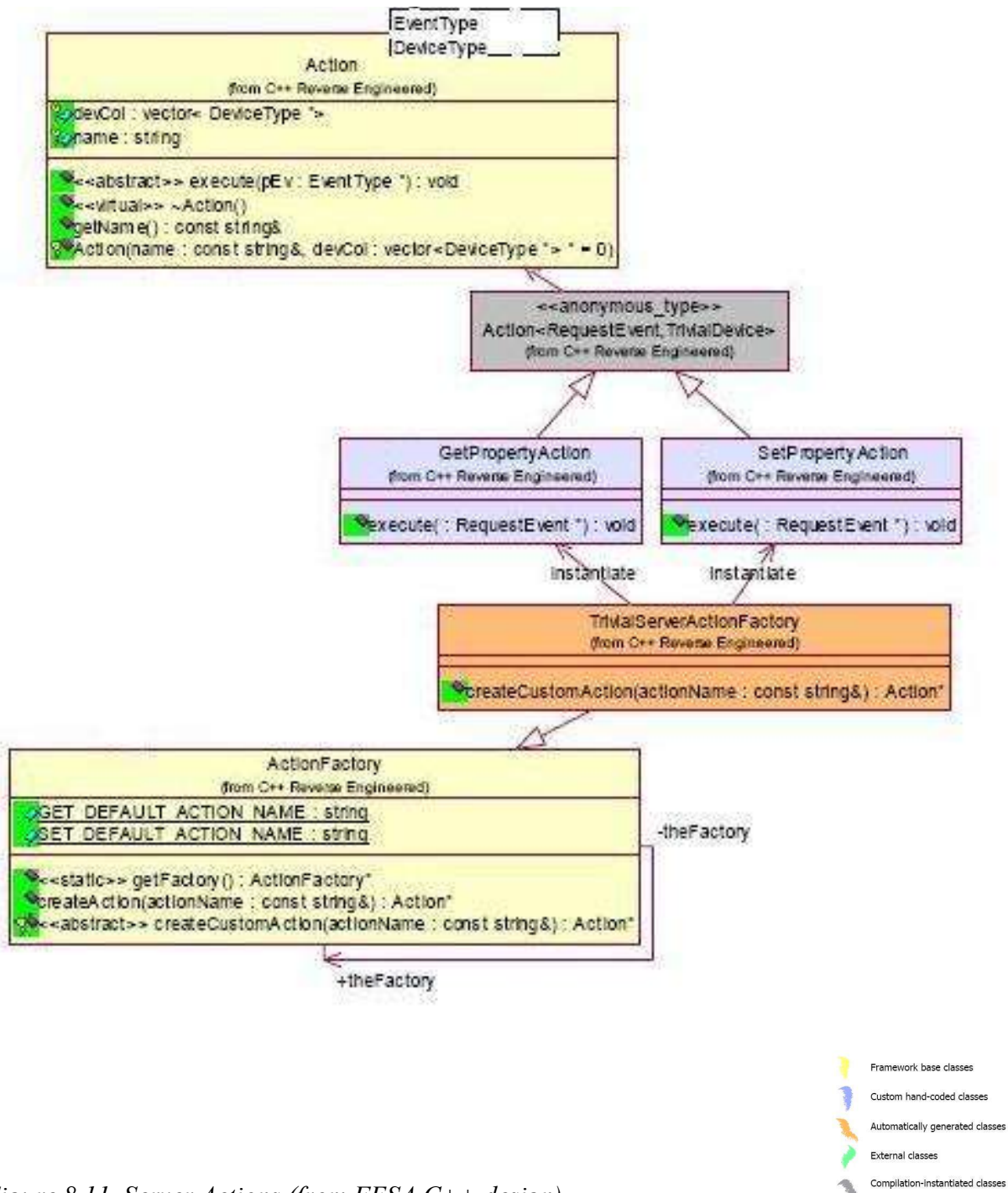


Figure 8.11, Server Actions (from FESA C++ design).

Also complex properties may be associated with one or more filters. Each filter has a value chosen by the client in a range of possible values and this is used as discriminator in the code to modify the behaviour of the associated property.

If we analyze the classes in this package, we have the **AbstractEquipmentInterface** which define the base class of any **Equipment Interface** and it implements the basic methods. This class is extended by **EquipmentInterface** which is a template class with **GlobalStoreType**, **DomainStoreType** (see par 8.2.5) and **DevInstType** as parameters. **DevInstType** makes possible to instantiate a class for a particular type of device.

EquipmentInterface keeps a reference to the **DeviceFactory** (see par. 8.2.5) to be able to retrieve the device collection. It also manages the property collection. Each property maintains the link to its

own get/set server actions . It has also to manage the different properties and the actions related to them and finally it has to make accessible all the fields with the data related to a particular device. EquipmentInterface uses the ServerActionFactory to create and to customize the AbstractServerActions, this class is used to provide the default get/set actions which are used in the simple properties.

Finally we have the ServerAction which extends the AbstractServerActions. It is a template class and its parameters are GlobalStoreType, DomainStoreType (see par 8.2.5), Data_Type and Filter_Type. Data_Type is used to distinguish the different server action declared and the Filter_Type to make it possible to implement variations in the code depending to the value of some filters.

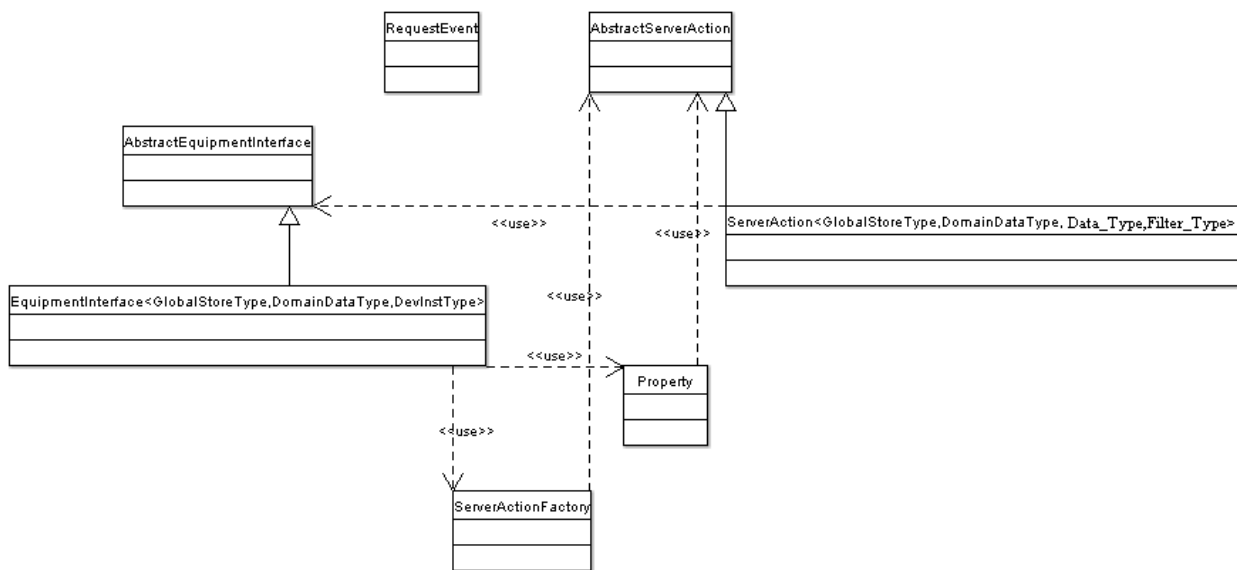


Figure 8.12, Classes of Interface package.

8.2.4 RT

This package handles the operations concerning the real time activity, i.e. the relationship between the device, the real time actions and the events. We have different kinds of event source which can be custom sources (completely defined by the equipment specialist) or timing event sources which are related to the events produced by the tgm. The producted events act as triggers for real time actions and the scheduler is used in between to associate the right action to each event.

Until now we are talking of logical events but, associated to a single logical event, we can have one or more real events and the same real-time action can be triggered by different real events with the same result.

Each time domain has a set of real-time events; Logical events instead do not depend on the time domain, and can be associated to one or more real events. Only when we create an instance of the equipment, we chose the time domain and the real events that we use.

We can use a sequence diagram to explain better how the FESA framework handles the real-time activity.

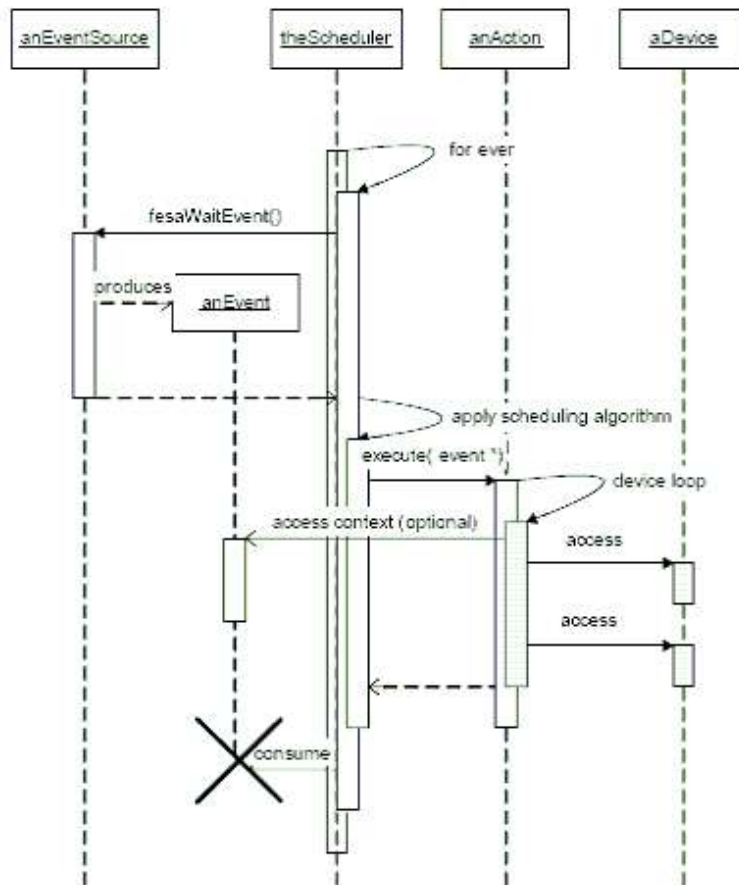


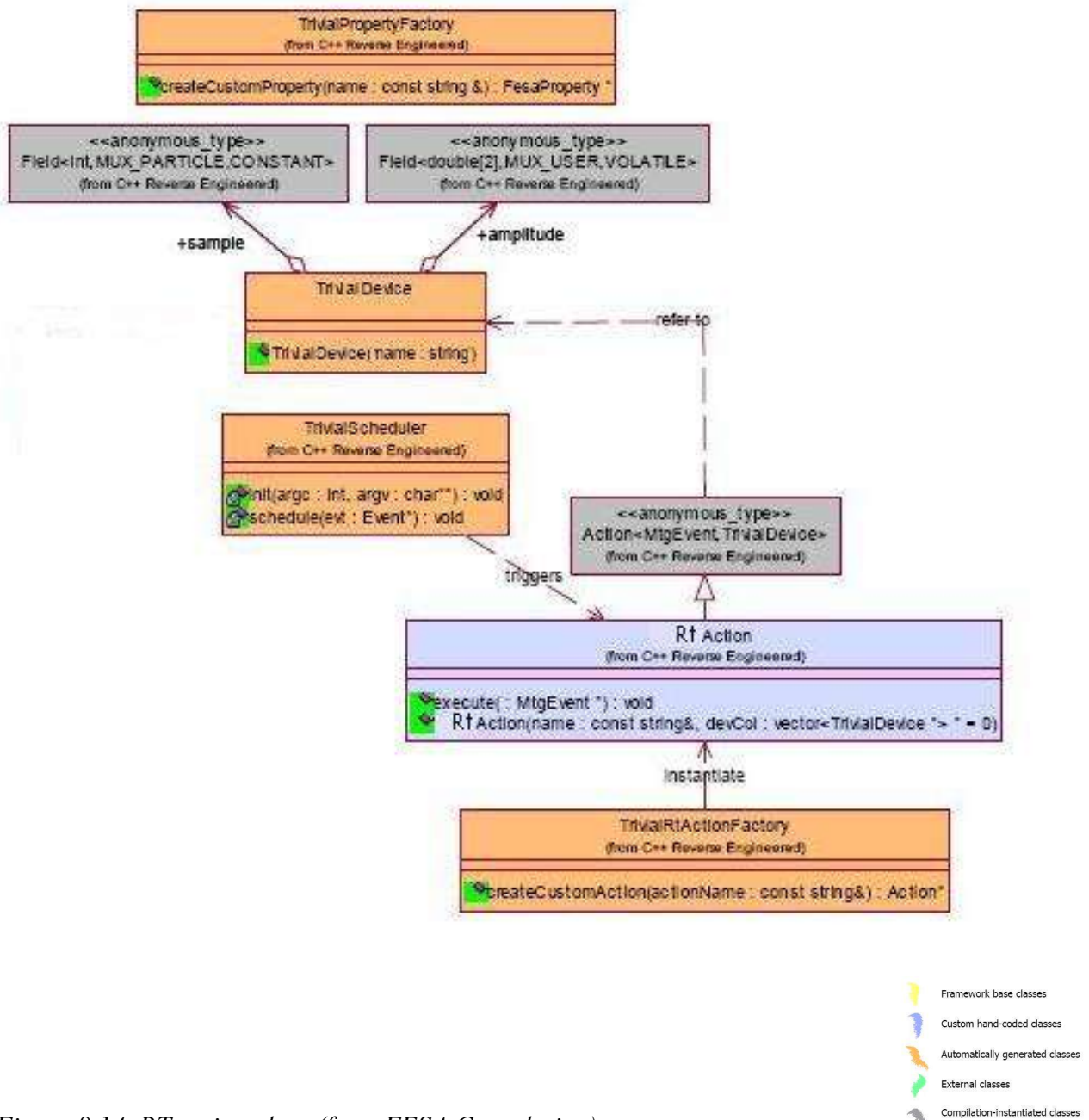
Figure 8.13, Real-time sequence diagram (from CERN Front-End Software Architecture for accelerator controls).

The scenario depicts the typical behaviour of an RT-task observed after customizing the FESA framework for a specific device and set of actions. First, the RT-task subscribes to events fired by the central timing process (or any other HW sources). This process orchestrates the accelerator's pulsed-mode of operation, and the RT-task block-waits on event reception. Whenever an event occurs, the framework examines its type and triggers the appropriate action as specified in the equipment configuration. In the sequence diagram, activities represented by rectangles filled with a dashed pattern, are those that differ from one RT-task to another and that the programmer does code.[20]

This diagram illustrates two key points:

- 1- The specific (i.e. custom) part of an RT-task essentially lies in the body of the elementary actions that the RT-task triggers.[20]
- 2- The scheduling algorithm whereby the dispatcher triggers appropriate actions in response to incoming events is provided by the FESA framework. Furthermore, the scheduling algorithm is fully configurable by the equipment specialist. [20]

As shown by the next diagrams the equipment specialist should only provide the code that is executed by the real-time actions, it means that he should implement only the `execute()` methods of the real time actions defined.



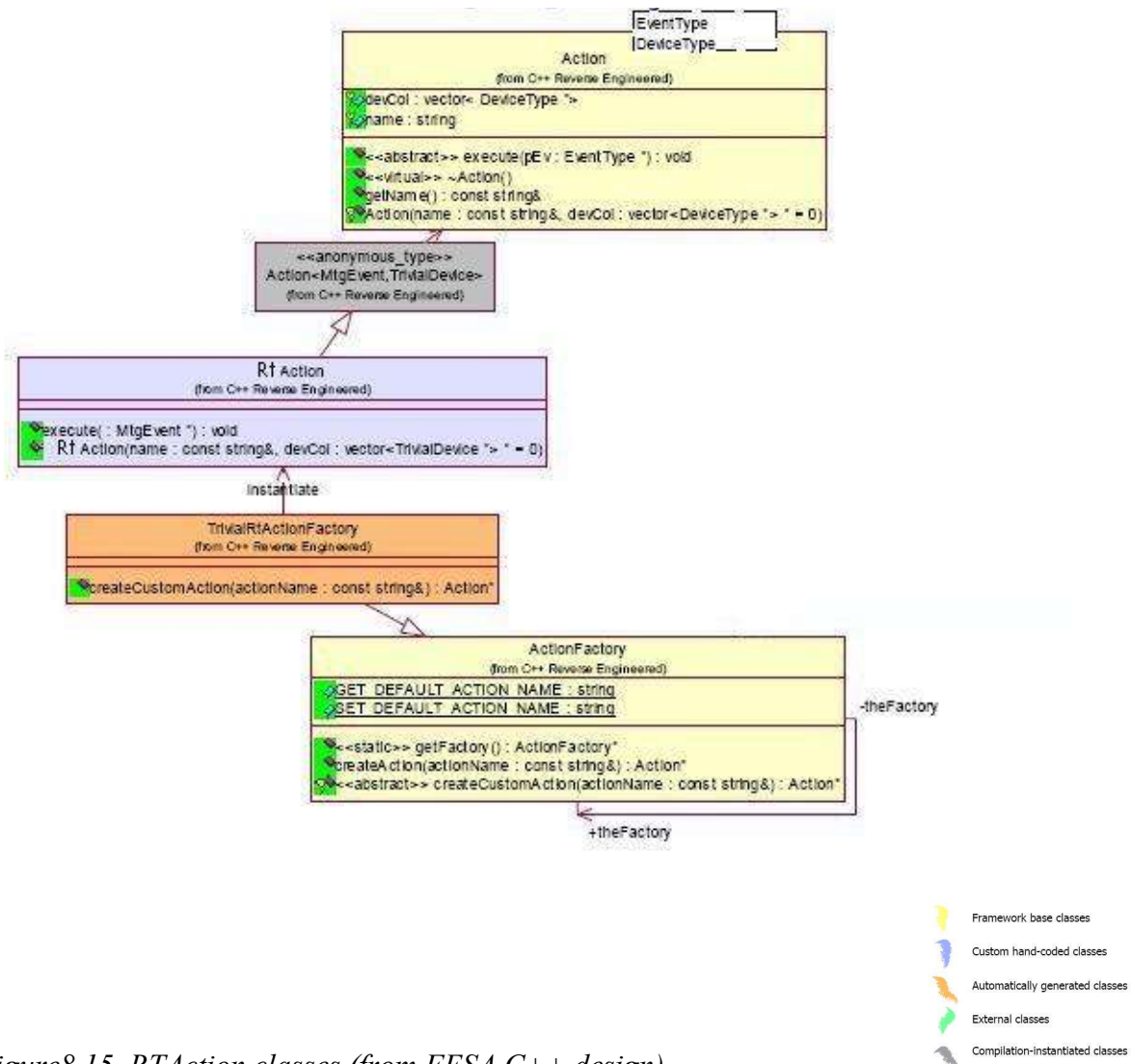


Figure8.15, RTAction classes (from FESA C++ design).

Analysing in detail the classes of this package we have the `abstractEquipmentRt` which is the mother class of `EquipmentRt`, a template class with `GlobalStoreType`, `DomainStoreType` (see par. 8.2.5) and `DevInstType` as parameters. As it is in the server part, `DevInstType` makes possible to instantiate a class for a particular type of device. This class uses the `RTActionFactory`, the `EventSourceFactory` and `RTScheduler`.

`RTActionFactory` produces `AbstractRTAction` and from that the `RTAction` is implemented and `EventSourceFactory` produces `AbstractEventSource` that can be implemented as:

- `CompositeEventSource`
- `TimerEventSource`
- `TimingEventSource`

The `AbstractEventSource` and also all its children uses `RTEvent` which is extended by `TimingContext`, which uses the `Timing` class, adding all the information related to the particular time domain.

Concerning the `RTScheduler`, it is the child class of `RTSchedulerBase` which is also the parent of `RTSubScheduler`. Both `RTScheduler` and `RTSubScheduler` handle the relationship between events and actions and the sub-scheduler is used to link more than one real event to a logical one.

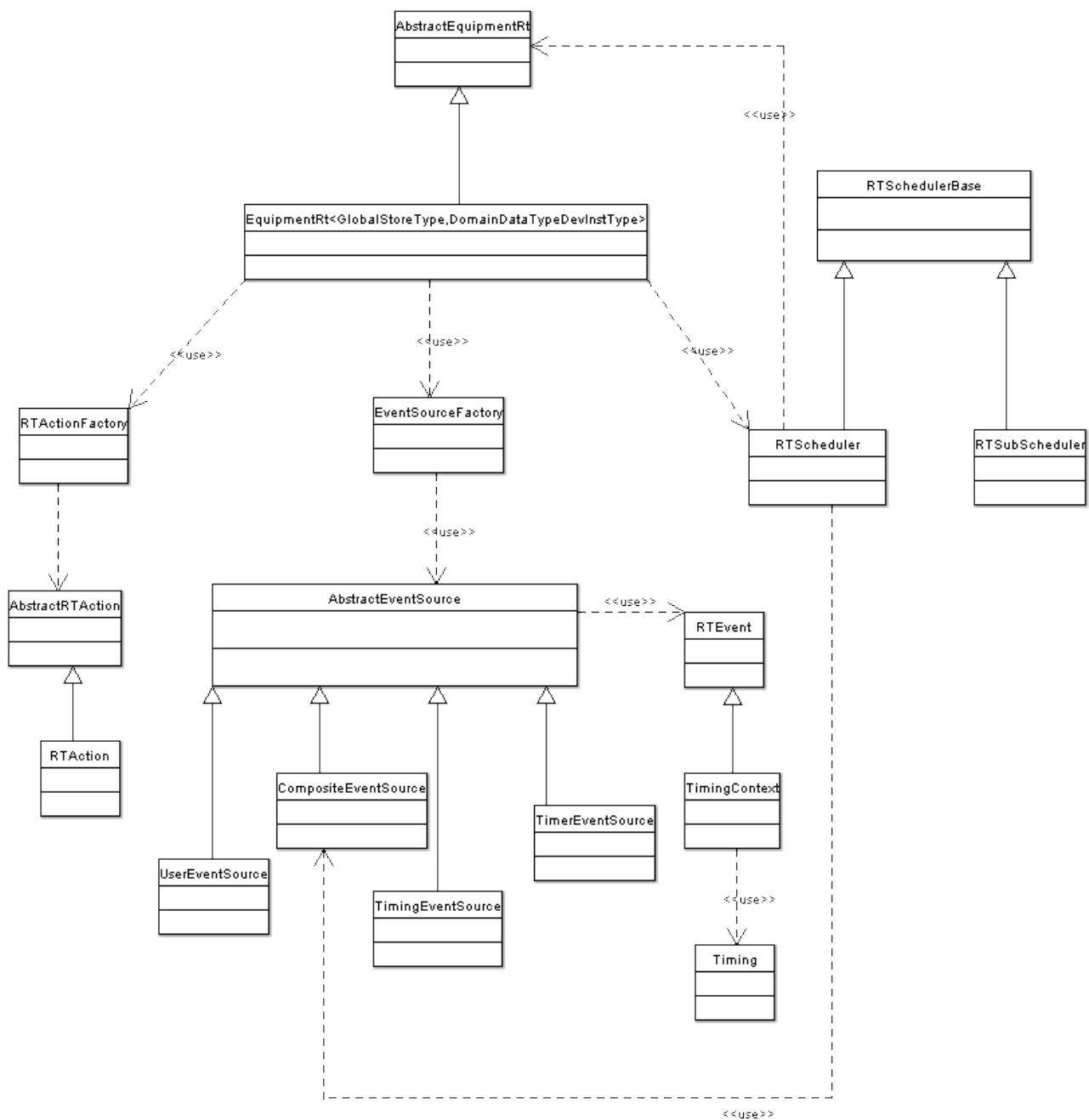


Figure 8.16, Classes of RT package.

8.2.5 Datastore

This package defines all the classes needed to describe the device.

The device model represents the software abstraction of an underlying device and it can be considered a data holder of fields that are continuously updated and transferred to and from the hardware in order to ensure that the real device and its software proxy reflect each other's state at run-time. Fields are made for the fine grained fabrics of the device model and every piece of information about the underlying hardware device is stored in them. They are full fledged objects that provide access methods, get and set, to store and retrieve their value (we will see it later).

We have different kinds of fields:

- *Hardware address fields*: they consist of a three-part combination of type / logical-unit / channel string fields for encoding the hardware addressing of a device. The type designates the hardware board family. The logical unit typically identifies the board index within the VME crate. The channel typically refers to a specific port of the board which is used to connect the specific device instance. For devices which are connected through more than one hardware module, it is possible to rely on up to three sets of such address fields
- *Fields*: the framework does not impose any detailed class hierarchy for these standard fields. The programmer has complete freedom for defining what fields actually stand for and they are the ones used to describe the state of the equipment at each time. They could be a scalar, an array or an array2D of homogeneous elements which can be standard java types (bool, signed char, short, long, long long, float, double, char) or types defined by the user such as enumerate and bit enumerate of 16 or 32 bit if we want to work with a bit map.
- *Fault Fields*: these fields are used to trigger alarms and, to each of them, a boolean is associated to say if an Alarm has to be raised or not.
- *Interrupt Fields*: They can be referred to as implicit triggers of real-time actions by the equipment's scheduler and they are important for the real time activity. The device model may refer to one or several interrupt fields. These fields are important for define the real-time activities and each Interrupt field is associated to a particular group of real-time event.

A criterion to group fields is the multiplexing type. The accelerator complex provides beams to several users, making it a shared resource that relies on a time multiplexing scheme orchestrated by the central timing-system: the basic period of the accelerator is split as a set of successive timeslots during which the settings of a specific user stay valid. Switching from one multiplexing context to the next is triggered by the timing system, which in turn causes a switch of equipment settings from one user to the next. Accounting for this multiplexing behavior at the device level requires that fields accommodate not a single value but a set of them, namely one different value for each different user.

Possible criteria of multiplexing are:

- NONE (not multiplexed),
- USER (cycle user),
- PARTICLE (particle type),
- DESTINATION (beam target).

Another important attribute to define a field is the Persistency because Fields are written to and read from the FEC memory at run-time. For backup and data-management purposes, they can be assigned different persistency levels which are:

- FINAL (database constant),
- PERSISTENT (periodic backup save into persistent-storage),
- VOLATILE (RAM data).

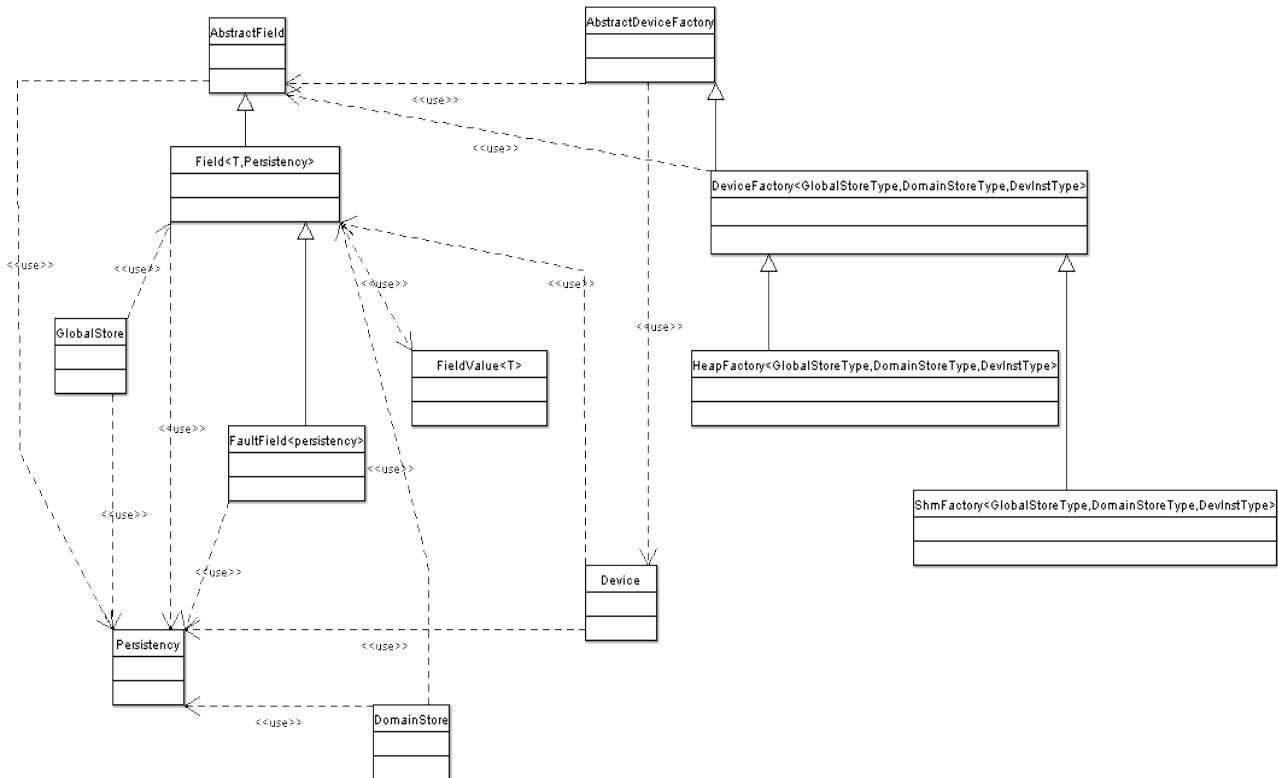


Figure 8.17, classes of Datastore package.

As shown in figure 8.13, we have the AbstractDeviceFactory which defines the main structure of the model concerning data information of the device. The Device is the result of the operations. It is the parent class of DeviceFactory which is a template class and it has three parameters of GlobalStoreType, DomainStoreType and DevInstType to adapt the general class to the particular Device. GlobalStoreType defines the set of fields that are shared between all the instances of the equipment module. They are fields owned not by a particular instance but owned by the class.

With DomainStoreType we have the same situation except that the fields are shared only by instances of a particular timing domain. DeviceFactory has two children classes: HeapFactory and ShmFactory which are template classes too. The task of these two classes is to manage the memory for the particular device. The first class handles the heap and the second is used if only if the device needs shared memory.

The Device is defined using fields and we have the AbstractField as frame of the Field which is implementing it. The Field is a template class, we can have a different Persistence (volatile, persistent, final depending how information are stored and kept in the memory) and different data type. There is a particular Field: FaultField which extends Field and it is used to trigger the alarms (we will see them later). It is also possible to specify that some property cannot be used if a particular fault happened.

As shown in the following figure the Equipment specialist does not have to write any code concerning the classes related to this package closely. The Device classes is generated automatically using the information of the design.

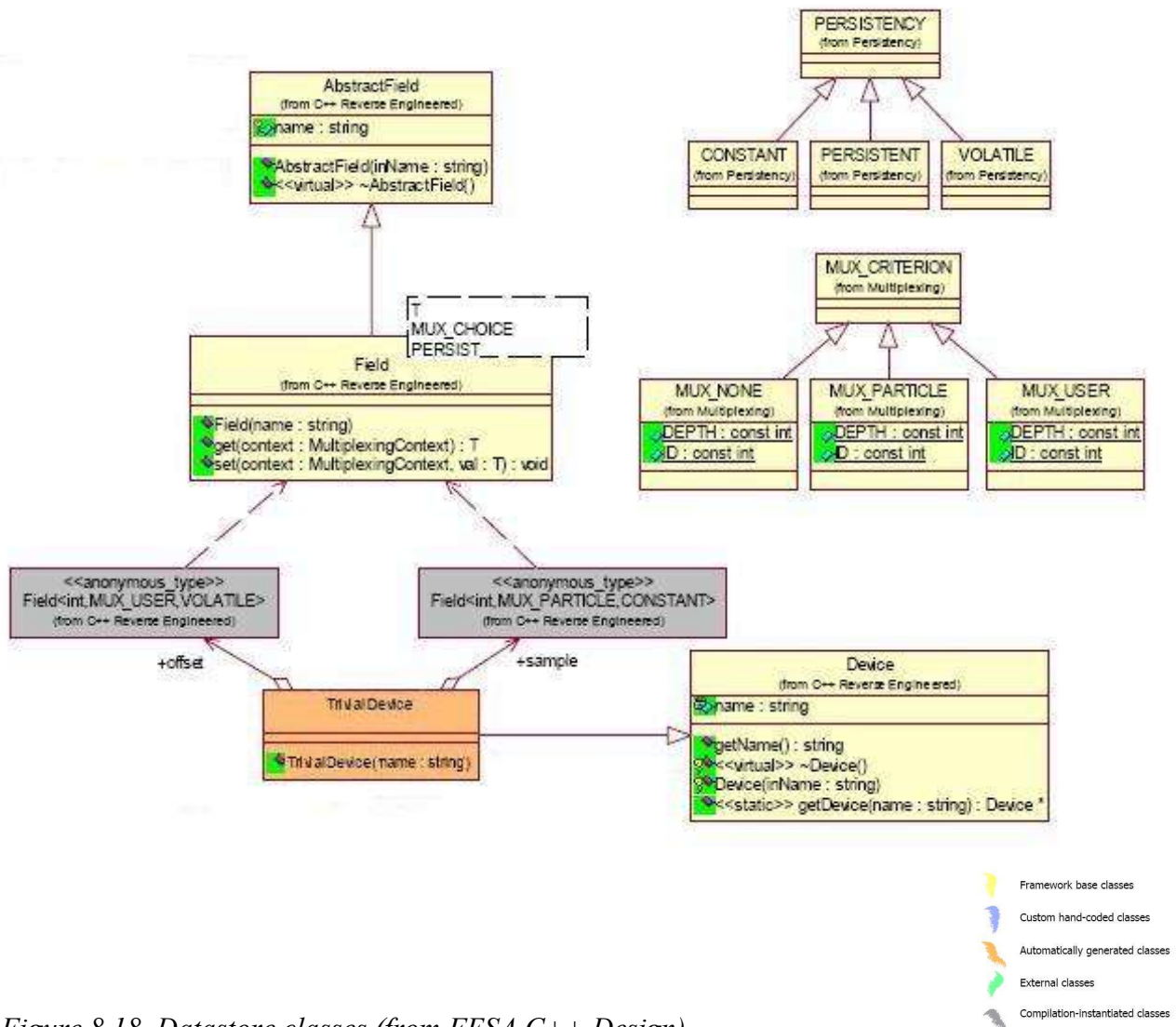


Figure 8.18, Datastore classes (from FESA C++ Design).

8.2.6 Persistency

In this package we have the PersistencyManager class which is used to obtain persistency for persistent fields, handling all the operations needed by the management of persistent storage.

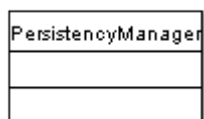


Figure 8.19, Persistency class.

8.2.7 Synchronization

Through the classes of this package the equipment software is attached to the central timing system of the accelerator complex and its real-time activity is synchronized to the overall orchestration of the machine. The telegram (see par. 4.4) is represented by the class TelegramDescriptor which has a Telegram pointer as a field. To realize multiplexing we have the MultiplexingContext which describes the context and uses the MultiplexingManager which needs both TelegramDescriptor and RollingIndexManager (this is used to realize partial multiplexing).

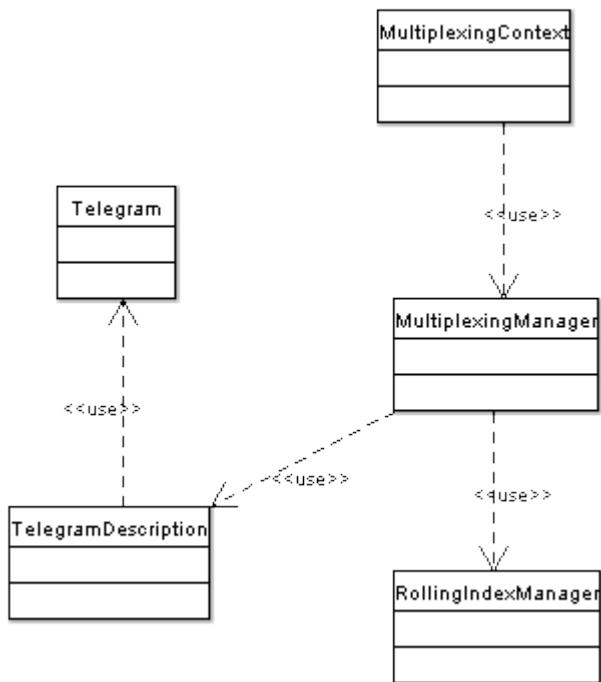


Figure 8.20, Synchronization classes.

All the classes of this package are general and common for all the equipment models.

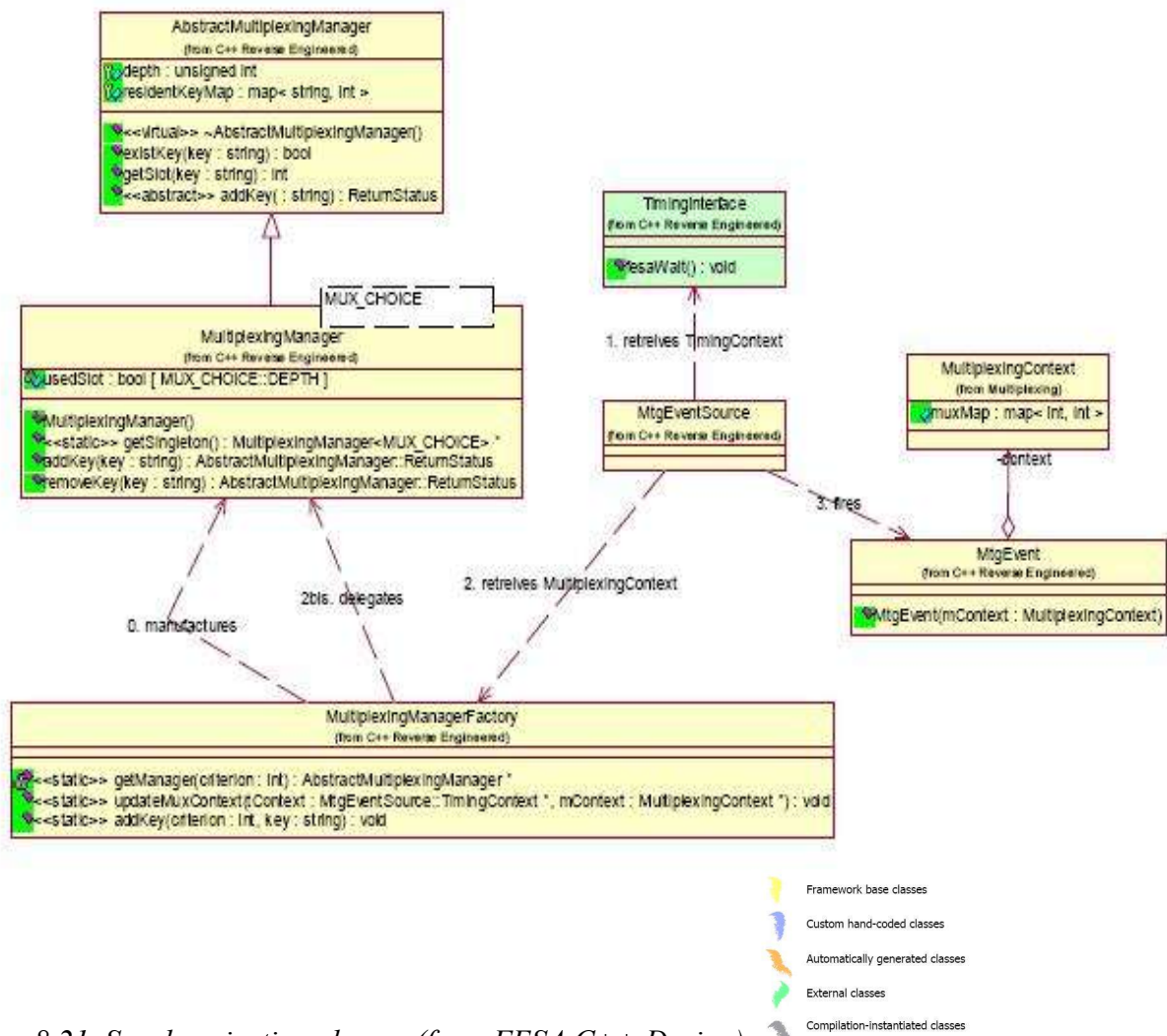


Figure 8.21, Synchronization classes (from FESA C++ Design).

8.2.8 Exception

This package includes all the exceptions that can be thrown by FESA methods, we have:

- *FesaBadConfig* (this kind of exception is caused by wrong configuration),
- *FesaIoError* (this kind of exception is caused by errors in reading or writing the hardware memory),
- *FesaBadParameter* (this kind of exception is caused by wrong parameters given to FESA methods),
- *FesaTypeMismatch* (this kind of exception is caused by errors of variable type),
- *FesaInvalidResource* (all fault fields - see par. 7.2.5- can invalidate some properties and if a client tries to use one of these properties, a *FesaInvalidResource* exception is thrown),
- *FesaInternalError* (this kind of exception is caused by internal errors).

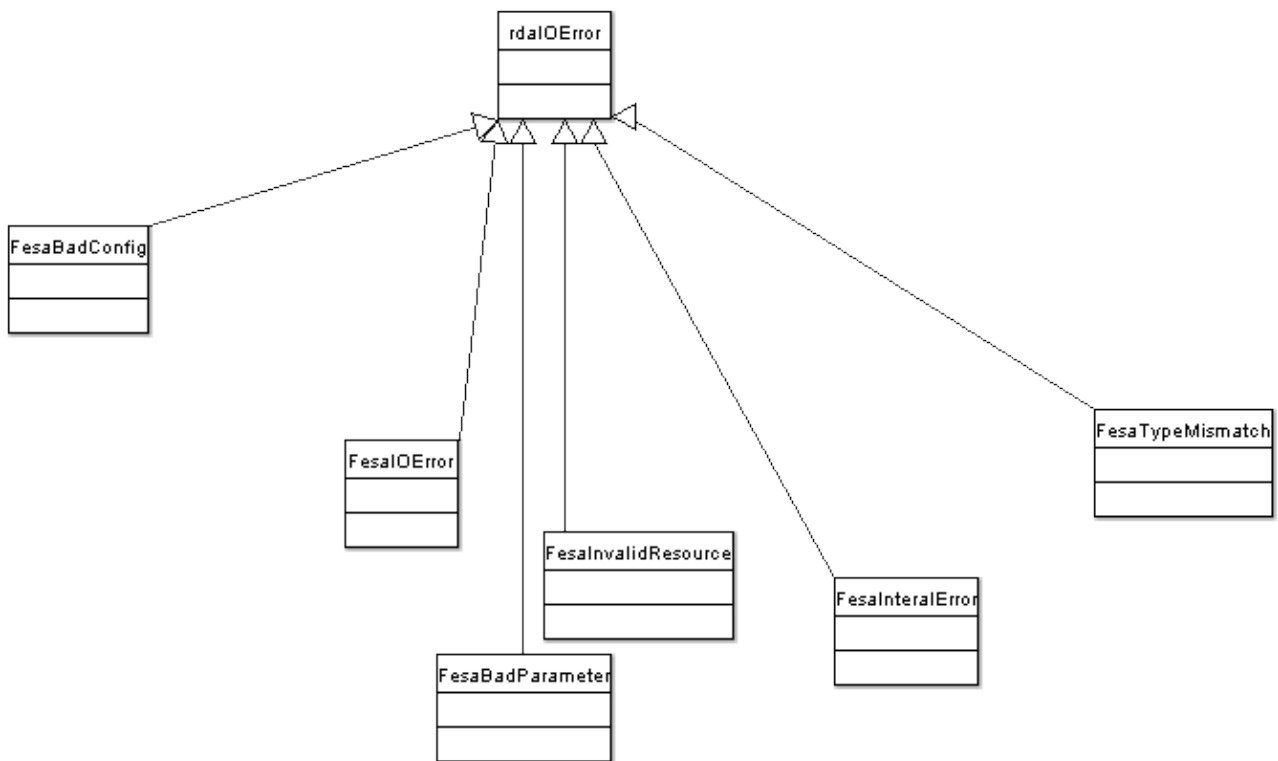


Figure 8.22, Exception classes.

8.2.9 Utility

In this package we have some class that are used in the framework for xml parsing and to work with strings, I will not enter into details.

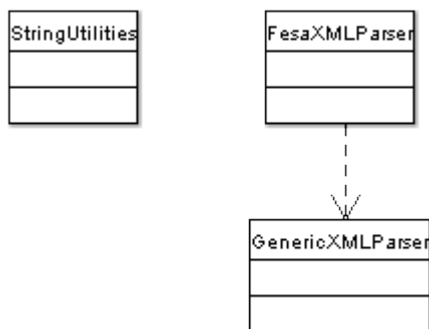


Figure 8.23, Synchronization classes.

8.2.10 Logging

For an equipment-software, the purpose of logging is to notify «interested parties» that something noteworthy, unusual, or abnormal occurred within the course of execution. To this aim, FESA defines an API and mechanism for developers to log run-time information with a range or severity levels that conform to the log4j standard. Logging support takes the form of a log class derived by C++ style string-streams. This makes logging similar to writing messages to the standard output, apart that it also requires setting a severity level (debug, info, warning, error or fatal). The TraceLogger and the TraceAction classes are defined in this package: the first class extends the

ostream class adding facilities to manage the logging and the second one extends the AbstractServerAction class to trace also information about the actions which are executed.

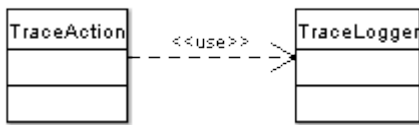


Figure 8.24, Logging classes.

8.2.11 Recorder

Recorder and History are the two classes of this package. The aim of this package is to store information every time a field is updated by a set operation. With all this information (History) it is possible to know what has changed and send to a user, who subscribed a property, only the minimum amount of information needed (see Notification in par 8.2.2). To obtain that, the Recorder and History classes are used.

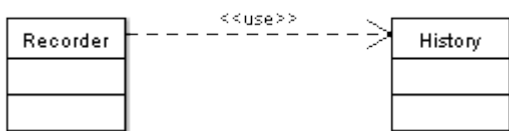


Figure 8.25, Record classes.

8.2.12 Sorting

In this package there are all the classes related to the sorting operations and they are used into the scheduling unit to specify rules to select some devices in the pool of all devices instances.

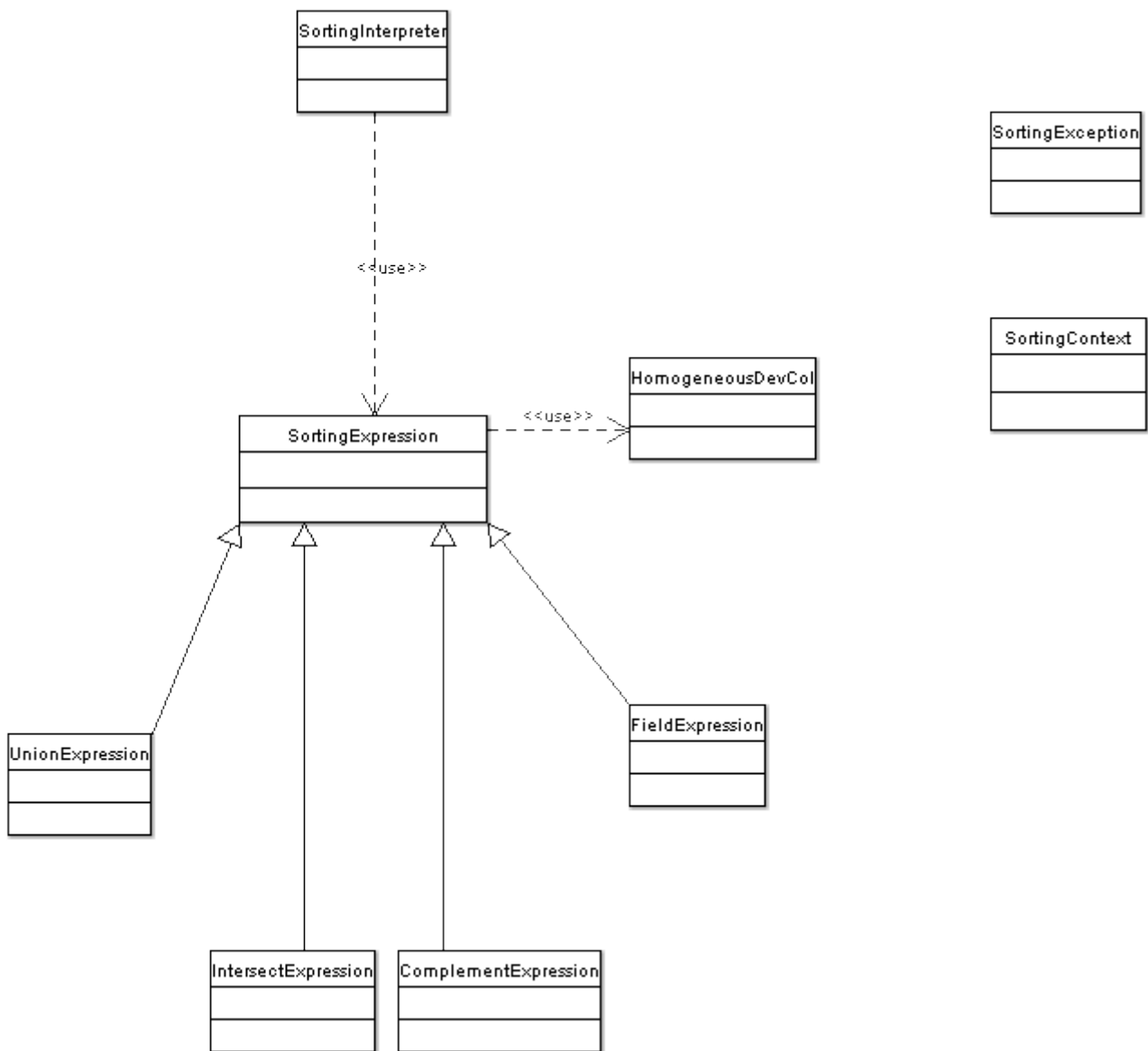


Figure 8.26, *Sorting classes.*

First of all we have the `SortingInterpreter` which use `SortingExpression`. `SortingExpression` uses the `HomogeneousSetCollection` which is the collection that has to be sorted and it could be implemented as:

- `UnionExpression`
- `IntersectExpression`
- `ComplementExpression`
- `FieldExpression`

Also we have other two classes that provide utilities: one is `SortingContext` and the other one is `SortingException`.

8.2.13 PLC

PlcCommunication is the main class that is used to make possible the communication with the plc (see par. 4.3.2) used by the device. This class uses the PlcConnection which handles all the operations concerning the use of the sockets and PlcCommunicationError class for handling all the errors that can occur. We have also the PlcDefs and the PlcType which are .h files with define and include directives for the plc classes.

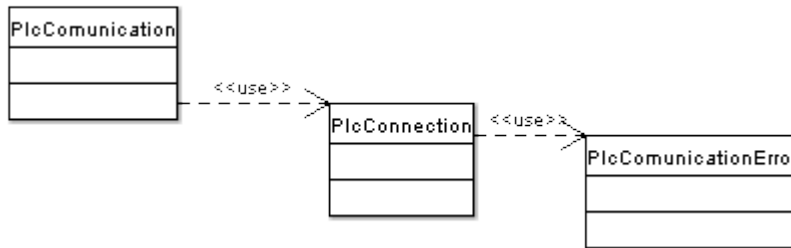


Figure 8.27, PLC classes.

8.3 Steps to create a FESA Equipment module

There are some steps that make possible the creation of an equipment module using FESA framework and these steps are supported by different FESA tools. Let us see the different steps and the tools that can help us.

8.3.1 Design

The first step is the Design and for that we have a special tool (Design tool) which make possible to realize an unambiguous description of the software module that we want to realize.

To obtain that we have to:

- define the internal representation of the device through the different fields needed (*Data*);
- define the public interface with all the services which can be remotely accessed through the middleware (*Properties*) and the *Alarms* that can be raised by the module;
- define the enumerates, bit-maps and constant values that may be used (*Custom- types*);
- define the different actions, *Server-actions* or *Rt-actions*, that can be invoked by properties or triggered by real-time events separately;
- define the real-time events (*Event*) that can happen and to which Rt-action they are related (*Scheduling*);
- declare the different time domains under which the equipment can work (*Target-Timing-Domain*);
- declare the standard services (*std-service*) used, such as the PLC;

The following figure is a snapshot of the design tool graphical interface that we used to define and declare the different parts of the software equipment.

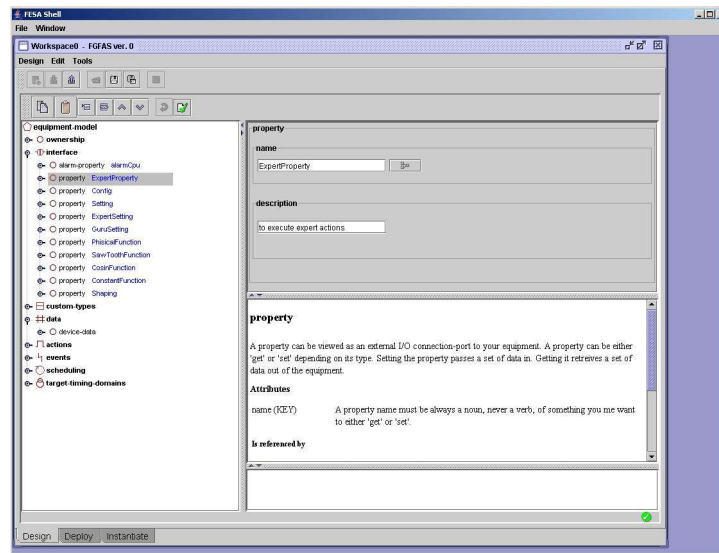


Figure 8.28, Fesa Design tool.

The design phase produces an xml document to describe the module and from this description, thanks to some Fesa Unix, it utilities is possible to generate the “frame” of our equipment module, i.e. all the classes needed to describe the equipment.

8.3.2 Code

At this step the programmer has to implement all the real time actions and server actions declared in the previous step, implementing the different execute() methods of the different actions. He can need to define some more classes, to encapsulate some utilities, but it is not mandatory.

8.3.3 Deliver

When the code is considered correct it can be delivered: all the code is saved in the common CVS repository and all the binaries are delivered for a particular platform. That makes it possible to allow subsequent deployment on one of several front-end computers.

8.3.4 Deploy

Before being able to create new instances of the equipment class implemented, we need to prepare one or several front-end computers that can host the class. That is the aim of the Deploying step through the Deployment tool. We have to choose a particular FEC and then define the deployment options chosen for our class.

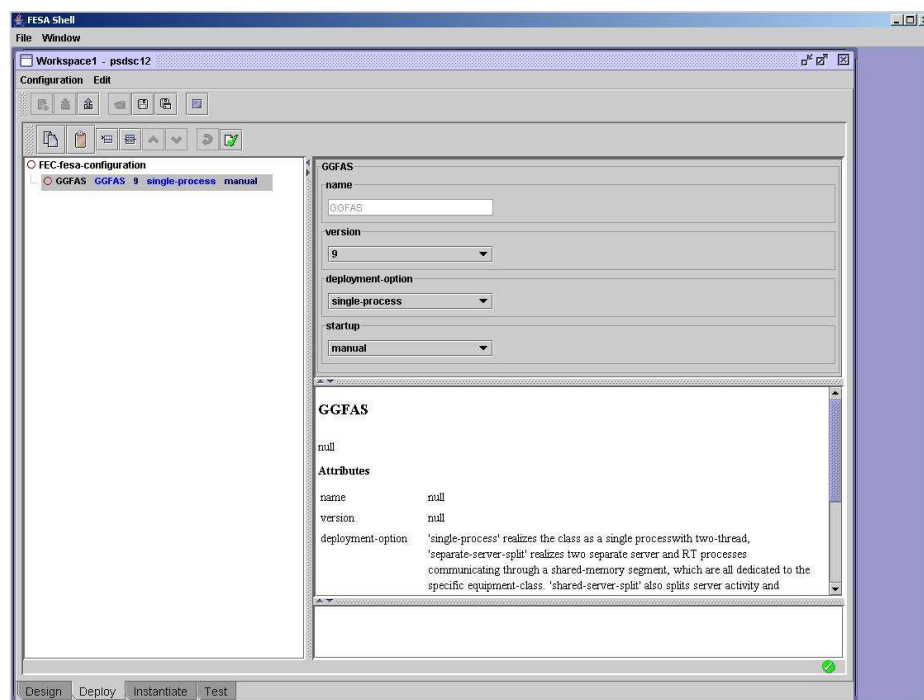


Figure 8.29, FESA Deployment tool.

First of all we have to choose how the different instances of our class are handled in the FEC, we have the following choices:

- *Single-process*, we have a single process with two threads (one for the real-time and the other for the server part).

- *Separate-server-split*, Server and RT processes are separated and they communicate through a shared-memory segment, which is all dedicated to the specific equipment class.
- *Shared-server-split*, the server and real-time activities are split in two processes relying on shared memory, but the server-part is merged with other equipment-classes within a shared server process.
- *Shared-server-unsplit*, both the server and the real-time parts of an equipment-class are gathered into a process which is shared with other equipment-classes. This mode of deployment should be avoided as much as possible, since a server process cannot accommodate more than one real-time task.
- *Shared-server-interface* if the equipment-class has no real-time part and a shared server process is used.
- *Separate-server-interface* if the equipment-class has no real-time part and a standalone process is used.

As in figure 8.30 we summarize what we have said till now.

At the deployment step we have also to choose if, at the start-up or reboot of the FEC, we want that the instances of our class are automatically launched or not.

After that we can *instantiate* our class.

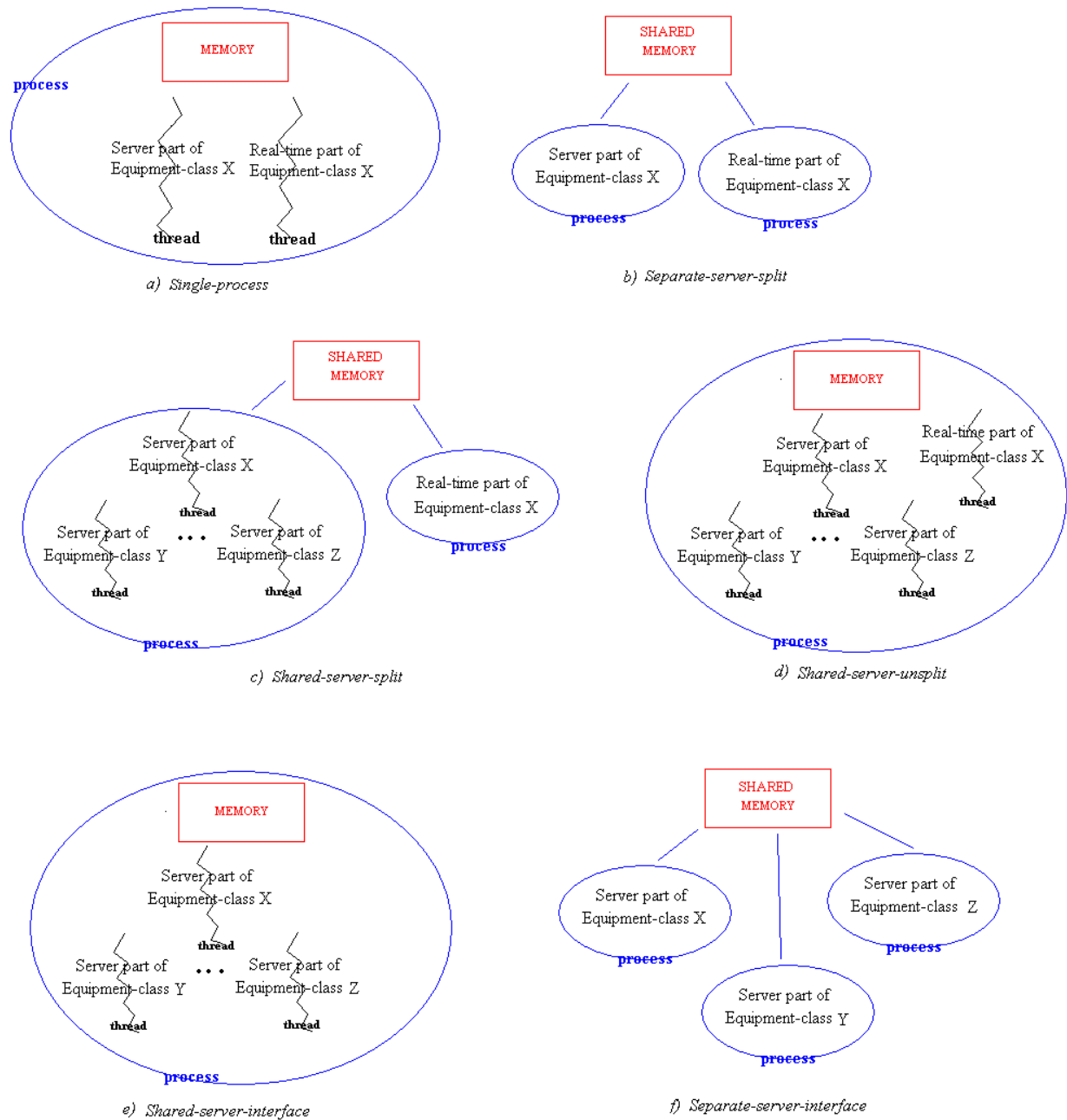


Figure 8.30, Deployment choices.

8.3.5 Instantiation

In this step we can create instances of our class and in general we configure a new instance of a class each time we connect a new hardware device to the front-end computer (through a dedicated electronic board or through a fieldbus).

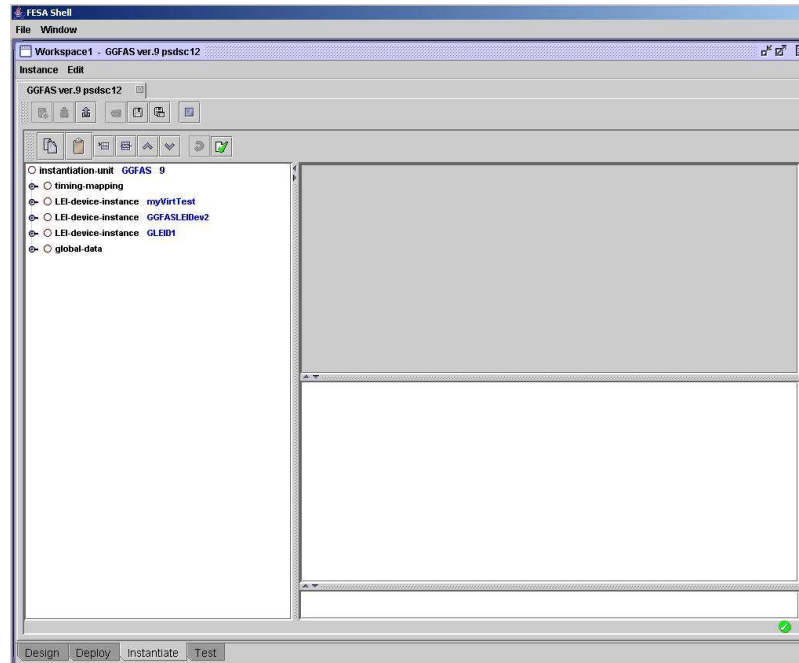


Figure 8.31, FESA Instantiation tool.

At the Instantiation step, using the Instantiation tool, we have to provide the following information that is needed to configure it:

- The name of the instance,
- A description,
- The name of the accelerator that is going to use it,
- The timing domain used,
- If we are using or not the multiplexing features,
- The real events used for all the logical events used,
- The hardware address (the type, the slot used in the FEC and the channel),
- The values of all the FINAL fields used by the equipment.

8.3.6 Test

After the Instantiation step we have the executable file for the particular front-end, then it is possible to execute the code, all the real time actions will be controlled by the real time activity and the server actions can be triggered by the users. To check that the code is correct it is possible to use a graphical tool (Navigator) to trigger all the different server actions manually. The Navigator is a generic application, driven by the automatically generated Java stubs which allow access to all properties in all declared device instances of the equipment class. It supports access to compound data types as well as atomic command response. The framework also provides system-level services to monitor the processes and follows their activity without degrading the real time performance of the front end processes. A front-end process activity tool allows the remote surveillance of any

FESA process, and allows the run-time control of parameters like the verbosity level and the logging history depth. [17]

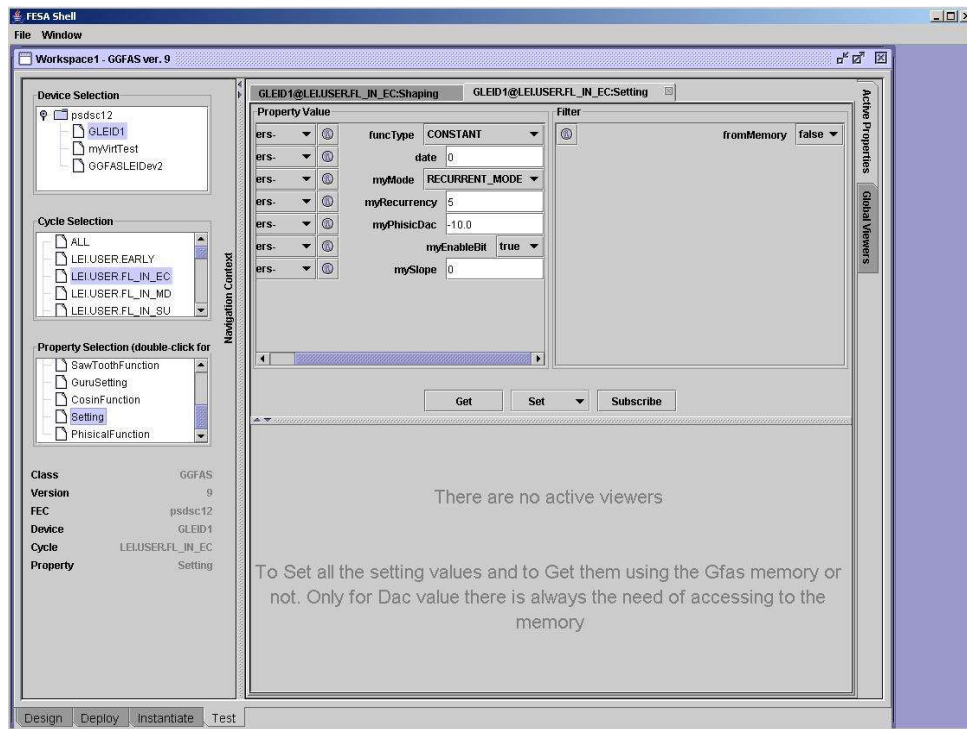


Figure 8.32, FESA Navigator.

Chapter 9

Software Specification

Now we can introduce the software requirements for the GFAS module.

To design the module I had to take into account the previous version of GFAS, the one realized using the old framework (GM). The hard part of the work was to understand all the functionalities of the module mainly just reading the code.

The previous version had a totally different structure, with little concern for modularity and code reuse. For FESA these two features of the code have a main role and in developing the GFAS FESA class I consider them as main issues.

We can identify two main parts:

- 1) Server part,
- 2) Real time part.

This separation was also in the GM version because there were two files: one with all the code to manage the real time task and an other with all the server operations available for the software module.

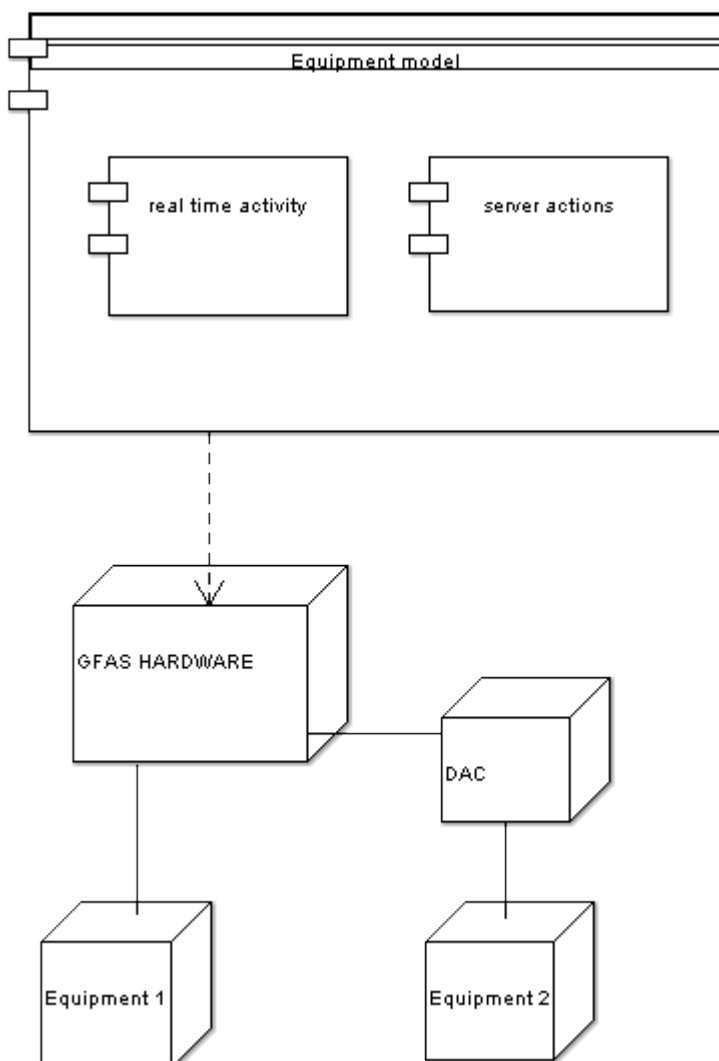


Figure 9.1, General architecture.

9.1 Server part

In the realization of the Server part we have many innovations in the design, thanks to the new FESA facilities.

In the previous GFAS version, the one realized with the GM framework, the properties were around forty because they were atomic and they could just retrieve the information concerning one field. In FESA is possible to get or set a group of them, which makes the interface much more compact and easier to manage.

As we will see I decided to use only ten properties, organized in three different groups depending on the type of user: common user, expert user or guru user (see Ch 6).

We do not have any check on the kind of user who triggers a property, the grouping represents just a guideline to let him find easily properties that he could be interested in.

Now I will list the properties that I chose for the GFAS design and the criterias according to which I grouped the fields associated to each of them.

1.1) Common user properties:

- 1.1.1) Configuration (to retrieve all the read only fields that are used in the GFAS),
- 1.1.2) Setting (with all the volatile parameters that are used to control the GFAS) ,
- 1.1.3) Physical function (with all the utilities to work directly with the physical representation),
- 1.1.4) Saw-tooth function (with all the utilities to work with the saw-tooth functions),
- 1.1.5) Cosine function (with all the utilities to work with the cosine functions),
- 1.1.6) Constant function (with all the utilities to work with the constant functions),
- 1.1.7) Shaping (with the utilities to modify the reference value and the delay of a function).

1.2) Expert user properties:

- 1.2.1) Expert property (to trigger some action that only an Expert user is interested in),
- 1.2.2) Expert setting (to set and get volatile information to control the GFAS working with the hardware representation).

1.3) Guru properties

- 1.3.1) Guru setting (to get and set all the volatile information to control the GFAS that needs a complete knowledge of the organization of GFAS memory).

Let's describe the server actions related to the properties introduced.

1.1) Common user properties:

1.1.1) Configuration

a simple get-action used to retrieve from the internal representation:

- type of connection,
- minimum physical value accepted,
- maximum physical value accepted,
- scaling factor decided,
- shaping code,

- name of the master for the double/triple PPM,
- extra-condition for the double/triple PPM,
- the virtual number.

1.1.2) Setting

- i) a get-action to retrieve (from the GFAS memory or from the internal representation):
 - the type of the function,
 - the GFAS mode,
 - the date when the last function has been loaded,
 - recurrence value,
 - the physical value of the DAC,
 - the value of the slope,
 - the enable bit.
- ii) a set-action to set (writing also the GFAS memory):
 - the GFAS mode,
 - the recurrence value,
 - the physical value of DAC,
 - the value of the slope,
 - the enable bit to enable or disable each function.

1.1.3) Physical function

- i) a get-action to retrieve (from the GFAS memory or from the local representation):
 - the physical representation of the function,
- ii) a set-action to set (writing also in the GFAS memory)
 - the physical representation of the function.

1.1.4) Saw-tooth function

- i) a get-action to retrieve (from the local representation):
 - the physical value of the gain of the saw-tooth function loaded previously,
 - the frequency of the saw-tooth function loaded previously.
- ii) a set action to load a saw tooth function (saving it into GFAS memory) through:
 - the physical value of the gain,
 - the frequency.

1.1.5) Cosine function

- i) a get-action to retrieve (from the local representation):
 - the physical value of the gain of the cosine function loaded previously,
 - the frequency of the cosine function loaded previously.
- ii) a set action to load a saw tooth function (saving it into GFAS memory) through:
 - the physical value of the gain,
 - the frequency.

1.1.6) Constant function

- i) a get-action to retrieve (from the local representation):
 - the physical value of the gain of the constant function loaded previously.
- ii) a set action to load a constant function (saving it into GFAS memory) through:
 - the physical value of the gain.

1.1.7) Shaping

- i) a get-action to retrieve (if it's possible according to the shaping code) from internal representation :
 - the ref value,
 - the delay.
- ii) a set-action to modify (if it's possible according to the shaping code) without saving it into the GFAS memory:

- the ref value,
- the delay.

1.2) Expert user properties:

1.2.1) Expert property

a set-action to trigger one of the following operations:

- reset the GFAS memory,
- induce conversion of the function loaded,
- trigger a software event start,
- trigger the main software start,
- copy the current function in the part of the memory specified by a function index,
- select a function as function to be generated.

1.2.2) Expert setting

- i) a get-action to retrieve (from the GFAS memory or from the internal representation):
 - the length of the function considered,
 - the hardware representation of the function,
 - the digital value of the DAC,
 - the status word.
- ii) a set-action to load (saving it also into the GFAS memory):
 - the function using the hardware representation.

1.3) Guru properties

1.3.1) Guru setting

- i) a get-action to retrieve from GFAS memory:
 - the conversion flag (0 if neither the force conversion or the conversion flag are set, 1 if the conversion flag is set and 2 if both flags are set),
 - the index of the function that is going to be generated,
 - the indices of the functions accepted,
 - the indices of the functions that have to be converted,
 - the indices of the disabled functions.
- ii) a set-action to set into GFAS memory:
 - the conversion flag (0 if neither the force conversion or the conversion flag are set, 1 if the conversion flag is set and 2 if both flags are set),
 - the index of the function that is going to be generated.

If we are using a multiplexed context we have up to 32 functions (and not just one), then we have to distinguish between operations that are made on the GFAS and the operations concerning just one function, the one of the particular cycle selected.

The properties that change depending on the cycle are:

- 1.1.2) Setting: each function has its own enable bit,
- 1.1.3) Physical function : we work with the representation of a function associated to a particular cycle,
- 1.1.4) Saw-tooth function,
- 1.1.5) Cosine function,
- 1.1.6) Constant function,
- 1.1.7) Shaping function,
- 1.2.1) Expert properties (except resetting of GFAS all the operations concern the particular function of a cycle),
- 1.2.2) Expert setting (except getting the digital value of DAC and the status word, all the operations concern the particular function of a cycle),

Before describing the real time part let's analyse better the virtual GFAS.

We have a Real GFAS (virtual number equals -1) and up to 5 Virtual GFAS (virtual number equals the number of the associated Real one).

The Real GFAS behaves in the same way as a common one but, for virtual GFAS, it is not possible to use all the properties shown but just a part of them and also they behave in a different way.

To summarize, the only operations allowed are

1) in the Setting property:

- get the type of the function,
- get the date when the current function has been loaded,
- get and set the enable bit but its value is the one of the internal representation and not the one from the memory.

2) in the Physical function property:

- get physical representation of the function only from the local representation,
- set the physical representation of the function which is used together with all the enabled functions of all other virtual GFAS to compute the composite function. The result is loaded in the hardware memory of the real GFAS and the conversion into the internal representation is induced.

3) in the Expert setting property:

- get the length of the function considered,
- get the hardware representation of the function from the local representation,
- set the hardware representation of the function and to compute the composite one, using it together with all the enabled function of all other virtual GFAS. The result is loaded in the hardware memory of the real GFAS.

9.2 Real time part

Concerning the real-time part, FESA separates completely all the operations that are common to all the equipment and the actions that are related to the particular module. In the GM version of GFAS this separation was not present.

Analysing the code I find two different actions: one that makes it possible to acquire information about the GFAS behavior and another one to control the generation of functions.

Then we have two different logical events that can trigger two different actions:

2.1) Acquisition event, associated with the Acquisition action to get

- The name of the machine,
- The user,
- The index of the function generated,
- The status word of the GFAS.

2.2) Control event associated to the Control action used to:

- induce the generation of the right function; if double/triple PPM is used, further action is needed (see the next paragraph) .

Let's analyse better point 2.2.

For each channel we can have up to three different instances of GFAS one of them is called the master and the other two the slaves.

Then there are three different situations, for each channel we can have:

- just an instance of the device, this is the basic behaviour,
- a master and a slave, we are using the double PPM feature,
- a master and two slaves, we are using the triple PPM feature.

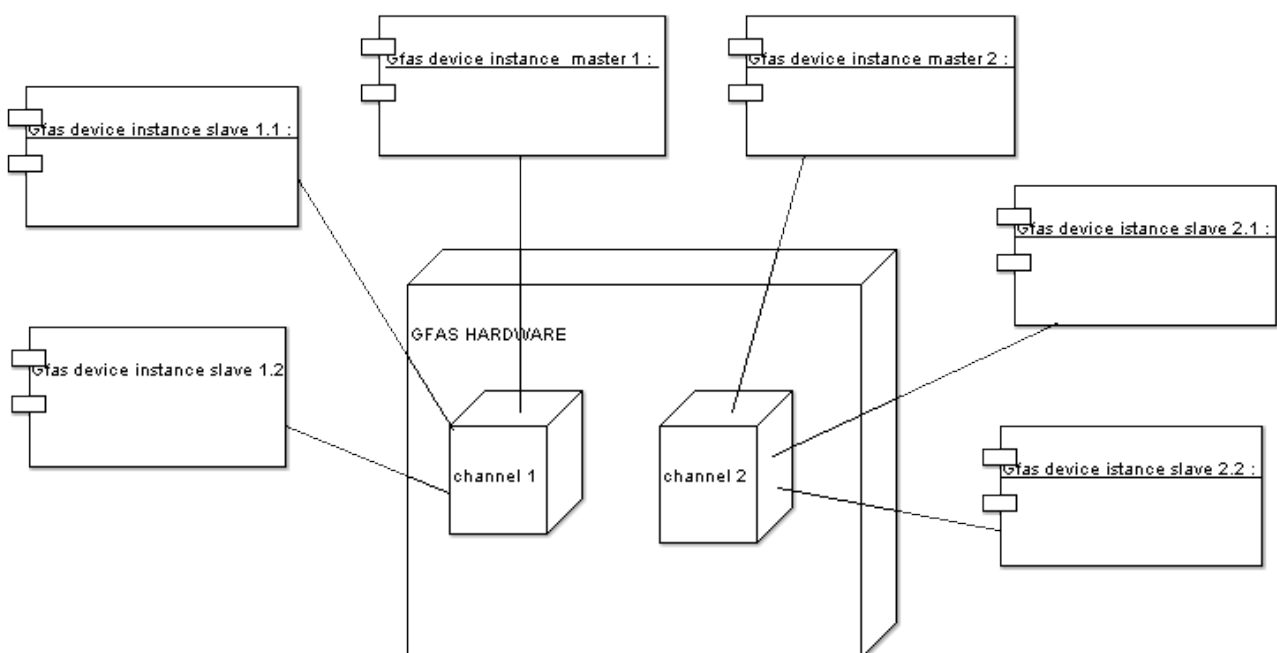


Figure 9.2, Master and slave device instances for triple PPM .

Let's analyse each scenario.

a) basic behaviour

- The cycles and the associated index are retrieved from the telegram,
- We take one of the 32 functions according to the convention that we take the one with the index equal to cycle index modulo 32,
- We generate that function if it exists and if it is enabled.

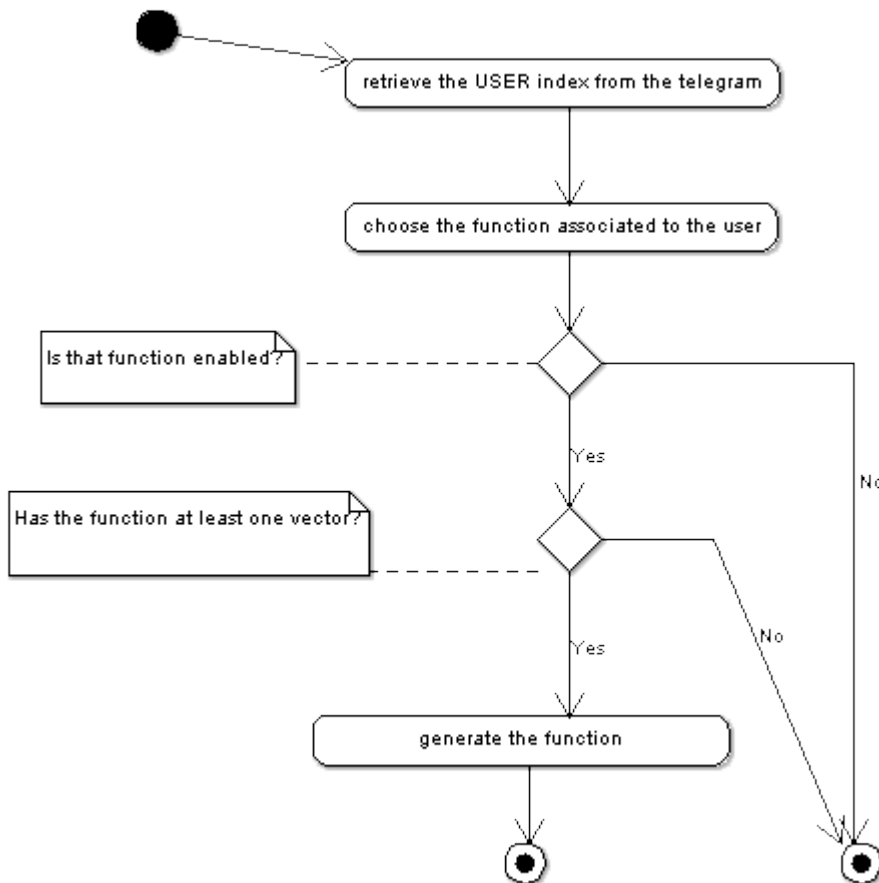


Figure 9.3, Basic behaviour.

b-c) double or triple PPM use

i) the instance is the master one:

- cycles and the associated index are retrieved from the telegram,
- we take one of the 32 functions according the convention that we take the one with the index equal to cycle index % 32,
- we generate that function if it exist and if it is enabled.

ii) the instance is the slave or one of the two slaves:

- cycles and the associated index are retrieved from the telegram,
- we take one of the 32 function according to the convention that we take the one with the index equal to USER' s index modulo 32,

- we analyse the extra-condition which is made of the quadruple <machine name, group name, value 1, value 2>. If the machine name in the telegram matches the one in the extra-condition and the group field in the telegram has one of the two values specified in the extra-condition, then we consider the extra condition verified,
- if the extra-condition is not verified, we do nothing, otherwise if it is verified we load the function into the master memory and we force the conversion according to the internal representation. Then we generate that function if it exists and if it is enabled.

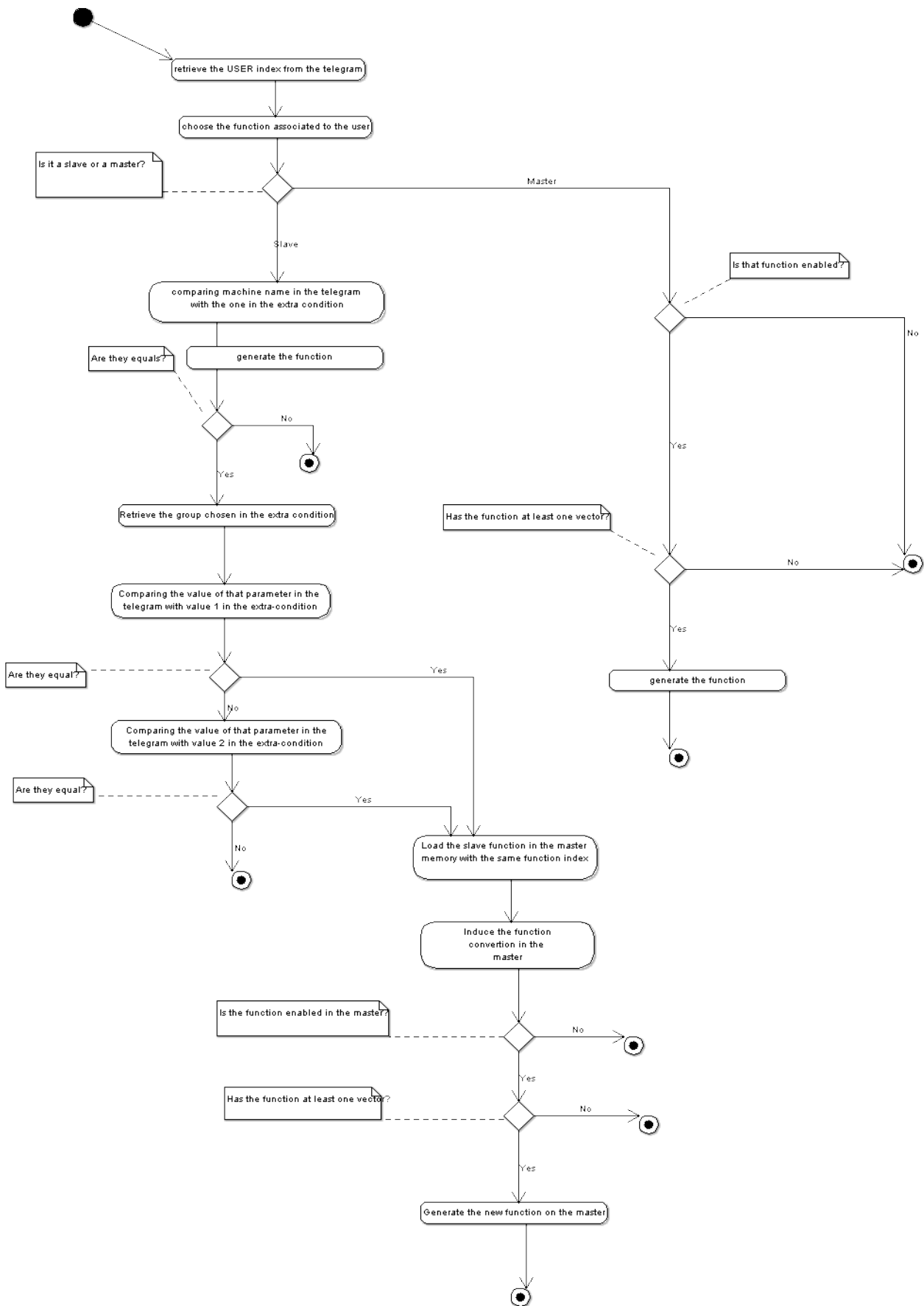


Figure 9.4, Double/triple PPM behaviour.

Chapter 10

Code design

Now I describe the representation of the GFAS using FESA conventions. After this first introduction, we will see the description of the classes that have been designed and their relationship with FESA framework classes (see Ch. 8).

10.1 FESA design

Taking into account the different server and real time actions we decided to use the following fields for the abstraction of the GFAS in the FESA equipment module.

1) DATA – device data:

1.1) We have some FINAL fields:

- *hwIType, hwILun, hwICh* : the hardware address (type, lun, channel),
- *physicalUnit*: physical unit used for the output,
- *connection* : type of connection of the GFAS and the equipment,
- *shapingCode*: the shaping code to specify the manipulating operation (see par. 5.2.6),
- *min* : minimum allowed physical value of the output,
- *max* : maximum allowed physical value of the output,
- *scalingFactor* : the scaling factor,
- *mySinSlave* : the sine slave name (0 if it doesn't exist),
- *myMuxMaster* : Master name for double or triple PPM (0 if it doesn't exist),
- *otherMuxCondition* : Extra-condition for double or triple PPM (0 if it doesn't exist).

1.2) We have some VOLATILE fields which are NOT multiplexed:

- *cpuFault*: fault field used by alarm,
- *disabled* : disabled word (with the information of function enabled or not),
- *gfasModeWord* : GFAS mode,
- *recurrence* : Recurrent value (for recurrent mode),
- *slope* : Slope value.

1.3) We have some VOLATILE fields which are multiplexed by USER:

- *funcType* : type of the function,
- *hwFun* : the hardware representation of the function,
- *gain* : the gain (it is meaningful just for cosine, saw-tooth or constant function types),
- *frequency* : the frequency (it is meaningful just for cosine and saw-tooth function types),
- *date* : the date of when the function has been loaded.

1.4) We have also some interrupt-fields to trigger the logical events:

- *controlIT* : a controlling interrupt,
- *acquisitionIT* : an acquisition interrupt.

In the current version of FESA is not possible to have array and array2D of custom types and, as we said in chapter 8, we can have only long, short, long long, float, double, char and signed char. Then we cannot use unsigned except for constant unsigned int that are defined as one of the possible custom types.

Because of that I had to use long for hwFun (hardware function table) . In the GM version the hardware representation of the function was made using unsigned short int but this was not precise because in the hardware representation we can have also -1 (a negative value) when an Internal stop is present that could produce ambiguity between the maximum value 0xFFFF and -1. I used long to have the same range of positive values of an unsigned short and a non ambiguous representation for -1.

For the hwFun I used an array2D to use the same structure, in the GM version it was not possible to use it because array were not supplied by the framework.

That is a table with the list of all the fields chosen for the device and for each field we have the multiplexing criterion used, the persistency, its type and dimension (see par. 8.2.5).

Field	Mux-criterion	Persistency	type	dim
connection	NONE	FINAL	CONNECTION_TYPE	
max	NONE	FINAL	Double	Scalar
min	NONE	FINAL	Double	Scalar
myMuxSlave	NONE	FINAL	Char	Array(30)
mySinSlave	NONE	FINAL	Char	Array(30)
otherMuxCondition	NONE	FINAL	Char	Array2D(4,30)
physicalUnit	NONE	FINAL	PHISIC_UNIT	
scalingFactor	NONE	FINAL	Double	Scalar
shapingCode	NONE	FINAL	SHAPING_CODE	
virtualNumber	NONE	FINAL	Char	Array(30)
acquisitionIT	NONE	VOLATILE	Short	Scalar
controlIT	NONE	VOLATILE	Short	Scalar
cpuFault	NONE	VOLATILE	Bool	Scalar
date	USER	VOLATILE	Long	Scalar
disabled	NONE	VOLATILE	Long	Scalar
frequency	USER	VOLATILE	Double	Scalar
funcType	USER	VOLATILE	FUN_TYPE	
gain	USER	VOLATILE	Double	Scalar
gfasModeWord	NONE	VOLATILE	GFAS_MODE	
hw1Ch	NONE	VOLATILE	Char	Array(60)
hw1Lun	NONE	VOLATILE	Char	Array(60)
hw1Type	NONE	VOLATILE	Char	Array(60)
hwFun	USER	VOLATILE	Short	Array2D(512,3)
recurrency	NONE	VOLATILE	Long	Scalar
slope	NONE	VOLATILE	Short	Scalar

2) CUSTOM-TYPES

2.1) enum:

- connection_type,
- expert_action,
- fun_type,
- gfas_mode,
- physic_unit,
- shaping_code.

2.2) bit_enum_32:

- fun_bit_map.

2.3) bit_enum_16 :

- status_bit_map.

2.4) constant:

- hw_max_len,
- hw_vect_dim,
- max_fun_number,
- physic_fun_len.

In the following table we have a more detailed description of the enumerates and constant declared with their values.

Name	Fesa-Type	Value
CONNECTION_TYPE	ENUM	U_DIGITAL=3 B_DIGITAL=2 U_DAC=1 B_DAC=0
EXPERT_ACTION	ENUM	SETV=5 SELECT=4 COPYPL=3 INITL=2 SW_EVENT=1 SW_START=0
		b0=fun0 b1=fun1 b2=fun2 b3=fun3 b4=fun4 b5=fun5 b6=fun6 b7=fun7 b8=fun8 b9=fun9 b10=fun10 b11=fun11 b12=fun12 b13=fun13 b14=fun14 b15=fun15 b16=fun16 b17=fun17 b18=fun18 b19=fun19 b20=fun20 b21=fun21 b22=fun22 b23=fun23 b24=fun24 b25=fun25 b26=fun26 b27=fun27 b28=fun28 b29=fun29 b30=fun30 b31=fun31
FUN_TYPE	ENUM	COMMON=3 CONSTANT=2 TOOTH=1 COS=0
GFAS_MODE	ENUM	NO_GENERATION=7 TEST2=6 TEST1=5 LOW_JITTER_RECURRENT=4 RECURRENT_MODE=3 LOW_JITTER=2 DAC_MODE=1 NORMAL_MODE=0
HW_MAX_LEN	CONST_UINT	512
HW_VECT_DIM	CONST_UINT	3
MAX_FUN_NUMBER	CONST_UINT	32
PHISIC_MAX_LEN	CONST_UINT	511

PHISIC_UNIT	ENUM	HZ=3 V=1 A=0
SHAPING_CODE	ENUM	NOTHING=7 DELAY=6 REF=5 REF_DELAY=4 EDIT=3 EDIT_DELAY=2 EDIT_REF=1 NORMAL_TREAT=0
STATUS_BIT_MAP	BIT_ENUM_16	b0=fun2_0 b1=fun2_1 b2=fun2_2 b3=fun2_3 b4=fun2_4 b5=cpuBusy b6=vmeReady b7=cpuRun b8=funAbort b9=funPaused b10=funInvalid b11=reccurMode b12=lowJitMode b13=dacMode b14=test1Mode b15=test2Mode

3) INTERFACE

3.1) simple properties:

- Config.

3.2) complex properties:

- ConstantFunction,
- CosinFunction,
- ExpertProperty,
- ExpertSetting,
- GuruSetting,
- PhisicalFunction,
- SawToothFunction,
- Setting,
- Shaping.

3.3) alarm properties:

- AlarmCpu.

Let us see the details through the following list of properties. For each of them we indicate:

- if it involves only get actions (RO) or if it involves both set and get actions (RW), or if it is a procedural only action (PO).
- if it is a simple or complex property (see par. 8.2.3),
- a list with all the in/out values that are used by the set and procedural or get actions respectively,
- a list of filters used by it (see par. 8.2.3).

Config RO SIMPLE		
connection max min myMuxMaster otherMuxCondition scalingFactor shapingCode	data-field-ref-item(connection) data-field-ref-item(max) data-field-ref-item(min) data-field-ref-item(myMuxMaster) data-field-ref-item(otherMuxCondition) data-field-ref-item(scalingFactor) data-field-ref-item(shapingCode)	CONNECTION_TYPE Double Double char/array(30) char/array2D(4,30) double/scalar SHAPING_CODE
ConstantFunction RW COMPLEX		
myGain	value-item	double/scalar
CosinFunction RW COMPLEX		
myFreq myGain mySlave	value-item value-item value-item	double/scalar double/scalar char/array(10)
ExpertProperty PO COMPLEX		
funToCopy typeOfAction	value-item filter-item	long/scalar EXPERT_ACTION
ExpertSetting RW COMPLEX		
fromMemory myDigitalDac myHwFunction myLenght myStatus	filter-item value-item value-item value-item value-item	bool/scalar long/scalar long/array2D(512,3) short/scalar STATUS_BIT_MAP
GuruSetting RW COMPLEX		
myAccepted myConversionFlag myDisabled myToBeConverted myToBeGenerated	value-item value-item value-item value-item value-item	FUN_BIT_MAP short/scalar FUN_BIT_MAP FUN_BIT_MAP FUN_BIT_MAP
PhisicalFunction RW COMPLEX		
fromMemory myPhisic	filter-item value-item	bool/scalar double/array(1023)
SawToothFunction RW COMPLEX		
myFreq myGain	value-item value-item	double/scalar double/scalar
Setting RW COMPLEX		
Date fromMemory funcType myEnableBit myMode myPhisicDac myRecurrency mySlope	data-field-ref-item(date) filter-item data-field-ref-item(funcType) value-item value-item value-item value-item value-item	long/scalar bool/scalar FUN_TYPE bool/scalar GFAS_MODE double/scalar short/scalar short/scalar
Shaping RW COMPLEX		
code myDelay myRef	filter-item value-item value-item	SHAPING_CODE double/scalar double/scalar

alarmCpu RO SIMPLE (Alarm)		
cpuFault	fault-field-ref-item(cpuFault)	bool/scalar

4) ACTIONS:

4.1) Server-actions

- getCosinFun (get for the property ConstantFunction),
- setCosinFun (set for the property ConstantFunction),
- getConstantFun (get for the property CosinFunction),
- setConstantFun (set for the property CosinFunction),
- expertAction (set for the property ExpertProperty),
- getExpertSetting (get for the property ExpertSetting),
- setExpertSetting (set for the property ExpertSetting),
- getGuru (get for the property GuruSetting),
- setGuru (set for the property GuruSetting),
- getPhysicalFun (get for the property PhysicalFunction),
- setPhysicalFun (set for the property PhysicalFunction),
- getSawToothFun (get for the property SawToothFunction),
- setSawToothFun (set for the property SawToothFunction),
- getSetting (get for the property Setting),
- setSetting (set for the property Setting),
- getFunShape (get for the property Shaping),
- setFunShape (set for the property Shaping).

4.2) RT-ACTIONS

- doContol,
- doAquisition.

5) EVENTS

5.1) logical-event-group

- ContrEvent (mtg),
- AcqEvent (mtg).

6) SCHEDULING

6.1) scheduling-unit :

- rt_action-ref = doAquisition,
- trigger = device-group-implicit-event-ref aquisitionIT.

6.2) scheduling-unit :

- rt_action-ref = doControl,
- trigger = device-group-implicit-event-ref controlIT.

7) TARGET-TIMING-DOMAIN

- Psb,
- Sps,
- Cps,
- Lei,
- None.

10.2 GFAS classes

We can distinguish the classes in four different groups:

- 1) GENERATED: all the classes generated automatically by the FESA tools,
- 2) SERVER: all the classes related to the server operations,
- 3) RT: all the classes related to the real-time operations,
- 4) COMMON: all the classes need by both the server and the real-time part.

Groups 2, 3 and 4 are the result of this thesis work.

10.2.1 GENERATED classes

We have many generated classes and the following figure shows them and the classes of the FESA framework which are implemented or extended by them. All the information given at the Design step (see par. 8.3.1) is enough to generate the following classes without any ambiguities.

In figure 10.1 we can see the class FGFASDevice that extends the class Device with all the data fields needed to describe the internal representation of the GFAS. We have also the methods that we can use to retrieve and/or modify the device collection which is a group of devices instantiated in a particular FEC. The method getStandardStatus makes it possible to retrieve the status of the device (It is not the “hardware status word”, which we have already talked about, but it is related to the software activities). The FGFASEquipmentDefaultInterface and the FGFASEquipmentDefaultRT, which are related to the Server and the real time activity respectively, are the instantiation of the Equipment Interface and Equipment RT. The parameters used are FGFASGlobalStore, FGFASDomainStore and FGFASDevice. Figure 10.2 shows all the classes related to the types needed to use the GFAS.

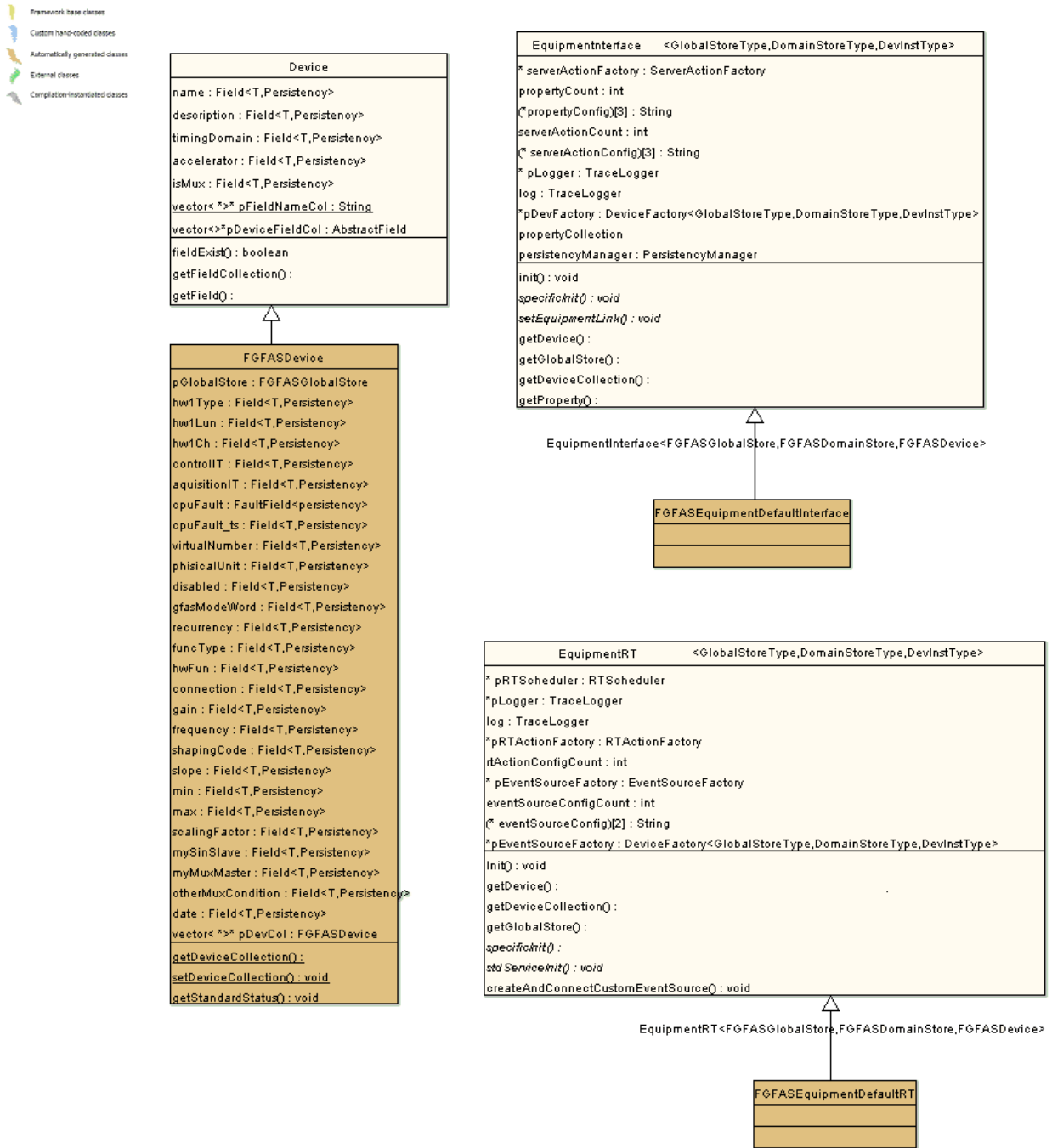


Figure 10.1, generated classes 1.

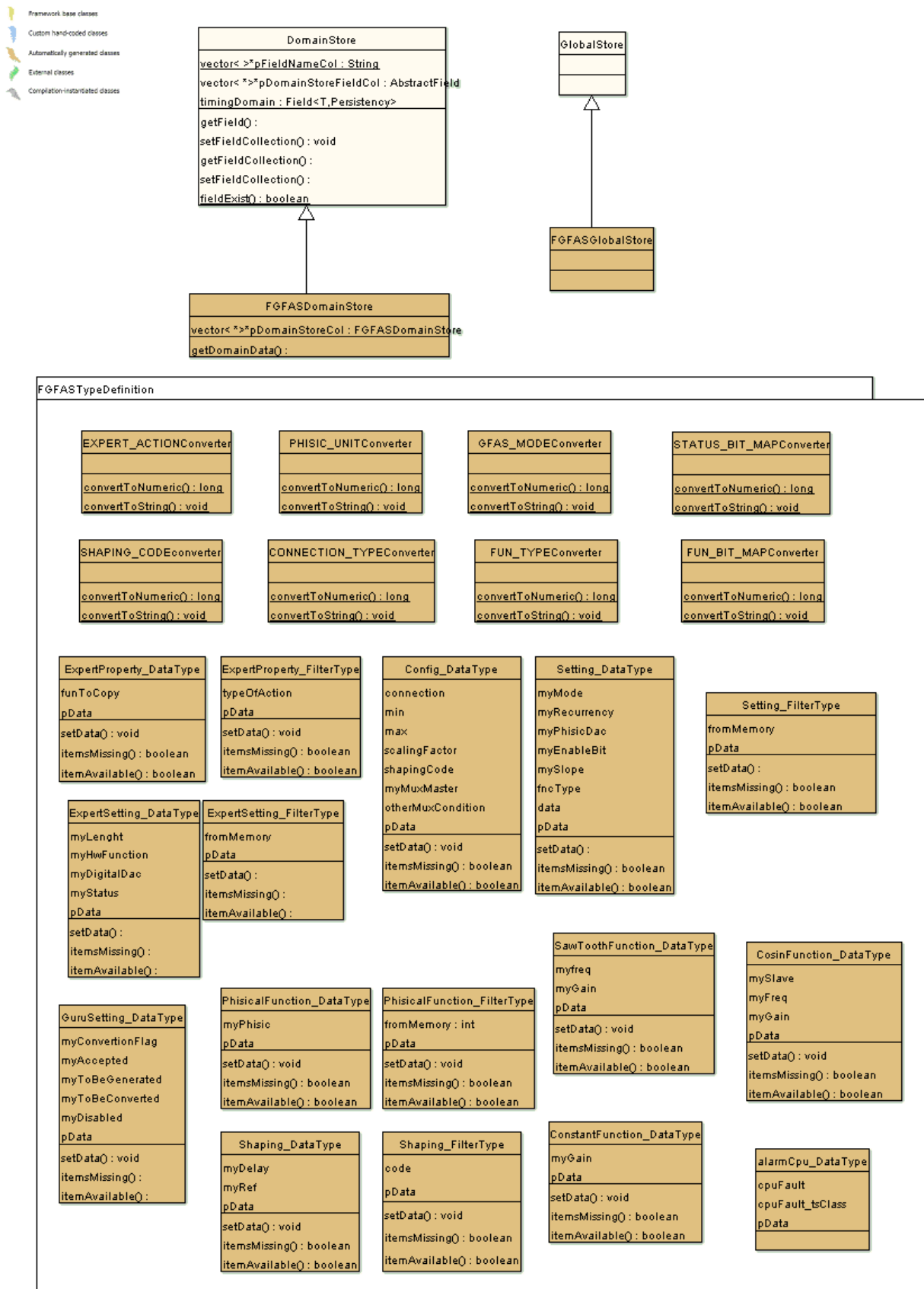


Figure 10.2, generated classes 2.

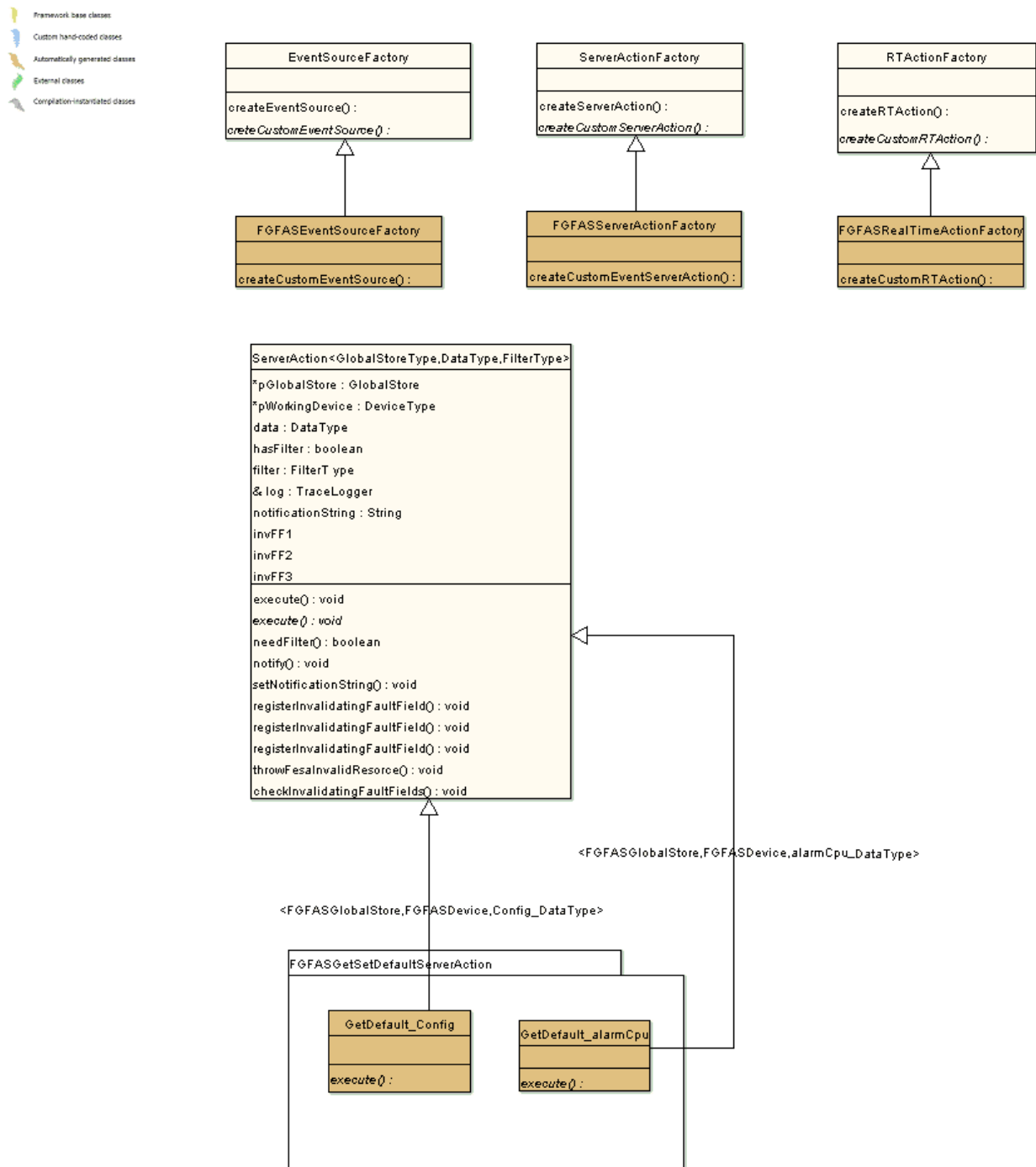


Figure 10.3, generated classes 3.

In Figure 10.3 we have the implementation of the EventSourceFactory, ServerActionFactory and RTActionFactory by FGFASEventSourceFactory, FGFASServerActionFactory and FGFASRTActionFactory. In all of them we have the implementation of the virtual method createCustomServerAction(), needed to generate events or actions for the GFAS.

This figure shows also two default server actions that are instantiated from the ServerAction template class:

- GetDefaultConfig, which uses as parameter `<FGFASGlobalStore,FGFASDomainStore, Config_DataType>`
- GetDefault_alarmCpu which uses as parameter `<FGFASGlobalStore,FGFASDomainStore,alarmCpu_DataType>`

These classes are generated and implemented automatically because in the design I defined them as SIMPLE get actions. Than there is no need to implementing them as the default implementation can be used.

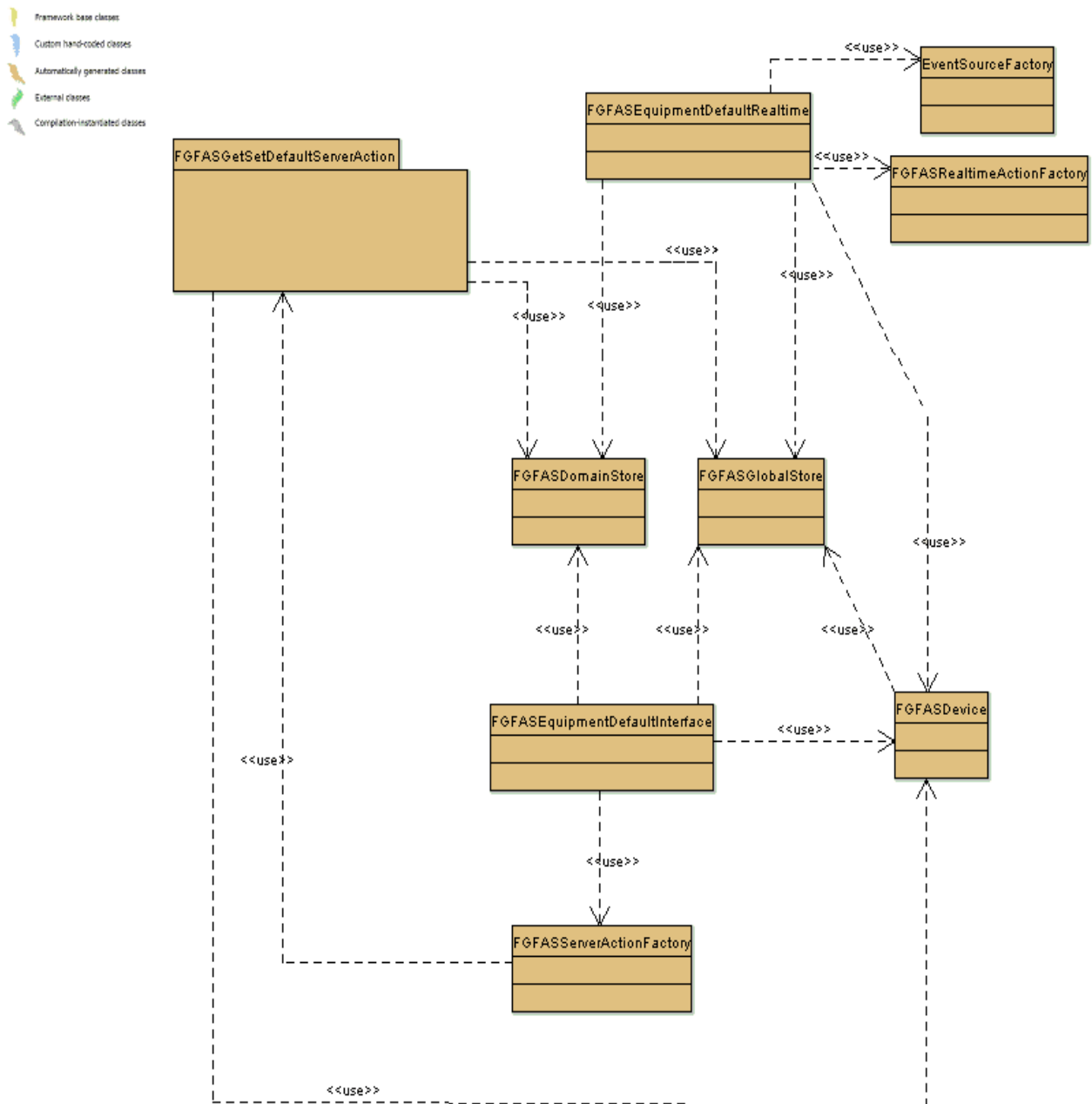


Figure 10.4, relationship between generated classes.

10.3.2 SERVER classes

Figure 10.5 shows all the Server-Actions (see par. 8.2.3) and the FGFAInterface, which extends the FGFAEquipmentDefaultInterface. This class has the method `init()` that we can use to initialise all the fields of the device and to do all the other operations needed when the server is started. For each server action we have an instance of the template class `ServerAction` and we use `FGFASGlobalStore`, `FGFASDomainStore`, `Data_Type` and `Filter_Type` (if a filter is used) as parameters. These last two parameters depends on the particular server action referred. For each class (coloured in blue in the next class diagram) I implemented the `execute()` method with the operation needed by the particular request. We can distinguish the actions that set some parameters of the device (figure 10.5) and the ones that get some values from the devices (figure 10.6).

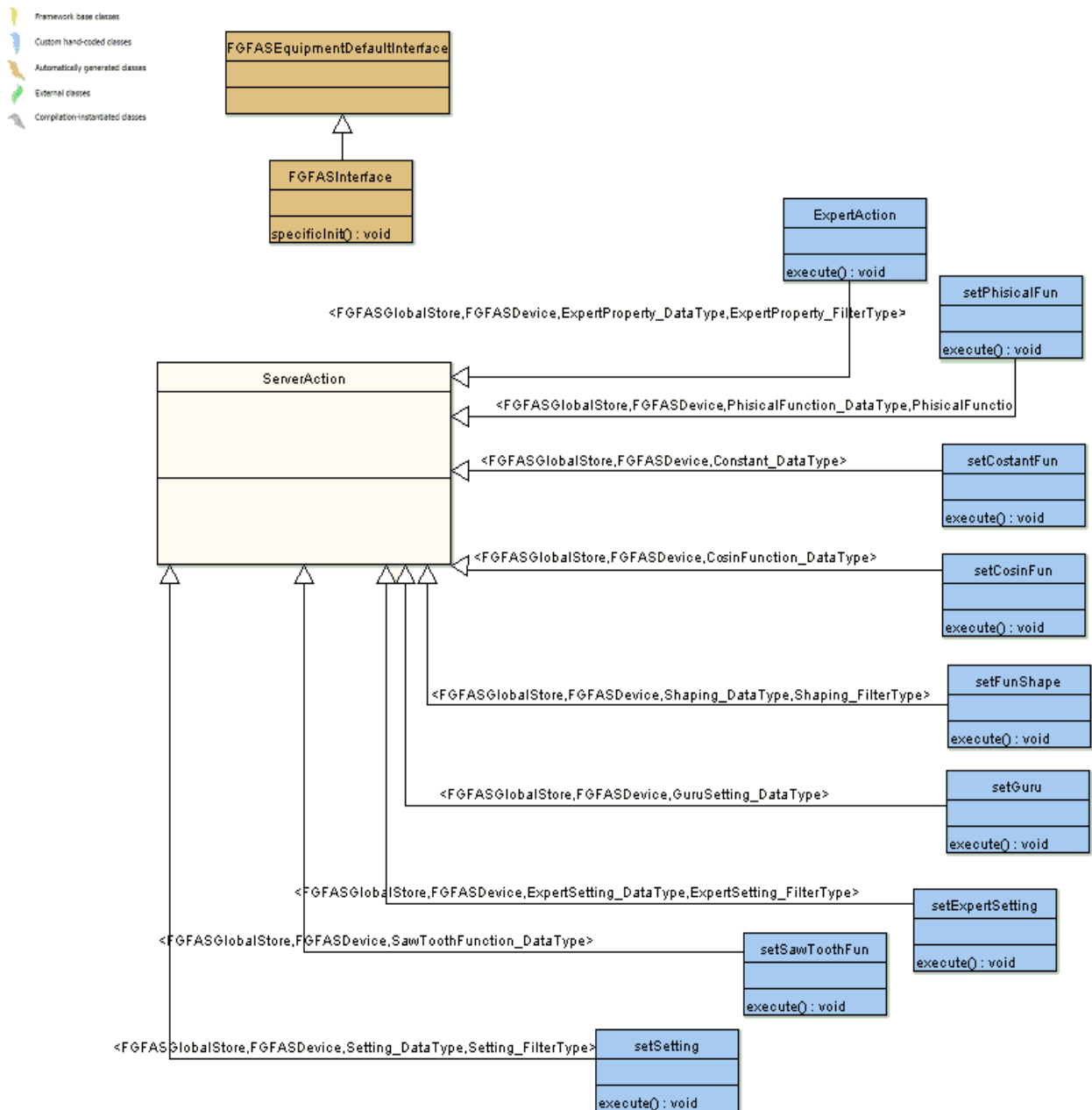


Figure 10.5 , Server classes (set).

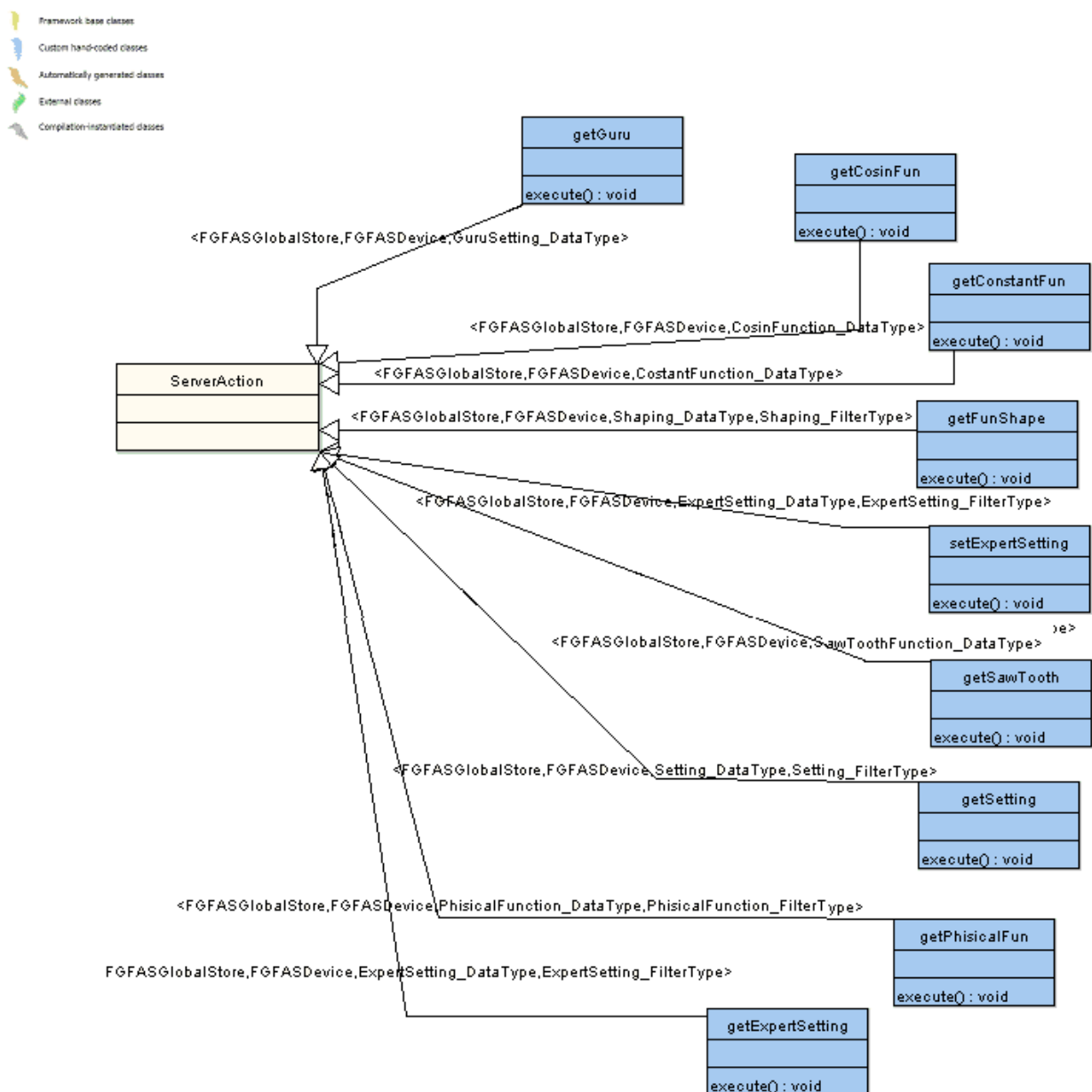


Figure 10.6, Server classes (get).

10.3.3 RT classes

Concerning the real time package we have the FGFASRT which extends the FGFASEquipmentDefaultRT.

It has the method `init()` that we can use to initialize all the fields of the device and to do all the other operations needed when the realtime activity is started.

Then, as show in the figure 10.6, we have two instances of RTAction class, one for each realtime action, and as parameter we used RTEvent, FGFASGlobalStore and FGFASDevice.

For both the classes (`doAquisition` and `doControl`) I implemented the method `execute()`, with the needed operations.

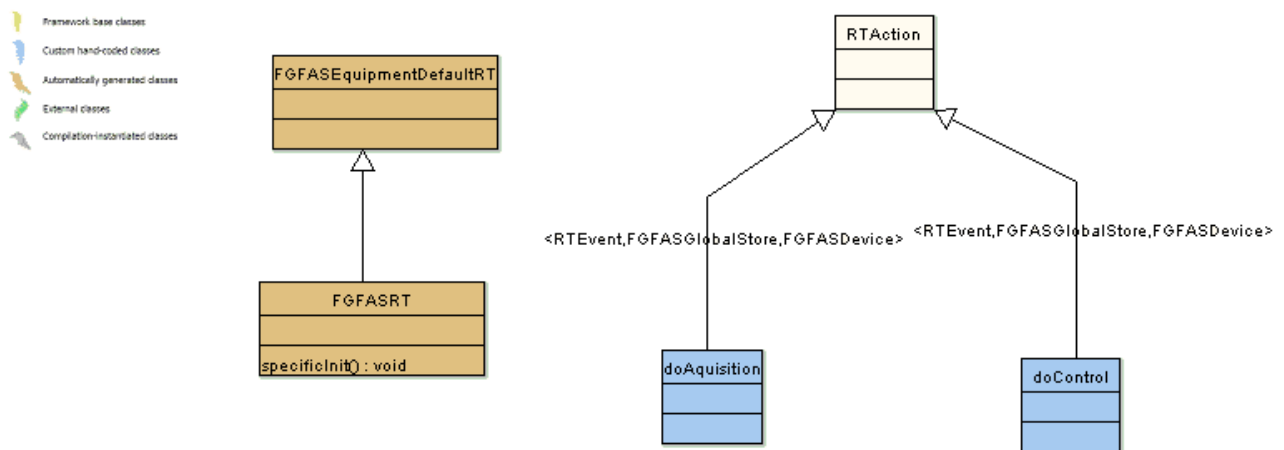


Figure 10.7, Real Time classes.

10.3.4 COMMON classes

To obtain the maximum modularity, analysing the code, I found that some utilities are independent from the framework and are linked only to the hardware device, then I decided to realize the two classes (GfasHwLib and GfasConvLib) that can be used regardless the kind of framework.

Another reason that led me to develop these classes is avoiding the duplication of code and I just decided to develop a third class (GfasVirtualUtilities).

GfasHwLib has all the static methods needed to read and write the different information from the GFAS memory.

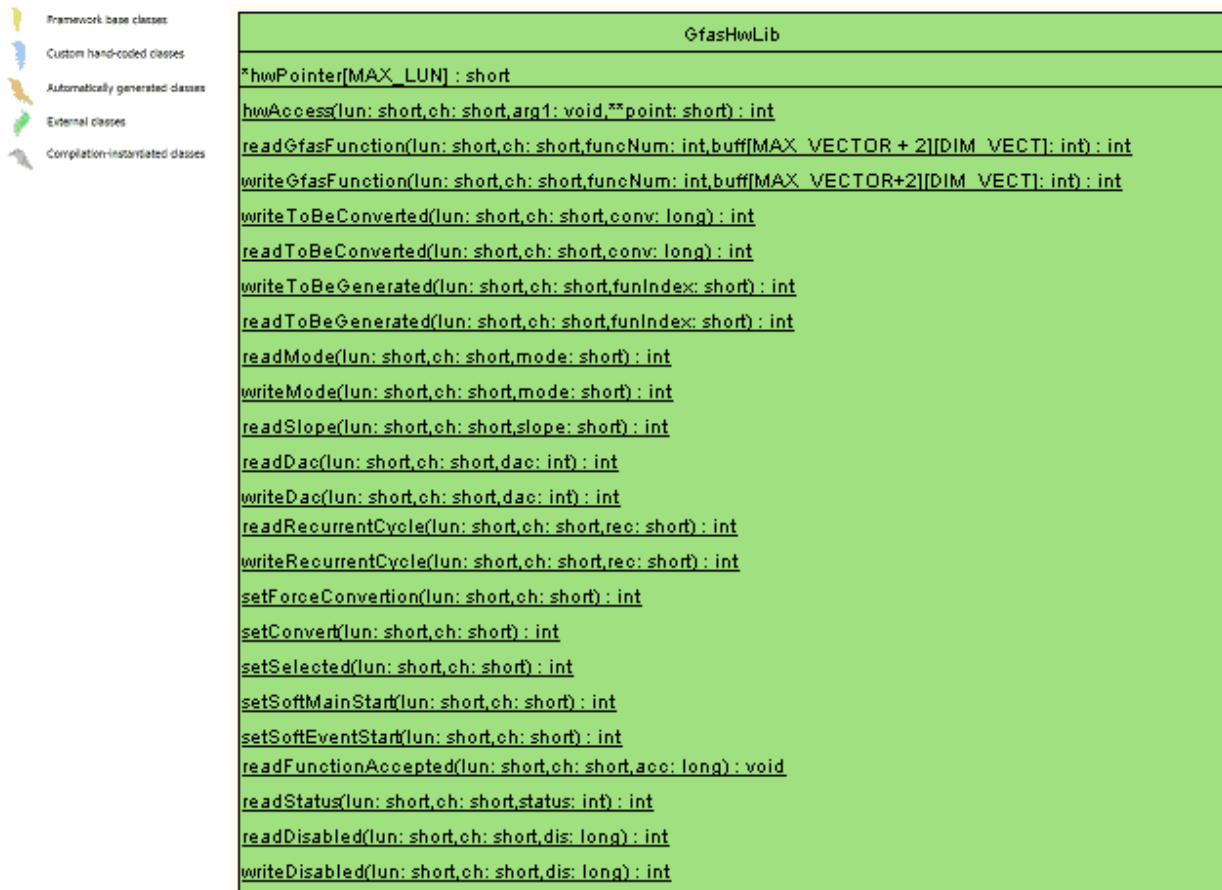


Figure 10.8, GfasHwLib.

Let us describe the different functions in details.

```
int hwAccess(short int lun, short int ch, unsigned short int **point);
```

to get the pointer to access the memory (it is a private function).

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

point is the pointer to the memory pointer,
return value is -1 in case of error 0 otherwise.

```
int readGfasFunction(short int lun, short int ch, int funcNum, int buff  
[MAX_VECTOR+2][DIM_VECT]);
```

to read a function from GFAS memory.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1),
funcNum is the number of function that the user wants to read.

output:

buff is the buffer used as output to retrieve the hardware function,
return value is -1 in case of error 0 otherwise.

```
int writeGfasFunction(short int lun, short int ch, int funcNum, const int buff  
[MAX_VECTOR+2][DIM_VECT]);
```

to write a function into the GFAS memory.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1),
funcNum is the number of function that the user wants to read,
buff is the buffer used as input to load the hardware function.

output:

return value is -1 in case of error 0 otherwise.

```
int readToBeConverted(short int lun, short int ch, unsigned long int& conv);
```

to read the bit map where every bit stands for one of the 32 functions (if the value of the i-th bit is 1, the i-th function has to be converted).

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

conv is the output with the 32 bit,
return value is -1 in case of error 0 otherwise.

```
int writeToBeConverted(short int lun, short int ch, unsigned long int conv);
```

to write the bit map where every bit stands for one of the 32 function (if the value of the i-th bit is 1, the i-th function has to be converted).

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1),
conv is the input with the 32 bit.

output:

return value is -1 in case of error 0 otherwise.

```
int readToBeGenerated(short int lun, short int ch, short int& funIndex);
```

to read the function number of the function that should be generated.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

funIndex is the output with the function number,
return value is -1 in case of error 0 otherwise.

```
int writeToBeGenerated(short int lun, short int ch, short int funIndex);
```

to write the function number of the function that should be generated.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1),
funIndex is the input with the function number.

output:

return value is -1 in case of error 0 otherwise.

```
int readMode(short int lun, short int ch, unsigned short int& mode);
```

to read the mode used.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

mode is the output with the mode value (0 = NORMAL, 1 = DAC, 2 = LOW JITTER, 3 = NORMAL RECURRENT, 4 = LOW JITTER RECURRENT, 5 = TEST1, 6 = TEST2, 7 = NO FUNCTION GENERATION),
return value is -1 in case of error 0 otherwise.

```
int writeMode(short int lun, short int ch, unsigned short int mode);
```

to write the mode used.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1),
mode is the input with the mode value (0 = NORMAL, 1 = DAC, 2 = LOW JITTER, 3 = NORMAL RECURRENT, 4 = LOW JITTER RECURRENT, 5 = TEST1, 6 = TEST2, 7 = NO FUNCTION GENERATION).

output:

return value is -1 in case of error 0 otherwise.

```
int readSlope(short int lun, short int ch, short int& slope);
```

to read the slope.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

slope is the output with the slope value,
return value is -1 in case of error 0 otherwise.

```
int writeSlope(short int lun, short int ch, short int slope);
```

to write the slope.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1),
slope is the input with the slope value.

output:

return value is -1 in case of error 0 otherwise.

```
int readDac(short int lun, short int ch, int& dac);
```

to read the DAC value.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

dac is the output with the hardware value of dac,
return value is -1 in case of error 0 otherwise.

```
int writeDac(short int lun, short int ch, int dac);
```

to write the DAC value.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1),
dac is the input with the hardware value of dac.

output:

return value is -1 in case of error 0 otherwise.

```
int readRecurrentCycle(short int lun, short int ch, short int&rec);
```

to read the number of recurrent cycles (-1 means infinite).

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

rec is the output with the number of cycles,
return value is -1 in case of error 0 otherwise.

```
int writeRecurrentCycle(short int lun, short int ch, short int rec);
```

to read the number of recurrent cycles (-1 means infinite).

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1),
rec is the input with the number of cycles.

output:

return value is -1 in case of error 0 otherwise.

```
int setForceConversion(short int lun, short int ch);
```

to set Force conversion flag.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1),

output:

return value is -1 in case of error 0 otherwise.

```
int getForceConversion(short int lun, short int ch, bool& conv);
```

to get Force conversion flag.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

conv is the Force Conversion flag,
return value is -1 in case of error 0 otherwise.

```
int setConvert(short int lun, short int ch);
```

to set conversion flag.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

return value is -1 in case of error 0 otherwise.

```
int getConvert(short int lun, short int ch, bool& conv);
```

to get conversion flag.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

conv is the Conversion flag,
return value is -1 in case of error 0 otherwise.

```
int setSelected(short int lun, short int ch);
```

to set select function flag.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

return value is -1 in case of error 0 otherwise.

```
int setSoftMainStart(short int lun, short int ch);
```

to generate a soft main start.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

return value is -1 in case of error 0 otherwise.

```
int setSoftEventStart(short int lun, short int ch);
```

to generate a soft start event.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

return value is -1 in case of error 0 otherwise.

```
int readFunctionAccepted(short int lun, short int ch, unsigned long int& acc);
```

to read the bit map where every bit stands for one of the 32 functions (if the value of the i-th bit is 1, the i-th function has been accepted).

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

acc is the output with the 32 bit,
return value is -1 in case of error 0 otherwise.

```
int readStatus(short int lun, short int ch, unsigned int& status);
```

to read the Status word.

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

status is the output with the value of the status word,
return value is -1 in case of error 0 otherwise.

```
int readDisabled(short int lun, short int ch, unsigned long int& dis);
```

to read the bit map where every bit stands for one of the 32 functions (if the value of the i-th bit is 1, if the i-th function is disabled).

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1).

output:

dis is the output with the 32 bit,
return value is -1 in case of error 0 otherwise.

```
int writeDisabled(short int lun, short int ch, unsigned long int dis);
```

to write the bit map where every bit stands for one of the 32 functions (if the value of the i-th bit is 1, the i-th function is disabled).

input:

lun is the board index within the VME crate,
ch is the channel (there are two possible channel 0 or 1),
dis is the input with the 32 bit.

output:

return value is -1 in case of error 0 otherwise.

The second class, that I developed, is GfasConvertingUtilLib with the following static methods that can be used:

- to convert the physical representation of a function into the hardware one and vice versa,
- to convert the physical representation of the DAC value into the digital one and vice versa,
- to compute the saw-tooth function,
- to compute the constant function,
- to compute the cosine function.

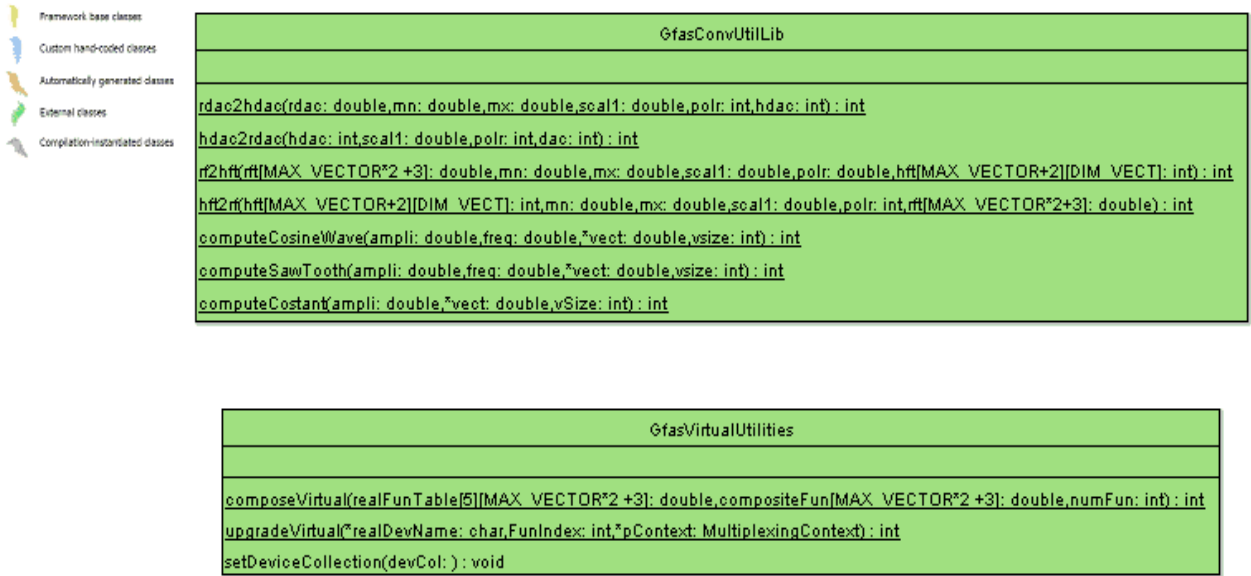


Figure 10.9, GfasConvUtil and GfasVirtualUtilities.

That is the description of all several static methods of GfasConvertingUtilLib with the input and output values.

```
int rdac2hdac(double rdac, double mn, double mx, double scal1, int polr, int &hdac);
```

to convert physical value of dac into digital one.

input:

rdac is the physical value of DAC,

mn is the minimum value possible for the physical values in rft,

mx is the maximum value possible for the physical values in rft,

scal1 is the scaling factor,

polr is the polarity (0 for bipolar DAC, 1 for unipolar DAC, 2 for digital output considered, bipolar, 3 for digital output considered unipolar).

output:

hdac is the output value for the digital value of DAC,

return value is -1 in case of error 0 otherwise.

```
int hdac2rdac(int hdac, double scal1, int polr, double &dac);
```

to convert digital value of dac into physical one.

input:

hdac is the digital value of DAC,

scal1 is the scaling factor,

polr is the polarity (0 for bipolar DAC, 1 for unipolar DAC, 2 for digital output considered, bipolar, 3 for digital output considered unipolar).

output:

rdac is the output value for the physical value of DAC,
return value -1 in case of error.

```
int rf2hft(const double rft[MAX_VECTOR*2+3], double mn, double mx, double
scal1,int polr,int hft[MAX_VECTOR+2][DIM_VECT]);
```

to build hardware function table out of real function table.

input:

rft is the input array of MAX_VECTOR*2+3 (see par. 5.2.1.1),

index	value
0	Number of vector
1	0
2	Origin value V0
3	T1
4	V1
5	T2
6	V2
.	.
.	.
.	.
1021	Tn
1022	Vn

mn is the minimum value possible for the physical values in *rft*,

mx is the maximum value possible for the physical values in *rft*,

scal1 is the scaling factor,

polr is the polarity (0 for bipolar DAC, 1 for unipolar DAC, 2 for digital output considered bipolar, 3 for digital output considered unipolar).

output:

hft the output array-2D of MAX_VECTOR+2 rows and DIM_VECT columns.(see par. 5.2.1.3)

hft[0][0]= number of vectors in the hardware function table	hft[0][1]= number of vector in the physical function table	hft[0][2]= Origin value V0
.	.	.
.	.	.
.	.	.
hft[i][0]= number of steps for the i-vector	hft[i][1]= multiplicity of basis step for the i-vector	hft [i][2]= amplitude V (i+1) in offset binary of i-vector
.	.	.
.	.	.
.	.	.
hft[511][0]= 0	hft[511][1]= 0	hft[511][1]= the value doesn't matter

return value is -1 in case of error 0 otherwise.

```
int hft2rf(const int hft[MAX_VECTOR+2][DIM_VECT], double mn, double mx, double
scall,int polr, double rft[MAX_VECTOR*2+3]);
```

to build real function table out of hardware function table.

input:

hft is the input array-2D of MAX_VECTOR+2 rows and DIM_VECT columns,

hft[0][0]= number of vectors in the hardware function table	hft[0][1]= number of vector in the physical function table	hft[0][2]= Origin value V0
.	.	.
.	.	.
.	.	.
hft[i][0]= number of steps for the i-vector	hft[i][1]= multiplicity of basis step for the i-vector	hft [i][2]= amplitude V (i+1) in offset binary of i-vector
.	.	.
.	.	.
.	.	.
hft[511][0]= 0	hft[511][1]= 0	hft[511][1]= the value doesn't matter

mn is the minimum value possible for the physical values in rft,

mx is the maximum value possible for the physical values in rft,

scall is the scaling factor,

polr is the polarity (0 for bipolar DAC, 1 for unipolar DAC, 2 for digital output considered bipolar, 3 for digital output considered unipolar).

output:

rft is the output array of MAX_VECTOR*2+3.

index	value
0	Number of vector
1	0
2	Origin value V0
3	T1
4	V1
5	T2
6	V2
.	.
.	.
.	.
1021	Tn
1022	Vn

return value is -1 in case of error 0 otherwise.

```
int computeCosinoWave(double phase_degs, int algo, double cos_ampli, double  
cos_freq, double* vect, int& vsize);
```

to compute a vector with the physical representation of a cosine wave.

input:

phase_degs is the phase,
cos_ampli is the amplitude,
cos_freq is the frequency,
algo is the algorithm A or B (PS/CO/Note 94-63).

output:

vect[MAX_VECTOR*2+3] is the output vector with the physical representation,

index	value
0	Number of vector
1	0
2	Origin value V0
3	T1
4	V1
5	T2
6	V2
.	.
.	.
.	.
1021	Tn
1022	Vn

vsize is the number of vectors used in the physical representation,
return value is -1 in case of error 0 otherwise.

```
int computeSawTooth(double ampli, double freq, double* vect, int& vsize);
```

to compute a vector with the physical representation of a saw-tooth function (no DC component).

input:

ampli is the saw-tooth amplitude, from 0 to max value,
freq is the saw-tooth frequency,

output:

vect[MAX_VECTOR*2+3] is the output vector with the physical representation,

index	value
0	Number of vector
1	0
2	Origin value V0
3	T1
4	V1
5	T2
6	V2
.	.
.	.
.	.
1021	Tn
1022	Vn

vsize is the number of vectors used in the physical representation,
return value is -1 in case of error 0 otherwise.

```
static int computeConstant(double ampli, double* vect, int& vsize);
```

to compute a vector with the physical representation of a constant function.

input:

ampli is the amplitude of function.

output:

vect[MAX_VECTOR*2+3] is the output vector with the physical representation,

index	value
0	Number of vector
1	0
2	Origin value V0
3	T1
4	V1
5	T2
6	V2
.	.
.	.
.	.
1021	Tn
1022	Vn

vsize is the number of vectors used in the physical representation,
return value is -1 in case of error 0 otherwise.

Finally we have the third class, GfasVirtualUtilities, it depends on the particular framework, then the only reason, why I decided to develop it, was to avoid replication of the code.
Now let us describe its functions.

```
int composeVirtual(double realFunTable[5][1023], double compositeFun[1023], int numFun);
```

to composite the real functions using all the virtual ones.

input:

realFunTable[5][1023] is the table with the physical representation of up to five virtual functions,
numFun number is the number of virtual function used.

output:

compositeFun[1023] is the function obtained using all the virtual functions,
return value is -1 in case of error 0 otherwise.

```
int upgradeVirtual(const char* realDevName, int FunIndex, MultiplexingContext* pContext);
```

to upgrade the real gfas using all the virtual gfas.

input:

realDevName is the name of the real Gfas,
FunIndex is the index of the function that we have to upgrade,
pContext is the multiplexing context in which we are working.

output:

return value is -1 in case of error 0 otherwise.

```
void setDeviceCollection(vector<FGFASDevice*>* devCol) {mydevCol = devCol;}  
to initialize 'mydevCol' with the right collection of device.
```

input:

devCol is the device collection with all the devices on a particular front end.

Chapter 11

Conclusions

Now it is time to conclude the presentation of the work that I have done at CERN.

To realize the equipment module I had many constraints because the result has to be equivalent to the previous version and all its operations had to be implemented.

The new framework features made it possible to realize a modular software with a high level of code reuse.



FESA tools were user friendly and they made working with the framework easy but for the GFAS a re-engineering work was necessary to understand the requirements of the software module.

Knowledge of the previous framework GM was necessary to distinguish between the code related to the GFAS and what was only related to the old framework and to its lack of modularity and code-reuse.

In the FESA GFAS the code common to all equipment is separated well from the rest to obtain the highest modularity.

The choice, that I made, to realize different classes for all the operations not related to the framework, made possible the re-use of all the converting functions for GFAS in LHC, which is using a different kind of framework and hardware.

The GFAS FESA class was one of the first classes realized using the FESA framework and I had a chance to see the framework evolving to the current version release after release while I was working.

Figure 11.1, GFAS board plugged into the psdsc12 VME.

At the beginning the first version of FESA GFAS had exactly the same design of the GM to make easier the test and debug of the code that I wrote. Only after this version was stable and well tested I moved to the new design reducing the number of properties by grouping together the previous properties as I already explained.

All the code has been written and tested on a front-end that is used for tests and it will be running soon on LEIR accelerator first and after that on the PS, PSB and SPS accelerators.

I tested the classes of the COMMON package outside the FESA environment, writing some test programs to make the tests simpler and more effective.

Tests represent an important part of the work, they were made both without timing domain and with the LEIR one, using the FESA Navigator and also the previous application software which was using the GFAS realized with GM framework.

All the results obtained with the FESA GFAS have been compared to the ones obtained with the GM GFAS. All of that because it was necessary to have a module equivalent to the previous one but with the innovations made possible by the new framework.

An oscilloscope has been used to display and trace the analogue functions produced using the GFAS device and its software FESA Equipment module.

All the source code, that I wrote, and the xml description of the design are available in the CVS repository and delivered for VME/LynxOS systems (ppc4).

References

- [1] Accelerator and Beam group web page <http://ab-div.web.cern.ch/ab-div/Index.html>
- [2] CERN web site <http://public.web.cern.ch/public/Content/Chapters/AboutCERN/AboutCERN-en.html>
- [3] CERN web site
<http://public.web.cern.ch/public/Content/Chapters/AboutCERN/HowStudyPrtcles/Accelerators/Accelerators-en.html>
- [4] CERN web site <http://public.web.cern.ch/public/Content/Chapters/AboutCERN/HowStudyPrtcles/CERNAccelComplex/CERNAccelComplex-en.html>
- [5] CERN web site <http://public.web.cern.ch/public/Content/Chapters/AboutCERN/HowStudyPrtcles/HowSeePrtcles/HowSeePrtcles-en.html>
- [6] CERN web site <http://public.web.cern.ch/public/Content/Chapters/AboutCERN/HowStudyPrtcles/Experiments/Experiments-en.html>
- [7] LHC Design Report Volume 1 (Chapter 14 Control System)
- [8] J. Serrano: The Ps Controls for newcomers 2nd edition. PS/CO/Note 99-22 (Tech.)
- [9] FESA Essentials. AB-CO-FC
- [10] FESA hangs on. AB-CO-FC
- [11] W. Heinze: THE EQUIPMENT MODULE FOR THE GFAS PS/CO/Note 93-108 (Tech.)
- [12] R. Maccaferri, W. Heinze: DAC versions for the GFAS. PS/CO/Note 94-73 (Tech.)
- [13] M. Gourber-Pace: Phase modulation control for 200 Mhz cavities. PS/CO/Note 94-63 (Spec.)
- [14] J. Lewis, J.C. Bau: Rapid SPS Super-cycles changes in the LHC era. AB-CO
- [15] J. Lewis, J.C. Bau, J. Serrano, D.Dominiguez, P. Alvarez Sanchez: Evolution of the CERN SPS timing system for LHC era. AB-CO
- [16] J. Lewis, J.C. Bau, M. Joker: The Central Beam and Cycle Management of the CERN Accelerator Complex.
- [17] A.Guerrero, J.J. Gras, J.L. Nougaret, M. Ludwig, M. Arruat, S. Jackson: CERN Front-end software architecture for accelerator controls. AB-CO-FC
- [18] A.Guerrero, J.J. Gras, J.L. Nougaret, M. Ludwig, M. Arruat, S. Jackson: FESA Interface Specification. AB-CO-FC
- [19] A.Guerrero, J.J. Gras, J.L. Nougaret, M. Ludwig, M. Arruat, S. Jackson: FESA Functional Specification. AB-CO-FC
- [20] A.Guerrero, J.J. Gras, J.L. Nougaret, M. Ludwig, M. Arruat, S. Jackson: CERN Front-End Software Architecture for accelerator controls. AB-CO-FC
- [21] Fesa C++ design. AB-CO-FC

Acknowledgments

I would like to thank CERN for giving me the chance to develop my thesis in the best possible conditions.

Thanks to my supervisor, Frank Di Maio, who has made me a part of the team since my first day, and has taken my opinions into consideration even when I was a newcomer.

Special thanks to Andrea Domenici for supporting me in this experience since the first mail sent to CERN.

Thanks Michel Arruat for having been a marvellous guide for the Fesa framework; I have learned so much from you, and working with you was a pleasure.

Thanks Wolfgang Heinze for the support for GFAS and for all your previous work.

For the pleasant working condition and for all the good times spent together, a special thanks to Nicolas De Metz-Noblat, Stephane Deghay, Alain Gagnaire, Yuri Georgievsky, Steen Jensen, Micheal Jonker, Tomasz Kokoszka, Krzysztof Kostro, Jean-Luc Nougaret, Bartolomiej Paszkowski, Anastasiya Radeva, Maciej Sobczak and Nikolai Trofimov.

For all the love and support, thanks to my parents: Franca Bovicelli and Antonio Taurelli.

Special thanks to Venusto Bovicelli, Maria Gabrielli, Quinta Lauri and Rodrigo Taurelli for all the values that they have handed down to me.

For all the fondness, thanks to Anna Maria Bovicelli, Margherita Bovicelli, Alfonso Ferrentino, Ioannis Grigoropoulos, Nikos Grigoropoulos, Enzo Scacciati, Riccardo Scacciati, Delfina Taurelli and Laura Taurelli.

Special thanks to Fiorella Cicogna, Carla Ferri and Chiara Della Volpe for all they taught me and for the love they have shown for their work.

Thanks to Mirko Balestri, Francesco Di Sandro and Cristina Rognini who have been with me regardless of the distance between us.

Reid, domo arigato gozaimasu.

For the help and good times thanks to Alan, Alessia, Anja, Annalisa, Arianna, Bibi, Cristina, Cucchia, Daniela, Daniele, Filip, Florent, Francesca, Francois, Gianluca, Gianni, Giulia, Gualtierio, Ian, Jacopo, John, Leonardo, Lucia, Marco, Maria, Martina, Matteo, Maurizio, Natalia, Natalie, Pana, Pascale, Patrizia, Raquel, Sabina, Sara, Simona, Simone, Sonia, Stefania, Stefano, Stefanos, Suzana, Valentina, Valerio and Viviano.