DAΦNE TECHNICAL NOTE
INFN - LNF, Accelerator Division

Frascati, February 14, 1994
Note: **C-9**

DANTE CONTROL SYSTEM DATA FLOW

M. Verola

1. Introduction

The DANTE (DAΦNE New Tools Environment) Control System is based on a hierarchical architecture made up by three levels:

PARADISE	the operator interface level or first level
PURGATORY	the central monitoring level or second level
HELL	the device interface level or third level

The three levels are linked vertically (from higher to lower levels and viceversa) by dedicated connections.

The Control System computing environment consists of several distributed CPUs, communication hardware (buses, adapters and cables) and memory modules. Every CPU communicates with the upper and/or the lower levels by means of the communication equipment in order to access common data, to send commands towards remote devices and to deliver various types of messages.

The aim of this report is to clarify the data flow related to the basic functions performed by the Control System. *Data flow* refers to the logical and physical path followed by the data flowing through the three levels within the Control System. Let me remark that a clear understanding of the data flow yields a much better understanding of the Control System itself.

First I will introduce the fundamental choices followed in the data management organization, then I will give a logical description of the data flow, and finally I will explore the internal mechanism of the implementation.

2. Data Structures

In this section I will focus on the data structures managed by the Control System, providing a logical description of their utilization and functionality together with their relationship to each of the three architectural levels.

2.1. Implementation strategies

Basically there are two main models to exchange data in a multiprocessor environment:

- *message passing*
- *shared memory*

Message passing is the mechanism of sending data packed in a suitable format from a transmitting CPU to a receiving one over a communication channel. This model is typical of a set of computing systems, each of them owning private memory and linked by means of a communication network. In fact, this model requires that each software application operates exclusively in a private environment which cannot be directly modified by other applications. This assumption implies that any interaction between applications is achieved through an explicit exchange of messages. To ensure message delivery and data integrity a *communication protocol* must be implemented. Protocols give the rules for passing messages, specify the details of message formats, and describe how to handle error conditions. Thus a communication software is required to solve synchronization problems between CPUs, to arrange data to be sent into standard packets with a proper header, to guarantee the delivery of the messages by means of acknowledgment packets and to ensure the correct sequence of processing the packets.

Shared memory is the facility of making use of a single common area of memory which every CPU can access directly to put or get data. This model is straightforward to use and has the interesting property to make available promptly every stored data to every CPU that can access that memory area. Furthermore in our computing environment the stored data do not need any format conversion to be read or written, because they can keep exactly the same format handled by each CPU. In fact all the CPUs are similar and run the same operating system and application software. Format conversion is generally required in the message passing mechanism, where the data have to be packed/unpacked in a special fashion before transmitting and after receiving, generating a communication overhead and increasing the CPU workload. The crucial issue in a shared memory environment is keeping *data coherence*, that is ensuring that multiple instances of the same logical data contain exactly the same values.

In our control system the shared memory option has been chosen to simplify the system architecture and the software complexity. Data coherence is automatically guaranteed since only a single copy of each type of data is kept at a well defined memory location.

2.2. Data structures description

Before dealing with the data organization in the Control System, I will review briefly each level's main tasks and some terminology.

The CPUs at the first level, the higher one, constitute the *consoles*. This is the human interface level, devoted to the interaction with the operators. At this level it is possible to monitor the global machine status and to issue commands to set each attribute of the controlled devices.

At the second level there is a single CPU, named *CARON*, that performs the link between the higher and the lower level CPUs, gating information and checking for invalid elements.

At the third level there are several CPUs, named *DEVILs*, dedicated to interfacing the machine devices.

All the CPUs of the first and the second level are able to access a 256MB memory area; in other words, they can share a common memory called *Virtual Central Memory (VCM)*. The third level CPUs can access only a part of the VCM (3MB), as it will be explained below. The VCM is the main mechanism provided by the system to exchange information; it is treated as standard addressable memory by every CPU and can be accessed directly by running applications.

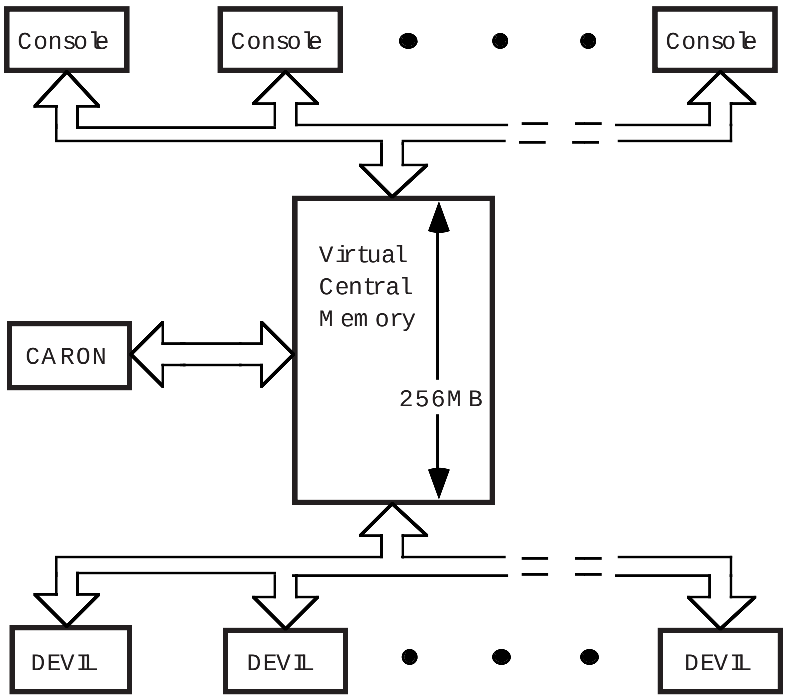

Fig.1. The Virtual Central Memory and the computing environment.
The VCM is directly addressable by all the CPUs to store and/or fetch data.

The VCM is based on top of the physical memory (RAM) of each DEVIL and CARON. These CPUs are equipped with a 4MB RAM memory, but only a 3MB area belongs to the VCM. This limit is due to the fact that the following relation had to be satisfied:

$$N \times M \leq 256MB$$

where N is the number of DEVILs (including CARON) and M is the quantity to be computed, representing the amount of RAM memory that takes part in the VCM. As it has been estimated roughly 70 DEVILs in our system configuration, M was chosen equal to 3MB.

You can imagine the VCM as a 256MB single memory area, that is actually composed by stripes of memory, whose size is 3MB. The set of the RAM memories of the DEVILs and CARON is named *Real Distributed Memory (RDM).*

Summarizing, the RDM is logically mapped in the VCM, hiding to the CPUs the physical location of data.

The VCM contains the following types of data structures:

• *Real Time Data Base (RTDB)*
• *Mailboxes*
• *Service Area*

The RTDB is the container of all the information concerning the machine status. It is formed by the database records describing the various devices controlled by the system. It is a *real time* database because at any time it represents exactly the current machine status and it is always ready and available to be consulted by every CPU. The database records are heterogeneous data structures. Each type of device has a corresponding descriptive record in the RTDB with its own organization. Within a database record there are as many fields as needed to describe exhaustively the related device. Generally each field represents a device attribute to be monitored.

The mailboxes constitute the logical mechanism to send messages between contiguous levels (for more details about mailboxes internal implementation, refer to *DANTE Mailboxes System* [3]). A mailbox is a memory area in which some CPU puts a message and from which another CPU can read that message. It is worth to notice that it is a smart and easy way to implement message passing using shared memory.

A console uses a mailbox to issue a request to the second level, and this one uses a mailbox to send up to the first level informative messages. On the other hand CARON communicates with the third level by means of mailboxes. Mailboxes devoted to the communication between the consoles and CARON are physically located in the CARON's memory. They are twice as many as the number of consoles, because they must handle the communication from a console to CARON and from CARON to console. The same configuration is adopted between the second and the third level: CARON exchanges messages with each DEVIL, making use of a pair of mailboxes, which are physically located in the DEVIL's memory.

The Service Area is used to keep trace of the system activities, recording command logs, warning messages and errors. It is physically located in the CARON's memory.

As a final consideration I wish to highlight that in case of one DEVIL's failure, only the data on its memory become unavailable, without affecting the overall behavior of the rest of the VCM.

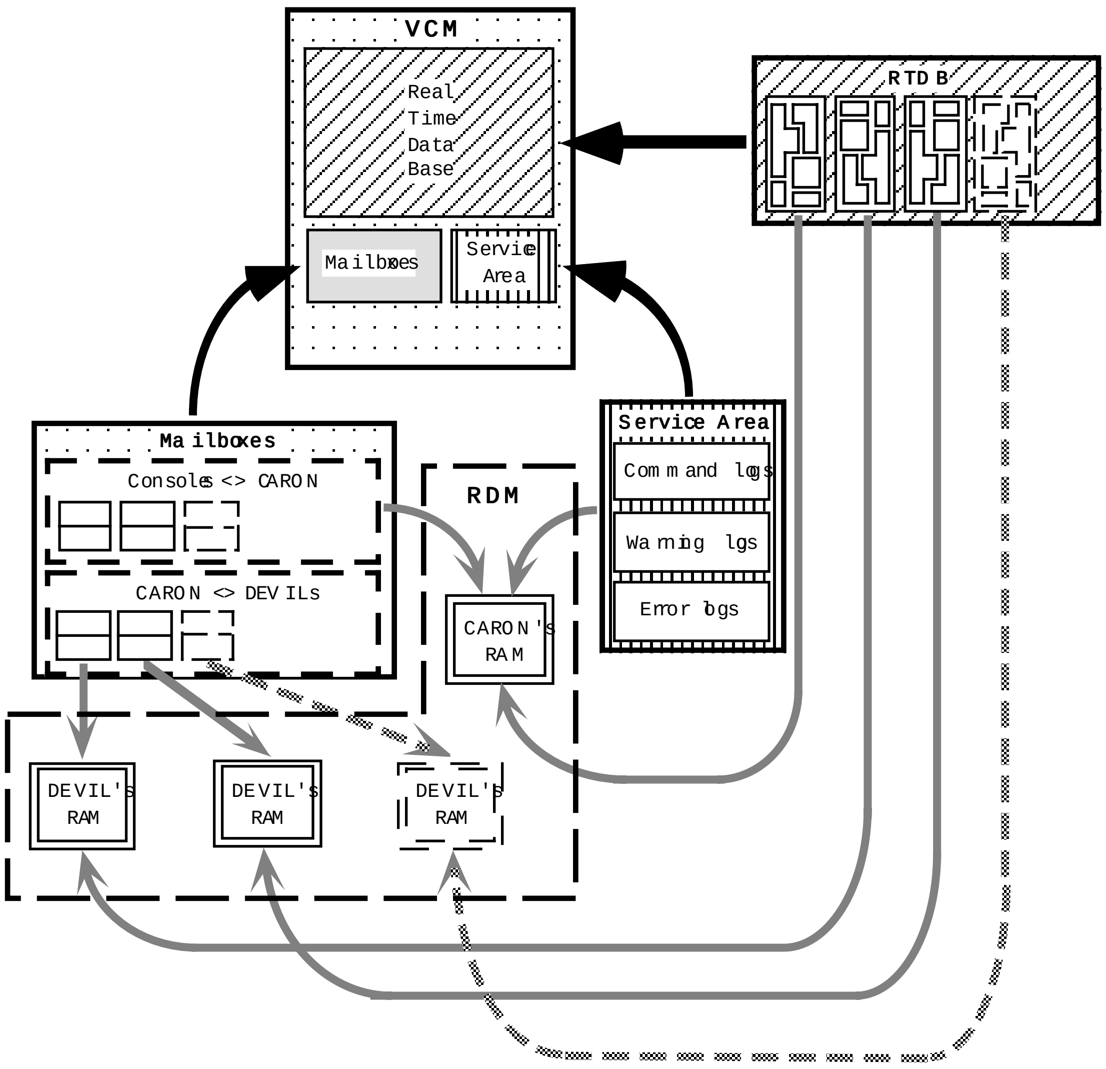

Fig.2. Logical and physical data organization.
Black arrows mean logical mapping whilst gray arrows mean physical mapping.

3. <u>Data Flow</u>

The data flow within the Control System follows two main paths. The first one comes from the controlled devices to the RTDB through the DEVILs: it occurs when a device changes its status, causing the RTDB updating. The second one is activated by issuing a request of setting a device attribute from an operator console.

The amount of data flowing in these two directions is rather asymmetric: data produced by controlled devices are many more than those generated by the first level consoles. However this asymmetry is well balanced by the fact that the first ones are processed and stored in a much faster way since they involve only accesses to local memory, as I will describe in detail in the next subsections.

3.1. <u>RTDB updating</u>

One of the most important features of the Control System is the ability to update in real time the device descriptive records. The logical and physical components involved in performing this task are:

- *controlled devices*
- *links between devices and DEVILs*
- *DEVILs*
- *RTDB*
- *Control Program*

Each DEVIL has statically assigned a set of devices that it must control. In fact a DEVIL is inserted in a third level VME crate, containing several I/O adapter cards for interfacing the controlled devices. Therefore it is responsible for those devices linked to its VME crate and keeps their descriptive records in its RAM. In this way it physically contains that part of RTDB that only itself is allowed to modify and update. This feature prevents from synchronization problems due to multiple simultaneous writings from different CPUs, guaranteeing mutual exclusion. Furthermore it avoids remote memory access from DEVILs through communication links (a DEVIL always writes in its memory), gaining speed in the RTDB updating process.

Each DEVIL runs a software application made up by two concurrent loops:

- *Control Loop*
- *Command Loop*

The former is the subject of this subsection, while the latter will be explained in the next one.

The Control Loop is that piece of software that executes a cyclic monitoring on each device which has to be controlled by the specific DEVIL. At any occurrence of the loop the DEVIL checks if the first device linked to its VME crate has changed any significant value in its representative attributes. If it is so, the DEVIL gets from the device the changed value and consequently updates instantaneously the corresponding field in the device's descriptive record of the RTDB contained in its memory. Just from now this new value is available to all the operator consoles. If the device has not shown any meaningful change, the DEVIL inquires into the second device, performing the same actions, and so on, until it reaches the last controlled device, after that it repeats a new control iteration.

In summary the DEVIL runs the following algorithm in the Control Loop, assuming that it must control N devices (dev[1], ..., dev[N]):

```
1. i = 1
2. Check dev[i]
3. If dev[i] has changed Then
       get the new value and update the RTDB
   Endif
4. i = i+1
5. If i>N Then
       go to step 1.
   Else
       go to step 2.
   Endif
```

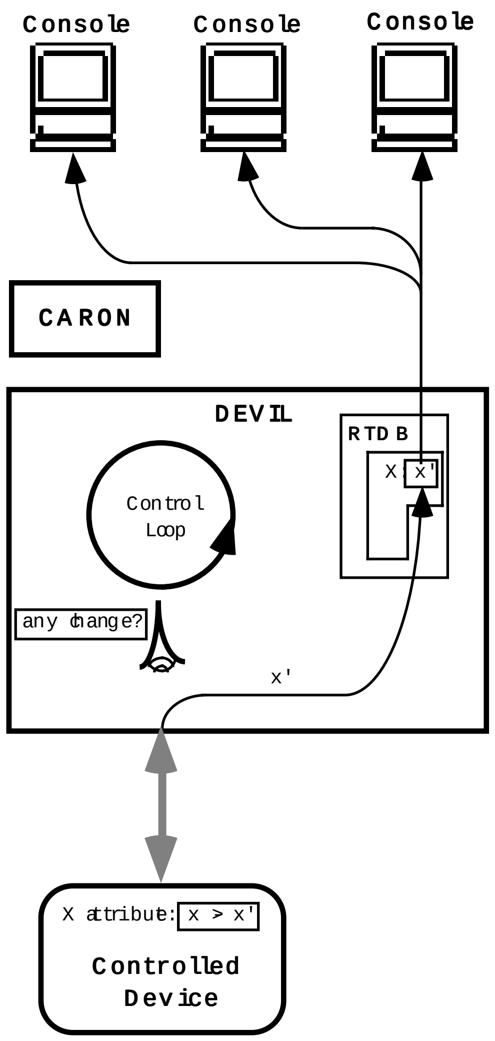

Fig.3. The RTDB update and the Control Loop dataflow.
The DEVIL is responsible to update the RTDB, directly accessed by the consoles.
CARON does not take part in the dataflow.

3.2. <u>RTDB addressing</u>

Once the device attributes have been updated in the RTDB, they become promptly available to be read by the application programs running on the consoles. Data fetching is carried out by means of an explicit access to a well defined memory location where the specified device attribute is stored, within its representative record.

Console programs can read directly data belonging to the RTDB, passing through the second and the third level communication hardware. This implementation does not constitute a real performance bottleneck, because the communication links between different levels, composed by interconnect system adapters inside the VME crate and coaxial or optical cables, are able to ensure more than enough bandwidth for our requirements. For instance you could visualize a plot of a remote oscilloscope in a graphic window on the screen of a first level console; this is a typical situation that requires a high data transfer rate.

3.3. Device Attributes Setting

Even if every console can access directly the RTDB, both for reading and for writing data, it is not allowed (to be precise, it is not advisable) that the console modifies the device attributes writing the desired target value in the RTDB. It has to follow an indirect path, asking the Control System to perform the necessary tasks to obtain the required result. Hence the console has to issue an appropriate request to the second level.

Issuing a request means asking for the execution of a specific action on a peculiar device or a set of devices. Consoles cannot set directly a device attribute, but such an action has to be delegated to the Control System by means of a specific command. This approach prevents from incorrect settings and meaningless actions, because the third level programs are able to check the requests and to act as a filter on the invalid ones.

When an operator wishes to change an attribute from the current value to a new one, it sets the new value in the appropriate graphic field in a user interface window displayed on the console screen at the first level. Then the console software is responsible to take that value and build up a command string to be delivered to CARON.

CARON executes the Command Loop which performs a cyclic inspection of the consoles' mailboxes. When a console puts a message in the proper mailbox, CARON realizes that there is a new incoming request and begins to serve it. First of all CARON extracts the message, parses it and interprets the command string, performing the necessary checks (such as controlling the existence of the target object, that is the device name, in a predefined list). At this point CARON selects the DEVIL associated to the target object and forwards the request, putting a message in that DEVIL's mailbox. The DEVIL, polling on its mailbox, finds the new message, analyzes it and executes the required command by sending to the device a specific stream of bytes. The device receives the command and sets the specified attribute to the new value (or starts an action for reaching that workpoint). Once the DEVIL has delivered the command to the device, it does not have to wait for its completion, but it can continue serving new requests or performing other tasks.

Skipping all the checks for invalid messages and assuming there are K consoles sending messages through their mailboxes (ConsoleToCARON_MB[1], ..., ConsoleToCARON_MB[K]), the algorithm executed by CARON looks like the following:

```
1. i = 1
2. Check ConsoleToCARON_MB[i]
3. If ConsoleToCARON_MB[i] is not empty Then
       - get the least recent message and analyze the command string
       - find out which DEVIL the command has to be forwarded to
       - put the message in the selected DEVIL's mailbox
   Endif
4. i = i+1
5. If i>K Then
       go to step 1.
   Else
       go to step 2.
   Endif
```

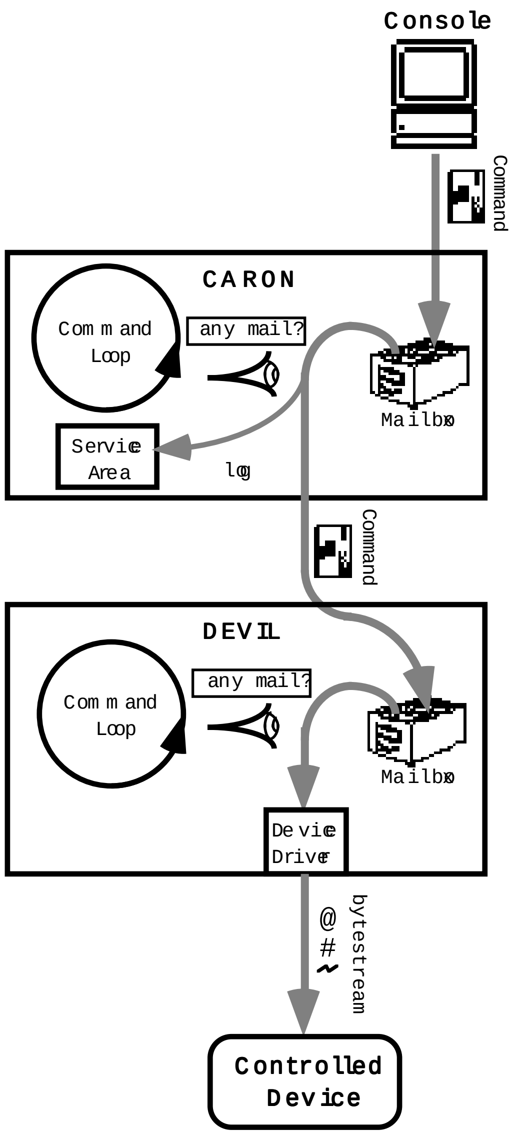

Fig.4. Request issuing and the Command Loop dataflow.
Commands are issued by using messages flowing through mailboxes.

To be precise during this loop CARON is concurrently running another loop, checking the content of the DEVILToCARON_MB[j] (j=1, ...,N, where N is the number of the DEVILs), to detect any error message coming from the third level, as it will be explained in the next section.

In the meantime the j-th DEVIL runs a similar loop:

```
1. Check CARONToDEVIL_MB[j]
2. If CARONToDEVIL_MB[j] is not empty Then
       - get the least recent message and analyze the command string
       - translate the command string in a the corresponding device oriented bytestream
       - send the bytestream to the device through the device driver
   Endif
3. go to step 1.
```

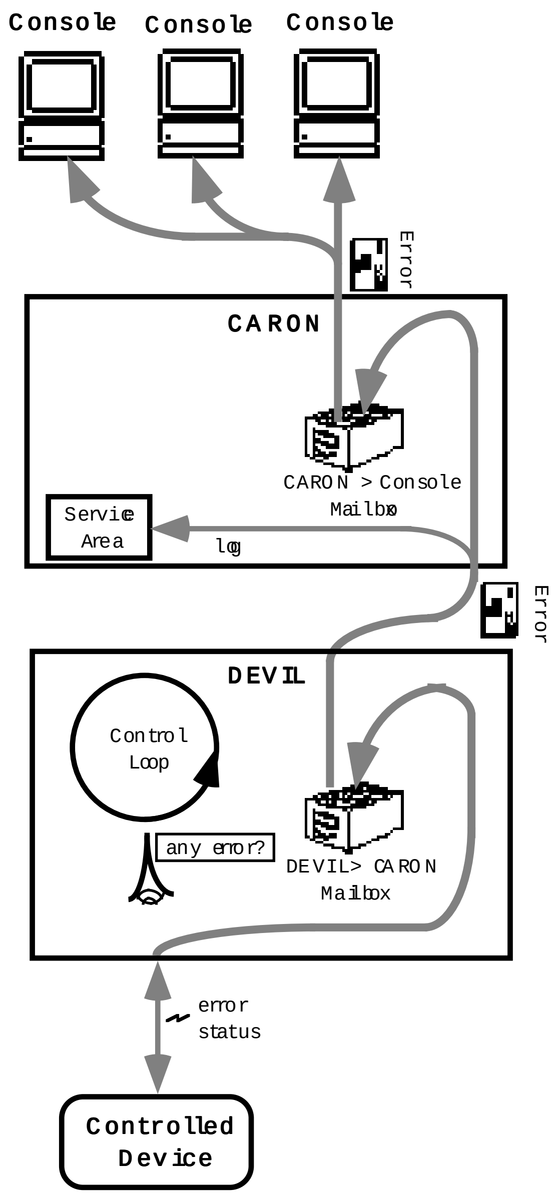

Fig.5. System Logs dataflow.
Error and warning logs are stored in a proper memory area. To simplify the picture the program loops performed by CARON and the Consoles, polling on their reading mailboxes, are not explicitly shown.

3.4. System Logs Recording

During its activity the Control System records any relevant event which could be useful to keep track of. They are:

- *Commands*
- *Errors*
- *Warnings*

When CARON accepts a command issued by a console, it writes a command log record prefixed by a timestamp in the Service Area, physically located in its own memory. On the other hand a command rejected because of an invalid element name specification causes CARON to record an error log in the Service Area and to send an error message back to those consoles that are up and running, using the mailbox mechanism.

In the same way, when a third level DEVIL detects an error condition from a controlled device, it sends an error message to CARON, which in its turn forwards the message to the upper level; also this communication path has been implemented using mailboxes, from the DEVIL to CARON and then from CARON to the consoles.

A warning log example is the notification to the consoles of the "birth" of a DEVIL CPU. When a DEVIL is powered on and becomes "alive", after loading the operating system and the Control System software, a warning message is generated.

Let me remind that the Service Area can be directly addressed and then consulted by the consoles.

4. Summary

A hierarchical architecture joined to a shared memory model are the basic features of the DANTE Control System. It can provide an easy and fast access to a Real Time Data Base and a reliable validation mechanism for checking commands issued towards controlled devices.

Its high modularity prevents from global system crash in case of unexpected failure in some part of it. CARON is the only component which performs a crucial and not replicated task within the system. It might be feasible to duplicate CARON, setting up an identical backup system, running synchronously with CARON, always ready to replace its master copy.

REFERENCES

[1] G. Di Pirro, C. Milardi, A. Stecchi, L. Trasatti, *DANTE: CONTROL SYSTEM FOR DAΦNE BASED ON PERSONAL COMPUTERS AND HIGH LEVEL TOOLS*, presented at the RT93, June 8-11th, 1993, Vancouver, Canada.

[2] Control Group, *DAΦNE CONTROL SYSTEM STATUS REPORT,* minireview held in October 13th, 1993, LNF, Frascati.

[3] G. Di Pirro, *DANTE Mailboxes System,* to be published in DAΦNE Technical Notes.

[4] B. Caccia, V. Dante, G. Di Pirro, C. Milardi, A. Stecchi, L. Trasatti, S. Valentini, *THE DANTE CONTROL SYSTEM*, LNF-92/054 (IR), June 12th, 1992.