# MARTe: A Multiplatform Real-Time Framework

André C. Neto, Filippo Sartori, Fabio Piccolo, Riccardo Vitelli, Gianmaria De Tommasi, Luca Zabeo, Antonio Barbalace, Horacio Fernandes, Daniel F. Valcárcel, Antonio J. N. Batista, and JET-EFDA CONTRIBUTORS

***Abstract*—Development of real-time applications is usually associated with nonportable code targeted at specific real-time operating systems. The boundary between hardware drivers, system services, and user code is commonly not well defined, making the development in the target host significantly difficult. The Multithreaded Application Real-Time executor (MARTe) is a framework built over a multiplatform library that allows the execution of the same code in different operating systems. The framework provides the high-level interfaces with hardware, external configuration programs, and user interfaces, assuring at the same time hard real-time performances. End-users of the framework are required to define and implement algorithms inside a well-defined block of software, named Generic Application Module (GAM), that is executed by the real-time scheduler. Each GAM is reconfigurable with a set of predefined configuration meta-parameters and interchanges information using a set of data pipes that are provided as inputs and required as output. Using these connections, different GAMs can be chained either in series or parallel. GAMs can be developed and debugged in a non-real-time system and, only once the robustness of the code and correctness of the algorithm are verified, deployed to the real-time system. The software also supplies a large set of utilities that greatly ease the interaction and debugging of a running system. Among the most useful are a highly efficient real-time logger, HTTP introspection of real-time objects, and HTTP remote configuration. MARTe is currently being used to successfully drive the plasma vertical stabilization controller on the largest magnetic confinement fusion device in the world, with a control loop cycle of 50 $\mu$s and a jitter under 1 $\mu$s. In this particular project, MARTe is used with the Real-Time Application Interface (RTAI)/Linux operating system exploiting the new $\times$86 multicore processors technology.**

***Index Terms*—Computer control systems, data acquisition systems, real-time computer applications, real-time software systems, software performance.**

Manuscript received May 20, 2009; revised October 16, 2009. Current version published April 14, 2010. This work was supported by the European Communities under the contract of Association between EURATOM/IST and was carried out within the framework of the European Fusion Development Agreement. See the Appendix of F. Romanelli *et al.*, Fusion Energy Conference 2008 (Proc. 22nd Int. Conf. Geneva, 2008) IAEA, (2008). The views and opinions expressed herein do not necessarily reflect those of the European Commission.

A. Neto, H. Fernandes, D. F. Valcárcel, and A. J. N. Batista are with Associação EURATOM/IST, Instituto de Plasmas e Fusão Nuclear, 1049-001 Lisboa, Portugal (e-mail: andre.neto@ipfn.ist.utl.pt; hf@ipfn.ist.utl.pt; daniel.valcarcel@ipfn.ist.utl.pt; toquim@ipfn.ist.utl.pt).

F. Sartori, F. Piccolo, and L. Zabeo are with the EURATOM/UKAEA Association, Culham Science Centre, Abingdon OX14 3DB, U.K. (e-mail: Filippo.Sartori@jet.uk; Fabio.Piccolo@jet.uk; Luca.Zabeo@jet.uk).

R. Vitelli is with the Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata," 00133 Rome, Italy (e-mail: ricardo.vitelli@uniroma2.it).

G. De Tommasi is with Associazione EURATOM/ENEA/CREATE, Università di Napoli Federico II, 80138 Napoli, Italy (e-mail: detommas@unina.it).

A. Barbalace is with the Associazione EURATOM/ENEA, Consorzio RFX, 35127 Padova, Italy (e-mail: barbalace@igi.cnr.it).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNS.2009.2037815

## I. Introduction

THE design of a real-time system is usually intimately related to the functional requirements of the project. Commonly, this implies the software development to be tailored to a particular hardware or operating system allowing the designers to take full advantage of particular optimizations. Unfortunately, this leads to a lack in modularity and portability. As the size of the projects grows, particularly when multidisciplinary teams are involved, modular development is very important, allowing each unit to be independently developed and tested. Factors such as code reusability and maintainability also play a very important role and allow the developers to easily recycle functionality among different projects. Indeed, the ideal development environment should provide a common library containing all the sharable functionality. An object-oriented language is a natural choice since it is designed to maximize the quantity of reusable code units.

The code can usually be divided in two major sets: hardware interaction and algorithms. The first category greatly depends on the scope of the project and on the hardware itself. On the other hand, algorithms are likely to be reused in other projects. Going a step further, one can require the hardware interface to be divided in two sections: The first, highly machine dependent, must be rewritten for each type of device; the other provides a unique interface for the rest of the code and can even be shared among the same category of devices.

Code can be reused by further specializing existing sources or by reconfiguring binaries. In order to use the latest approach, the entity must be able to change its functional behavior accordingly to an external configuration. This data-driven approach is greatly improved by standardizing the way this configuration is provided.

At the same time, real-time programs have great concerns regarding performance and reproducibility. Any support features must guarantee to have a minimum, or at least constant, impact on performance. Several design choices must be carefully addressed when providing code that will work as a pillar for a real-time system.

MARTe is a control C++ modular framework designed for real-time projects [1], providing an easy way of deploying, configuring, and connecting real-time algorithms, together with a hardware interfacing abstraction layer. At the same time, it implements a series of concepts that try to maximize code reuse, maintainability, and portability while compromising to the minimum the impact on performance.

## II. Real-Time Library

MARTe is built upon a C++ multiplatform real-time library named BaseLib2. The first key feature of the library is the ability

to run the same code in different operating systems (OS). It is organized in consecutive layers where the lowest level implements the different calls for each of the available systems. In order to guarantee portability, the remaining layers and the end-user code must use the functionalities provided by the library, avoiding OS-dependent calls. This layered scheme provides a logical organization on how the code is distributed. Top layers have a broader view over the library functionality and tend to accommodate code that is less critical for the operation of a system, granting at the same time a set of tools that greatly eases the development of applications. When porting the library to a new OS, only the lowest layer needs to be adjusted.

The layer closest to the operating system provides all the abstraction regarding the interaction with the file system, networking, threading, semaphores, atomic operations, and OS-dependent optimizations. BaseLib2 was already ported to the following OS: VxWorks, Linux, Linux/RTAI, Solaris, and MS Windows. The possibility of running the same code in different systems allows developers to write and debug code using tools that might not be available in the target arrangement. This is even more important when the final environment has no memory protection, making the debug of algorithms particularly difficult. Porting of operating systems that implement the POSIX standard for real-time application interfaces (1003.1b and 1003.1c) is rather straightforward since it was already performed.

BaseLib2 tries to maximize the concept of reference and to dissuade as much as possible the use of pointers. For this purpose, a garbage collector and a special root class object is available. Objects of this kind can be automatically constructed by the library and are destroyed by the garbage collector when they are no longer required by any external references. These objects are also allowed to have a name and are, upon construction, automatically added to an internal tree configuration database. By searching in this database, a reference to the object can be later obtained. This kind of approach tries to mitigate the errors that arise from the misuse of pointers, which are typically very hard to debug and reproduce.

In order to automatically create objects, the library provides a standard configuration language. This configuration is provided to the library as a stream and analyzed by a parser that instantiates the objects based on the requested classes. If the object is successfully created, it is supplementary configured by automatically validating and assigning the configuration parameters against its internal variables. Figs. 1 and 2 contain an example of a possible configuration. In order to guarantee real-time efficiency, the object is expected to perform the majority of the required validations and allocations at this stage. Once the object is successfully built, it is added to the previously described tree. A garbage-collectable container object is also provided, allowing to recursively create new nodes in the internal database.

A series of classes are dedicated to strings and data streaming. These permit to perform a series of advanced operations in character sequences and to perform input/output (I/O) operations without requiring the concept of beginning and end of transmission. Most of the I/O classes are abstracted to a level where the knowledge of the target media (file, socket, memory, etc.) is not required.

```
+HttpServer = {
  Class = HttpService
  Port  = 8084
}
+MARTe = {
  Class = MARTeContainer
  +RTThread1 = {
    Class = RealTimeThread
    +Controller = {
      Class = ControllerGAM
      NoPlasmaCurrentGain  = 40.0
      IPWaveform = {
        Times      = {0 120}
        Amplitudes = {0.5 0.5}
      }
    }
    +DAM = {
      Class = DAMGAM
      ProjectionMatrix = {
        0 = { 1.0 0.0 0.0 0.0 }
        1 = { 0.0 1.0 0.0 0.0 }
        2 = { 0.0 0.0 1.0 0.0 }
        3 = { 0.0 0.0 0.0 1.0 }
      }
      ...
    }
    ...
  }
...
```

Fig. 1. An example of a BaseLib2 configuration. The + signal before the name of the object will enforce it to be automatically created and configured by the library.

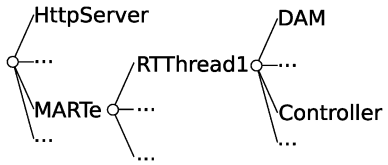


Fig. 2. The library is data driven using a syntax that allows to automatically create and configure objects. Once created these are added to an internal tree database which can later be used to search and retrieve references. This figure corresponds to the configuration presented in Fig. 1.

Live introspection and analysis of objects is provided by the CINT C++ interpreter [2], which permits to verify variables' internal states and expected behaviors. Due to its wide availability and simplicity, the Hypertext Transfer Protocol (HTTP) has been chosen as the standard transfer mechanism of information when querying objects. BaseLib2 provides an internal multithreaded HTTP server that can browse any objects that choose to enable an internal interface. The internal tree configuration database is also browsable and is rendered as a recursive tree with links to all the objects that have the HTTP interface implemented. This scheme allows library users to have a standard user interface for all the application built with the library. The GET and POST HTTP requests are also implemented, providing a way to actively interact with the objects. Multiple servers can coexist at the same time by using different ports.

Object-oriented languages provide a very good programming paradigm when designing an application. The concepts of data abstraction, inheritance, and modularity permit to define and even impose how the different pieces of the application should interact. Sometimes, and particularly true when providing a library, one cannot fully constrain or predict how something is

going to behave. In order to decouple concepts, higher level protocols are usually used. Applications developed using the BaseLib2 library can use a series of message classes that allow objects to communicate. These messages contain a sender and a receiver object address, specified as a unique path in the internal tree configuration database. A network-based message server enables the interaction between applications living in different machines. The library automatically searches for the object address in its database and, if found, calls the appropriate function. Messages can also be sent in synchronous mode where the sender expects a reply from the receiver. The content of the messages is free, and they are routinely used inside the library to start and stop services and by some objects to update its internal behavior.

One example of such an object is the state machine class. It is updated by changing its internal states accordingly to the messages received, and upon state changing, it broadcasts new messages to registered listeners. All of these actions are configurable without changing the source code.

The library also provides a series of algebraic and mathematical tools. The widest range is in matrix calculations where several matrix decomposition methods, like the single-value decomposition (SVD) and LU decomposition, are available.

### A. *Logger*

In order to assure the correct function of a system, a logger mechanism is of huge importance. A good logging scheme permits to expeditiously analyze any problems that may arise during operation. On the other side, the logger should not compromise in anyway the real-time performance.

When using BaseLib2 classes, a series of logging functions are available. By using these functions, it is possible to automatically append to the desired message information like the name of the object, its location in memory, the time when the message was issued, the thread and process identifiers, as well as the name of the machine where it was generated.

A minimum impact on the system is guaranteed by the use of a consumer/producer scheme. The process wishing to write a logger message appends it to a FIFO queue. A consumer running with very low priority processes the messages in the queue when processor time is available. This permits to even report information from an interrupt service routine where the scheduler is usually frozen.

The library makes no assumptions about how messages are consumed. The default behavior is to send it using the User Datagram Protocol (UDP) to a stated address. Using this scheme, a relay logger mechanism was developed, allowing to combine messages arriving from different machines.

This application has been designed so as to add as many logger processing nodes as required. When a relay logger node is installed, its default behavior is to route messages to another logger node. Nodes can also contain multiple plug-ins to process locally the arriving information. Two types of plug-ins are available: message proxy, allowing the same node to route to several machines, and data saving, storing all the incoming messages in a persistent media. The data storage plug-in can be configured to recycle saved files after a specified size or data age.

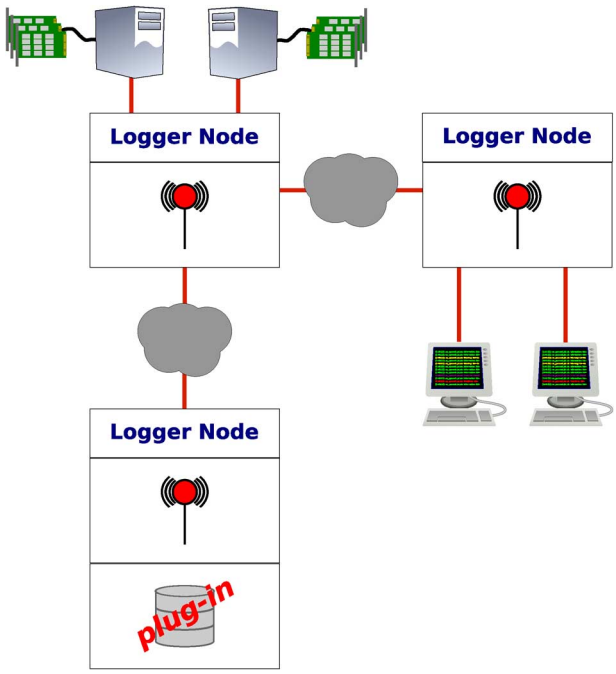


Fig. 3. Relay logger mechanism. Messages are automatically relayed across networks, allowing to add new logger nodes as required. In this figure, messages are produced in the acquisition network and further relayed to a node responsible for data persistence and another node for live visualization.

Since a system can produce hundreds, or even thousands, of information messages per day, a Java application that organizes messages accordingly to the available criteria has been developed. Messages are grouped first by producer, where each machine has a node in a tree, and inside every node, messages are grouped by severity, thread, and object producer name. Filtering enables us to follow only a narrowed subset of messages type (e.g., messages with a particular alarm level from a specific object in a subsystem). Fig. 3 shows a design where messages are produced in a private network and subsequently broadcasted to a data persistence node and a live information system.

## III. MARTe Components

MARTe is a modular collection of a series of components that permits the system to work as a whole. Some of the blocks are intended to be further adapted to the target application, while others are expected to be used as provided. In this section, we analyze in detail each of the basic components, starting from the atomic bit until arriving to the macroscopic vision of the system.

### A. *Generic Application Module*

The atomic element of MARTe is named Generic Application Module (GAM), and all applications built using the framework are designed around these components. A GAM is a block of code implementing an interface specified in the BaseLib2 library. Each GAM contains three communication points: one for configuration and two for data input and output. GAMs are setup using the same configuration syntax described before in the real-time library section. The core of a typical GAM processes the input accordingly to how it was configured and outputs the modified information. The only entry point that is mandatory is the configuration, and some GAMs—for instance, modules

that interface hardware—are usually read- or write-only. These modules also have an internal state that keeps track of its internal history.

During initialization, the modules declare what data they expect to receive and what information is going to be produced in output. Each type of input or output is declared as a configurable named signal with a data type associated. This is the only available way to chain GAMs and provides a clear boundary in the system: GAMs are not aware of other modules.

Although GAMs are unrestricted in functionality, the set of GAMs that interface with hardware can be conceptually extracted. These kinds of modules are named IOGAMs and provide a unique high-level interface to any kind of hardware. The connection between the IOGAM and the specific low-level code responsible for driving the hardware is performed through the specialization of a high-level class named generic acquisition module. This interface requires the number of hardware inputs and outputs to be specified and forces the existence of a reading and write functions. These functions must be implemented for each kind of devices, although it is common that devices belonging to the same family are able to share a common acquisition module. These high-level interfaces to the hardware can usually be configured to return the latest acquired value or to wait for a new sample to arrive (at the cost of delaying the execution), depending on the requirements of the application.

The most common type of GAMs allows the designers to interface with hardware, store acquired data, execute algorithms, take decisions based on the current state of the system, debug internal states, and provide live information about the system. When developing an application with MARTe, the greatest majority of the work resides in the development of the needed GAMs and their organization.

Another important feature in MARTe is the ability to perform simulation without interfering with the plant. Although this greatly depends on the target project, if up to some extent the system can be modeled, a GAM can be used to simulate the inputs (possibly using real signals saved beforehand) and another to predict the output of the system. This arrangement permits to debug and tune all the other modules or to test the possible effects that a change in a module configuration will produce. On the live system, the modules can later be swapped with the real interface to the hardware.

### B. Dynamic Data Buffer

Data is transferred between GAMs by using an optimized memory bus named Dynamic Data Buffer (DDB). The first role of this entity is to ensure coherency across the system, verifying if all the signals requested by each of the GAMs are produced by one module and in case of any inconsistency issue an error. Although it is not the default behavior, a GAM may write over a signal already produced by another module. This process, named patching, must be explicitly requested, otherwise it is assumed as an error. Fig. 4 illustrates the interaction between two GAMs through a DDB.

MARTe also provides a collection of GAMs to send signals between different systems, even in different machines. Depending on the system, real-time may only be guaranteed on the producer.

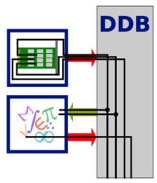


Fig. 4. The DDB is the only available way for GAMs to interchange information. In this figure, a GAM interacts with a hardware device and produces three signals, two of which are consumed by the subsequent module to produce a forth signal.

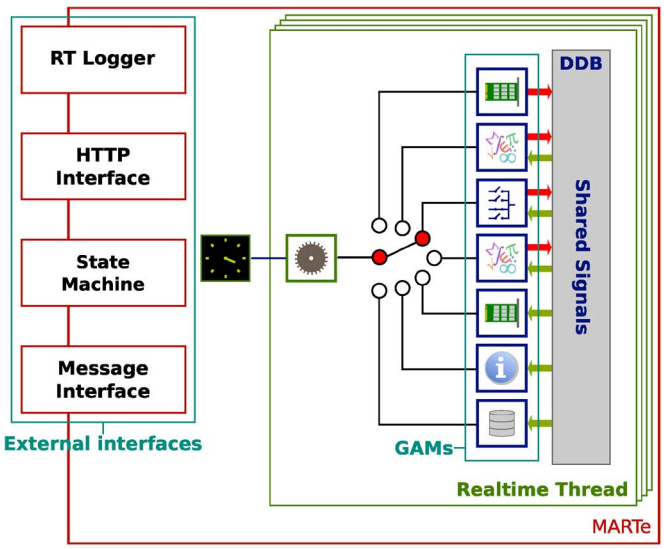


Fig. 5. The real-time thread acts a module micro-scheduler. In this figure a group of seven GAMs is executed at each cycle. The first GAM acquires data from a device and outputs it to the memory data buffer. Once processed some of the data is written back to the hardware.

### C. Real-Time Thread

The real-time thread is a container of GAMs and acts as a GAM microscheduler being responsible for their sequential execution. The thread can be configured to run in specific processors and can be assigned to a specified priority. It also tracks execution times and keeps a series of internal timing information about each of the GAMs for which it is responsible. The list of GAMs that are executed can be changed accordingly to the state of the system. This is performed through a message interface as described in the real-time library section. Depending on the severity of an error reported by one of the modules, the thread can be configured to take an action, which is usually the complete stop of execution.

MARTe must contain at least one real-time thread. When designing a new system, one of the major challenges is to decide what GAMs, and in what order, are to be executed. Fig. 5 depicts a possible set of modules that start by acquiring signals from a hardware device, processing and taking decisions upon this data, and finally outputting the signals to both a device and a storage scheme. Threads can be configured to run at a specific frequency, and all the GAMs are expected to execute within this time. A warning or an error is issued every time the execution time is larger than the specified cycle period.

### D. Synchronization

The framework requires the existence of at least a timing source. A shared variable in the real-time thread tracks the absolute time in microseconds. Depending on the project, this time is usually expected to be updated by an external hardware through an IOGAM, although sometimes the CPU clock can be used, particularly when developing and debugging a new project. MARTe provides ready-to-be-used high-precision time emulators based on the CPU time and optimized for each of the supported operating systems. A new control cycle is started when the absolute time is a multiple of the requested cycle period.

The previously described acquisition module also forces the presence of a synchronization function. When configuring MARTe, one chooses if this synchronization module has to work in interrupt or polling mode. In the first case, the execution of MARTe is arrested up to the arrival of an external interrupt. If polling is selected, the system will continuously query the interface until an answer is provided. In both cases, when an answer arrives, the absolute time is updated, and the framework checks if a new cycle is to begin.

As this is one of the most delicate parts of the system, a series of checks and timeouts can be configured. The system designer must decide what actions to perform when a timeout or error occurs.

### E. Multiple Real-Time Threads

MARTe is able to handle a collection of real-time threads. These can either be configured to run concurrently on the same processor or in parallel. The only way of sharing data between threads is to use a special IOGAM, provided by the framework, as the output of the thread producing data, connected to an input acquisition module in the thread consuming the signals. For MARTe these two synchronization GAMs emulate the presence of a physical hardware which would produce and consume the data, allowing to have the two threads completely decoupled. The output module can also be used as the timing source for the driven thread and can be configured to send under sampled filtered data, allowing to specify different running frequencies.

## IV. Interfacing to MARTe

As stated in the previous section, the only compulsory requirement for the startup of a MARTe-based application is the existence of at least one real-time thread. Fig. 5 illustrates a collection of available external interfaces to MARTe, which are not bounded to a particular project. These permit to interchange information with the framework, both by querying its components' internal values and by actively updating the state or values of some of the components.

### A. Message Interface

Using the BaseLib2 message protocol, it permits to send synchronous messages to the framework. MARTe internal services are started and stopped upon the receipt of a specific message. The interface is designed to be connected to an external object that is able do decode an external configuration protocol—for instance, from a human–machine interface—into a MARTe-recognized message. This is the preferred way to change the list of modules to be executed and to update GAMs parameters. After receiving a message, the framework verifies its validity and forwards it internally. If reconfiguration is required, the threads being executed are eventually stopped.

### B. HTTP Interface

The framework uses the HTTP server provided by BaseLib2 and supplies a collection of HTTP-based components that allow to setup a communication channel with all the internal elements. Examples of these utilities are a remote configuration file upload and a signal server to download data acquired during a certain period of time. Live diagnosis of threading and memory activity are also available, together with the list of all the registered objects. All the modules that expose render information are also highlighted.

## V. First Application—JET Vertical Stabilization

The Joint European Torus (JET) is the largest magnetic confinement fusion device [3] in the world. Plasma is confined inside the chamber using strong magnetic fields, and its position and shape controlled by a combination of coils. In order to obtain better fusion performances, the plasma is forced to be vertically elongated. Unfortunately, such plasmas are vertically unstable [4] and must be controlled in closed loop. The aim of the vertical stabilization (VS) system is to control the instability by driving the current in a set of poloidal field (PF) coils so that a radial magnetic field is produced. The loose or erroneous control of the instability can have huge negative impacts, as plasma disruptions (complete loss of thermal and magnetic energy) may occur, inducing large currents and forces in the machine vessel. An international ongoing project for the upgrade of the JET plasma control (PCU) [5] will allow to greatly enhance the current VS system. The requirements are the execution of the closed loop cycle within 50 $\mu$s with a maximum jitter of 2.5 $\mu$s, comprising the interface with hardware and data processing. An application fulfilling these requirements was designed using the MARTe framework, where a multidisciplinary team contributed to the different parts of the system: Hardware and software engineers developed the integration with hardware and the overall setup of the system, while control engineers produced the GAMs for modeling and control.

Although JET is a pulsed machine, with experiments that can last up to 2 min and where the presence of the VS system is vital, it was decided from the beginning to have the vertical stabilization always running and guaranteeing real time, making no distinction between operational phases. This is particularly relevant for future fusion devices like ITER [6], where subsystems will have to adapt to a steady-state operation reality. This was also only possible since no pulse-based assumptions were made in the design of the framework.

### A. System Configuration

The system runs on an Intel Quad-Core processor and uses the RTAI implementation [7], with a standard Linux kernel option to explicitly assign interrupts and Linux tasks to a single core, breaking the symmetric multiprocessor scheduler decisions. All MARTe threading activity is placed on the remaining three cores, allowing one to be completely dedicated to the real-

time thread. This single feature immediately guarantees that no external activity or spurious interrupts will deviate the real-time system from its high-priority activities. It is important to notice that all these configurations are completely transparent to MARTe and no code modifications had to be performed, not even to change in which cores threads should run or how memory should be arranged. The downside of this implementation is that, living in kernel space, software faults will have large impacts on the system, making it very hard to debug problems that are not closely related to the framework, mainly in the GAMs algorithms. This is where the multiplatform configuration proves to be a powerful mechanism as all the code can be tested as is in a more friendly environment like Linux or MS Windows.

### B. Hardware Interface and Synchronization

The hardware of the data acquisition system is based on the PICMG 3.0 Advanced Telecommunications Computing Architecture (ATCA) standard [8] and contains six data acquisition cards. Each board comprises 32 18-bit resolution analog-to-digital converters acquiring at 2 Msamples/s. The cards are connected to the controller computer using the Peripheral Component Interconnect Express (PCIe) point-to-point links through the ATCA backplane [9]. A rear transition module (RTM) connects the system to the radial field amplifier.

When designing the hardware interface driver for the data acquisition system, it was decided to use an interruptless environment with a data polling scheme, but without compromising the other interfaces to the computer, e.g., network access. The data acquisition boards map in the controller computer memory a set of four buffers as described in Fig. 6. The selected buffer is consecutively cycled every 50 $\mu$s by the firmware. The first value written is the header and contains the absolute time since the last trigger, followed by the values of the ADCs, and finally by the footer containing the same value as the header. The driver continuously queries the value of next header to be written, and as soon as it changes, it starts to check the footer. When these two values are the same, the driver signals the high-level IOGAM, which in turn broadcasts and updates MARTe absolute internal time. The framework uses the values provided by the high-resolution timers of the processor to measure the performance of the system and to keep track of the jitter associated with the synchronization mechanism. The firmware also assures synchronization between all boards.

### C. Application Modules

The overall system is composed of a collection of 18 modules. The first GAM synchronizes and retrieves data from 192 ADC channels, which are then written into the DDB. Using some of these signals, the second GAM produces a series of synthetic data, like threshold detection and linear combinations of magnetic signals. The observer GAM subsequently provides an estimate of the plasma vertical velocity, followed by a scheduler module deciding which of the following controller GAMs must be executed.

Four controller GAMs are then sequentially executed, and only the one previously selected by the scheduler, through a signal in the DDB, performs real work. Two modules, named vertical and divertor amplifier, are allowed to override the output of the controller with some special features, like dither or hysteresis, and are run before calling the module responsible for writing the output back to the plant and close the loop. A collection of six different GAMs acquire in memory the data for each type of signal: ADC, controller, debug, performance, waveforms, and asynchronous. Finally, a module performing live statistics of configured signals is executed. The scheme is depicted in Fig. 7, and mean values for the time of execution of these application modules are reported in Table I.

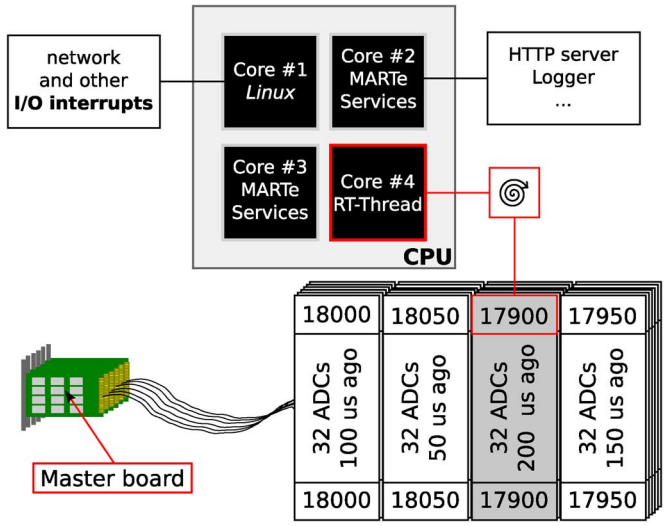


Fig. 6. Data synchronization is performed in the master board, which is guaranteed by the firmware to be the latest to have data available. Once new data is available, it is collected and a new MARTe cycle starts. The CPU core isolation scheme allows to protect the real-time environment from spurious and undesired interrupt sources.

### D. Interfacing With JET

The JET Control and Data Acquisition System (CODAS) [10] is the entity responsible for providing control, monitoring, and data acquisition to all the existent subsystems. As a JET subsystem, the vertical stabilization uses the tools provided by CODAS both for plant configuration and data retrieval. Between pulses, experts are allowed to change the system configuration by mainly updating values in the different GAMs. This is performed using a user interface designed specifically for MARTe that contains several access layers and where expert users and engineers are allowed to upload configuration files for each of the modules. VS operators are expected to act on an upper layer where atomic values are validated and already have a unique and direct physical meaning. While commissioning the system, the first approached was preferred as some configuration values and parameter organization required some time until converge. As of today, and with configuration files that can easily have more than 7000 lines, the requirement for an advanced user-interface was almost compulsory, allowing to capitalize all the VS experimental advanced features [11], minimizing the risks of configuration and regression faults.

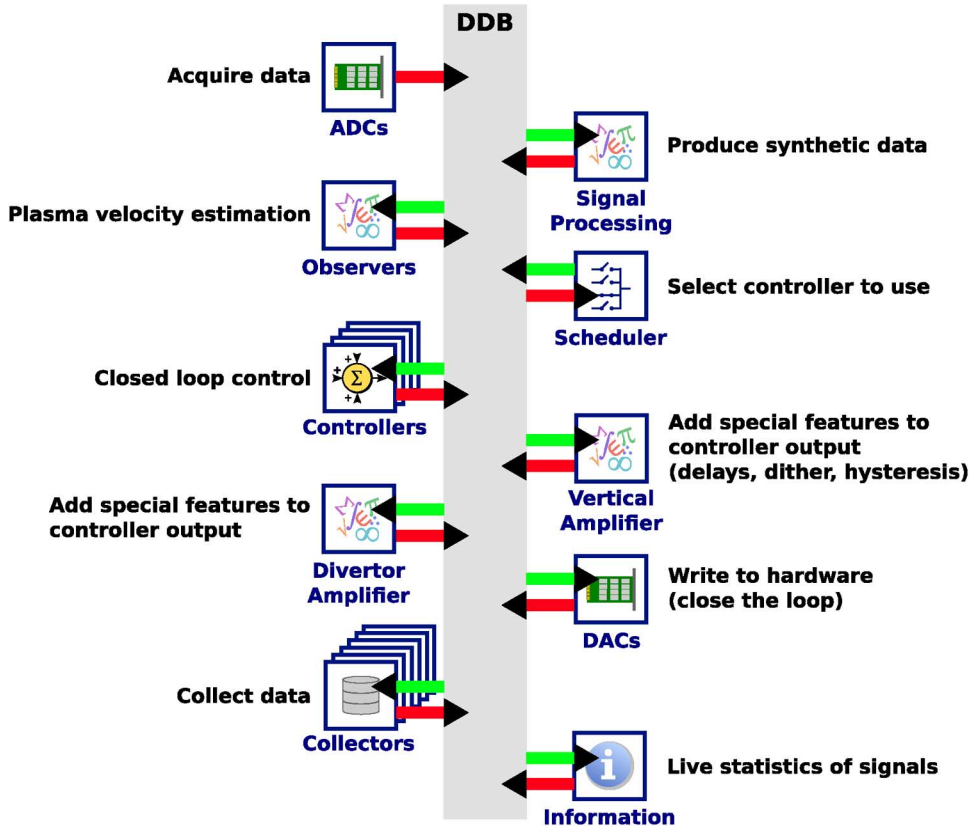


Fig. 7. All the modules executed for each control cycle by the real-time thread. The first module provides the synchronization with the hardware. All the code must be executed in less than 50 $\mu$s.

TABLE I
EXECUTION TIMES OF THE VS APPLICATION MODULES

| GAM | Mean ($\mu$s) | Std. dev. ($\mu$s) |
|---|---|---|
| ADC | 2.43 | 0.26 |
| Signal processing | 5.14 | 0.01 |
| Observer | 4.00 | 0.04 |
| Scheduler | 0.37 | 0.01 |
| Controller 1 | 1.01 | 0.02 |
| Controller 2 | 0.31 | 0.01 |
| Controller 3 | 0.28 | 0.02 |
| Controller 4 | 0.26 | 0.01 |
| Vertical amplifier | 0.85 | 0.03 |
| Divertor amplifier | 0.59 | 0.02 |
| DAC | 0.39 | 0.02 |
| Data collection 1 | 2.83 | 0.07 |
| Data collection 2 | 0.84 | 0.05 |
| Data collection 3 | 1.46 | 0.34 |
| Data collection 4 | 1.08 | 0.02 |
| Data collection 5 | 0.92 | 0.04 |
| Data collection 6 | 0.74 | 0.06 |
| Statistics | 1.24 | 0.02 |

### *E. First Results*

The new vertical stabilization architecture has been running in parallel with the previous version of the system since the Summer 2008, and it was already used several times to control the machine in closed loop. Several consecutive weeks of operation with no faults were already achieved with the system running and in real-time 24 h per day. During these long operation periods, the configuration was changed by the operators a considerable amount of times, and the system proved to have the capacity to withstand and react to distinct arrangements.

CODAS is collecting an average of 320 signals per pulse, amounting to several hundreds of megabytes of data. The data collection modules can be configured to acquire several windows at different frequencies, but higher importance is given to the data acquired by the ADCs, as the magnetic signals can be used later for simulation and modeling purposes.

One of the major accomplishments regarding the software side of the project was the achievement of jitters well under 1 $\mu$s, as shown in Fig. 8 where the standard deviation is 0.12 $\mu$s. These figures are always true, not only during the pulse, and were only possible due to key design decisions in the VS configuration, mainly the interrupt and processor isolation. Again, this was only possible due to the uncommitted way MARTe was designed, allowing to configure the system (e.g. threading) in a completely transparent way.

## VI. CONCLUSIONS

MARTe is a modular real-time multiplatform C++ framework tailored at, but not restricted to, the development and deployment of control systems. It is built upon a layer-based library that provides a series of key concepts that make code safer and easier to debug. Among these are a highly efficient

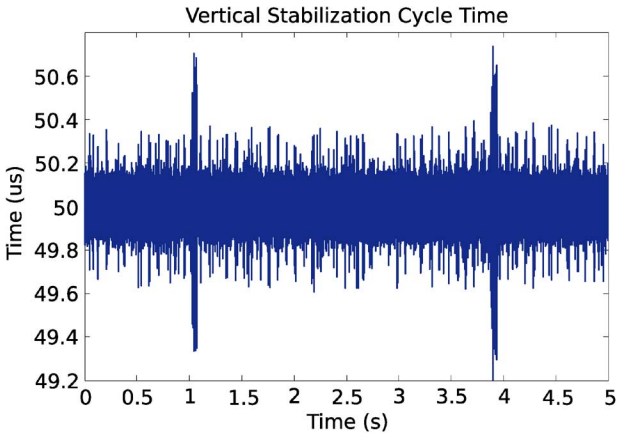


Fig. 8. The system is continuously real-time with a jitter inferior to 1 $\mu$s. In this figure, $10^5$ control cycles were performed, and the error relative to the 50 $\mu$s is due to the imprecision on the time measurement (using the processor timers) and to the natural jitter associated with accessing a memory location.

logger mechanism, built-in object introspection, garbage collection of named objects, and data-driven object creation and configuration.

One of the most important conceits provided by MARTe is the generic application module. It allows to provide a clear boundary between hardware interface, algorithms, and system configuration. Similar systems can be reused just by changing the modules that are to be executed by the real-time threads. In the same manner, hardware interface as seen by the platform is always kept constant, and only low-level drivers must be eventually developed. The structure is fully reconfigurable using a well-defined syntax.

The framework is already being used to drive the vertical stabilization of the JET tokamak, a vital system for the machine operation, where erroneous control can have drastic effects in the machine itself. The target values for the control system had a maximum cycle time of 50 $\mu$s with a jitter inferior to 2.5 $\mu$s. The worst jitter obtained has a maximum value of around 0.8 $\mu$s, mainly due to a wise combination of framework potentiality and software to hardware interface. In this cycle time, a series of 18 GAMs performing a wide variety of tasks are executed. MARTe continuously drives the vertical stabilization system in real time, 24 h per day, and several consecutive weeks of operation were already achieved with no faults.

MARTe is also the real-time executor for the plasma control in the COMPASS tokamak [12] and for the tomographic reconstruction [13] in the ISTTOK tokamak.

## REFERENCES

[1] G. D. Tommasi, F. Piccolo, A. Pironti, and F. Sartori, "A flexible software for real-time control in nuclear fusion experiments," *Control Eng. Practice*, vol. 14, no. 11, pp. 1387–1393, 2006.

[2] R. Brun and F. Rademakers, "Root-an object oriented data analysis framework," *Nucl. Instrum. Methods Phys. Res. Sec. A, Accel., Spectrom., Detect., Assoc. Equip.*, vol. 389, no. 1–2, pp. 81–86, 1997, New Computing Techniques in Physics Research V.

[3] F. Romanelli *et al.*, "Overview of JET results," *Nucl. Fusion*, vol. 49, no. 10, p. 104006, Oct. 2009.

[4] R. Albanese, E. Cocorrese, and G. Rubinacci, "Plasma modeling for the control of vertical instabilities in tokamaks," *Nucl. Fusion*, vol. 29, no. 6, pp. 1013–1023, 1989.

[5] F. Sartori, F. Crisanti, R. Albanese, G. Ambrosino, V. Toigo, J. Hay, and P. Lomas *et al.*, "The JET PCU project: An international plasma control project," *Fusion Eng. Design*, vol. 83, no. 2–3, pp. 202–206, 2008, Proceedings of the 6th IAEA Technical Meeting on Control, Data Acquisition, and Remote Participation for Fusion Research.

[6] F. Perkins, D. Post, N. Uckan, M. Azumi, D. Campbell, N. Ivanov, N. Sauthoff, M. Wakatani, W. Nevins, and M. Shimada *et al.*, "Chapter 1: Overview and summary," *Nucl. Fusion*, vol. 39, no. 12, pp. 2137–2174, Dec. 1999.

[7] A. Neto, F. Sartori, F. Piccolo, A. Barbalace, R. Vitelli, and H. Fernandes, "Linux real-time framework for fusion devices," *Fusion Eng. Design*, vol. 84, no. 7–11, pp. 1408–1411, Jun. 2009.

[8] *Picmg 3.0 Advanced Telecommunications Computing Architecture Base Standard*, [Online]. Available: http://www.advancedtca.org

[9] A. J. N. Batista, J. Sousa, and C. A. F. Varandas, "ATCA digital controller hardware for vertical stabilization of plasmas in tokamaks," *Rev. Sci. Instrum.*, vol. 77, no. 10, pp. 10F527–10F527-3, Oct. 2006.

[10] J. G. Krom, "The evolution of control and data acquisition at JET," *Fusion Eng. Design*, vol. 43, no. 3–4, pp. 265–273, 1999.

[11] F. Romanelli, J. Paméla, R. Kamendje, M. Watkins, S. Brezinsek, Y. Liang, X. Litaudon, T. Loarer, D. Moreau, D. Mazon, G. Saibene, F. Sartori, and P. de Vries, "Recent contribution of JET to the ITER physics," *Fusion Eng. Design*, vol. 84, no. 2–6, pp. 150–160, Jun. 2009.

[12] D. Valcárcel, A. Neto, J. Sousa, B. Carvalho, H. Fernandes, J. Fortunato, A. Gouveia, A. Batista, A. Fernandes, M. Correia, T. Pereira, I. Carvalho, A. Duarte, C. Varandas, M. Hron, F. Janky, and J. Písacka, "An atca embedded data acquisition and control system for the compass tokamak," *Fusion Eng. Design*, vol. 84, no. 7–11, pp. 1901–1904, 2009, Proceedings of the 25th Symposium on Fusion Technology—(SOFT-25).

[13] P. J. Carvalho, H. Thomsen, S. Gori, U. v. Toussaint, A. Weller, R. Coelho, A. Neto, T. Pereira, C. Silva, and H. Fernandes, "Fast tomographic methods for the tokamak isttok," in *Proc. AIP Conf.*, 2008, vol. 996, no. 1, pp. 199–206.