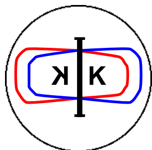

DAΦNE TECHNICAL NOTE

INFN - LNF, Accelerator Division

Frascati, September 30, 1994

Note: **C-15**

DANTE SYSTEM
SOFTWARE DESCRIPTION

M. Verola

1. Introduction

The DANTE Control System is made up of hardware equipment and software programs needed to perform controlling and monitoring tasks on DAΦNE. The software programs can be grouped in three different parts:

- *Human Interface & High Level Software Interface*
- *System Software*
- *Peripheral Device Software*

The Human Interface is the external appearance of the system and enables the operator to interact with underlying services. It uses the *Graphic User Interface* displayed on the computer screens at the first level. The High Level Software Interface is the software platform (*server*) which provides a communication channel towards the Control System services to external applications (*clients*).

The System Software is the heart of the Control System and defines its software architecture.

The Peripheral Device Software consists of all those device-oriented programs and driver routines that talk directly with the controlled elements to send specific commands and to retrieve information relevant to the machine status.

In this note I will describe the DANTE System Software, focusing on those programs that are inherently bound to the Control System (see also [1], [2] for understanding the DANTE logical structure).

2. Software Environment

The System Software runs on a hardware platform made up of several distributed CPUs and comprises many applications which share and exchange data. It has to guarantee the following services:

- fast message delivery
- error handling
- system events logging
- system failures notification and recovering
- on-line database coherence and reliability
- off-line database services.

The Control System programs run under Apple System 7 and are built using G graphical programming language, provided by LabVIEW Software [3]. Some routines for low level and time critical tasks are written in C or in Assembler language.

We have chosen LabVIEW environment because it can speed up software development, yielding high-quality code at reduced manpower. The most important features met by the LabVIEW Software environment are:

- built-in interactive user interface
- extensive library of functions and development tools
- debugging tools
- code reliability
- software development time reduction (respect to other general purpose programming languages, like FORTRAN, Pascal, C, ...)
- self-documenting
- ease to modify and to maintain
- link to external code written in a conventional programming language (Code Interface Node or CIN).

3. Memory Mapping

We use DMA (Direct Memory Access) to exchange information among all the CPUs. The VME memory is the main medium for sending messages, for storing the Real Time Data Base (RTDB) and for housing various data structures in the Service Area [2]. The communication boards installed in VME crates and in the Macintosh slots are configured to perform memory mapping between the VME memory modules and the CPU address space.

There is a pair of LEXTEL LL2000 boards (VMEbus-VMEbus Coupling System) linked by Coax or Fiber Optic Cables, depending on the distance, to connect two VME crates, and a CES MAC 7212 card (Macintosh to VME 32-bit Interface) linked by a flat cable to a CES VIC 8250 board (VMV to VME One Slot Interface) for Macintosh-VME crate communication. There is no software communication protocol because data are stored in shared memory locations, directly addressable by the CPUs. Figure 1 shows a scheme of the DANTE architecture.

Memory mapping is performed by initializing and setting specific registers on the communication boards. CES MAC 7212 provides 256 page descriptors, which are special registers used to map 1MB of VME memory into the Macintosh's address space, for a maximum of 256MB. LEXTEL LL2000 allows up to 4096 separate 4KB memory pages on the local VME bus to be mapped into any 4096 separate 4KB memory pages on the remote bus, for a maximum of 16MB.

The first level Consoles and the single second level CPU, CARON, can address all the VME memory in the system, while the third level CPUs, the DEVILs, can only access their own VME memory (Fig. 2). The Consoles and CARON are Apple Macintosh Computers, while the DEVILs, which are our peripheral VME CPUs, are standard Macintosh LC III logic boards assembled to a custom developed interface to VME, VSB and Ethernet, equipped with 4 additional MBytes of RAM memory (the DEVIL's VME memory). The Consoles do not own VME memory, while CARON borrows 4MB of VME memory (the CARON's VME memory) from a DEVIL CPU installed at the second level. This DEVIL acts as the System Logger, dumping RTDB records and logs (commands, warnings and errors) onto an external SCSI magnetic or optical disk.

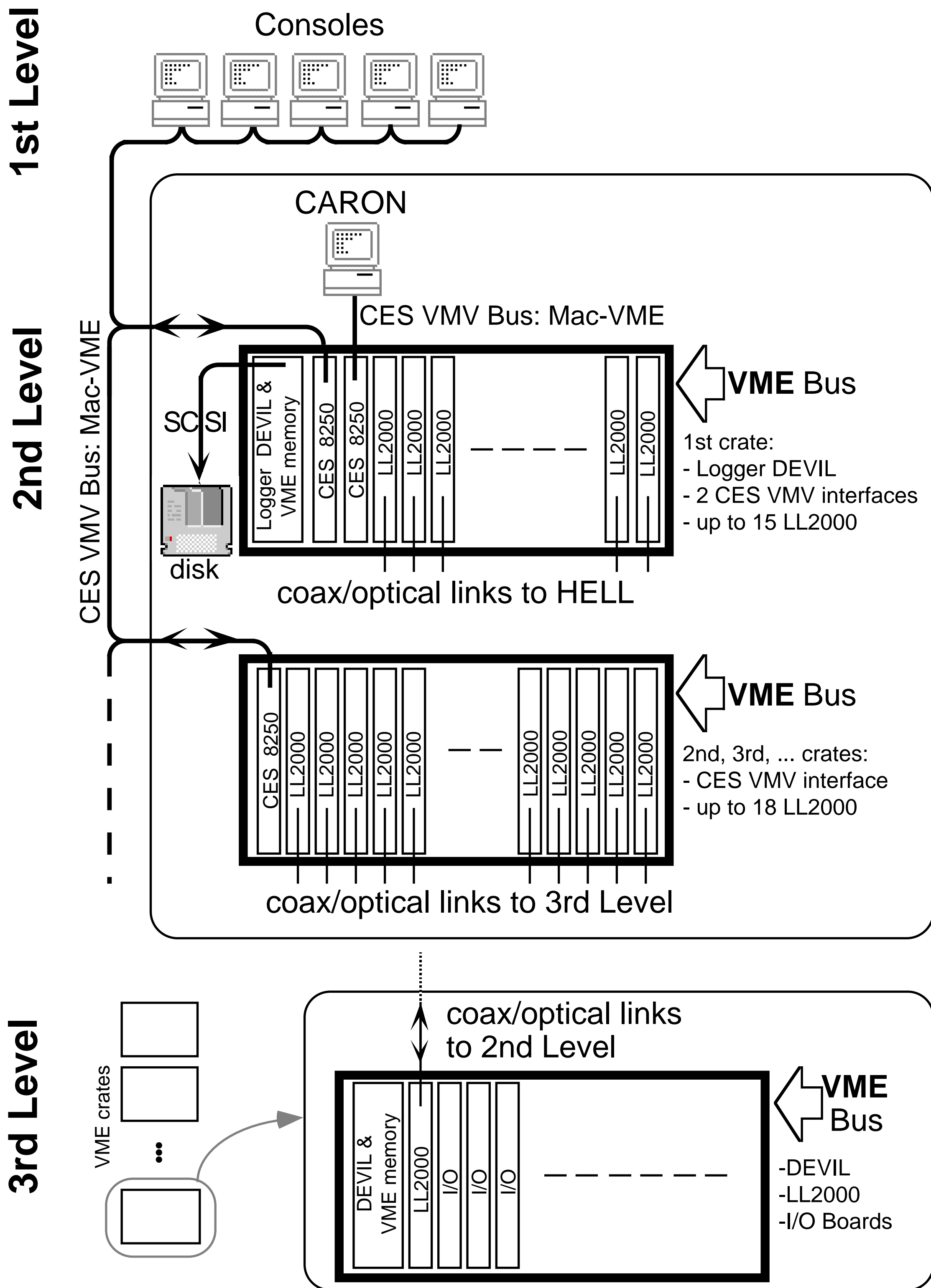

Fig. 1 - The DANTE Control System architecture.

The Ethernet network and the links to second level remote crates are not explicitly shown.

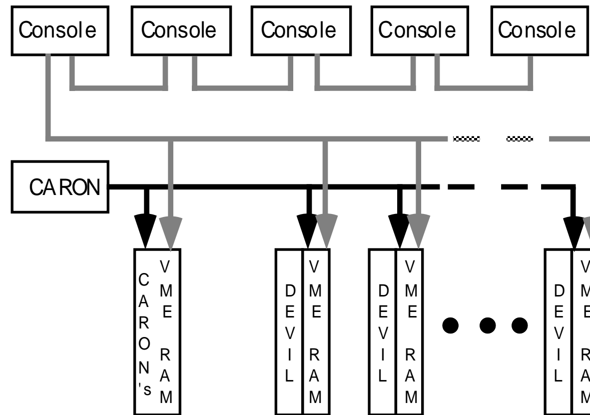

Fig. 2 - The Consoles and CARON accessing the VME memory.

Within this system architecture we have followed the general rule that each level is able to access all the VME memory of the lower levels, but not vice versa, realizing a hierarchical memory mapping.

Assuming that C_{add} is the base address of CARON's VME memory (4MB), Consoles and CARON have the following table of addresses (note that each DEVIL apportions only 3MB of its 4MB of VME memory to the Virtual Central Memory):

start	**end**	**size**	**description**
C_{add}	$C_{add} + \$003FFFFF$	*4MB*	*CARON's VME memory*
$C_{add} + \$00400000$	$C_{add} + \$006FFFFF$	*3MB*	1st *DEVIL's VME memory*
$C_{add} + \$00700000$	$C_{add} + \$009FFFFF$	*3MB*	2nd *DEVIL's VME memory*
•••••••••••••••	•••••••••••••••	••••	•••••••••••••••••••••
$C_{add} + \$00400000 + (i-1) \times \$00300000$	$C_{add} + \$00400000 + (i-1) \times \$00300000 + \$002FFFFF$	*3MB*	i^{th} *DEVIL's VME memory*

In this way all the VME memory in the system is addressable at contiguous addresses, even though it effectively resides on different memory boards on different VME crates.

The existing Ethernet network linking all the CPUs is mainly devoted to service tasks, like CPU remote bootstrapping, data uploading and downloading, severe error notification, software maintenance and program debugging. The communication protocol running on Ethernet is AppleTalk.

4. The System Software

It is worthwhile to divide Control System programs in relationship to the system levels:

- *First Level Software*
- *Second Level Software*
- *Third Level Software*

4.1. The First Level Software

The first level programs, running on the Consoles, consist of a common software kernel and many peculiar graphic interfaces. All the Consoles are equivalent respect to the running software and the provided services. We can generalize the structure of the software, by identifying three main blocks:

- *Graphic Interface*
- *Common Algorithmic Kernel*
- *Interactive Window Navigation System*

The Graphic Interface displayed on the Console screen depends on the element or set of elements that are being controlled and on the types of data that are being observed; hence it is tightly tailored to the peculiar features of the controlled elements. Furthermore there are different levels of interfaces, depending on their purpose: an *Expert Interface* for low level monitor, maintenance and machine optimization, and an *Operator Interface* for standard machine operations. An authentication mechanism will be implemented, based on personal identification codes, to ensure restricted access to the Control System services. Finally the graphic interface is built up by many customized graphic panels, which constitute *virtual instruments,* showing buttons, knobs, charts, graphs, plots, switches, sliders and all kinds of graphic objects that simulate on the Console screen the real *indicators* and *readouts* of the controlled element.

The Common Algorithmic Kernel is implemented by a Monitoring Loop performing two sets of tasks, related to the data flow direction. One set is for sending data (*messages*) from the Consoles to the DEVILs through CARON and includes an *Input Interface*, a *Command Builder* and a *Message Sender*. The other retrieves data (*Element Descriptive Record* stored in the RTDB) from the DEVIL's VME memory to visualize them on the Consoles, using a customized *RTDB Reader*, and looks for incoming messages from lower levels by means of a *Message Manager*.

The Input Interface is the whole of the graphic window with the underlying *Event Manager* routine, which checks if any kind of input has been performed, like, for instance, pushing a graphic button with the mouse pointer. When an input event is detected, a proper command message is built by the Command Builder and then the message is sent to CARON by the Command Sender using the Mailbox mechanism (Fig. 3).

The RTDB Reader is a recursive code (written in C language and linked as a CIN) which is able to read any element descriptive record stored in the VME memory, depending on the type descriptor (Fig. 4). Furthermore the Message Manager is responsible for handling error and warning messages coming from lower levels, displaying a concise description of the problem and possible actions to fix it.

The Interactive Window Navigation System allows to move within the large number of graphic interface panels and options by selecting fields in popup menus or by choosing among a set of icons.

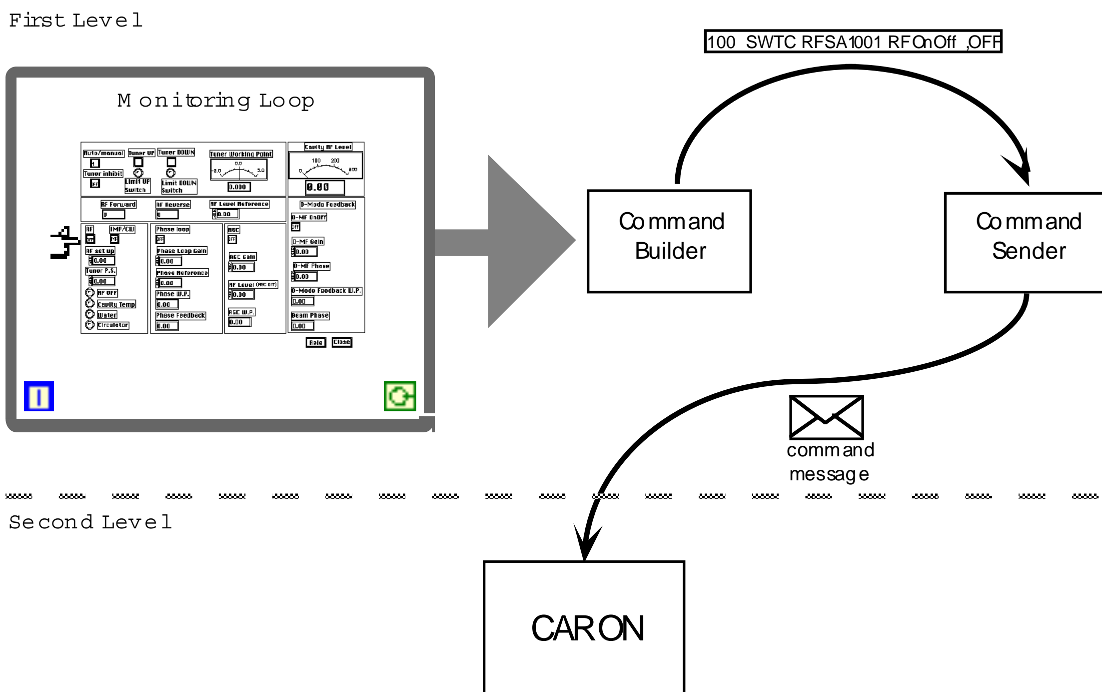

Fig. 3 - The Graphic Interface and the Common Algorithmic Kernel.

The procedure of command building and sending.

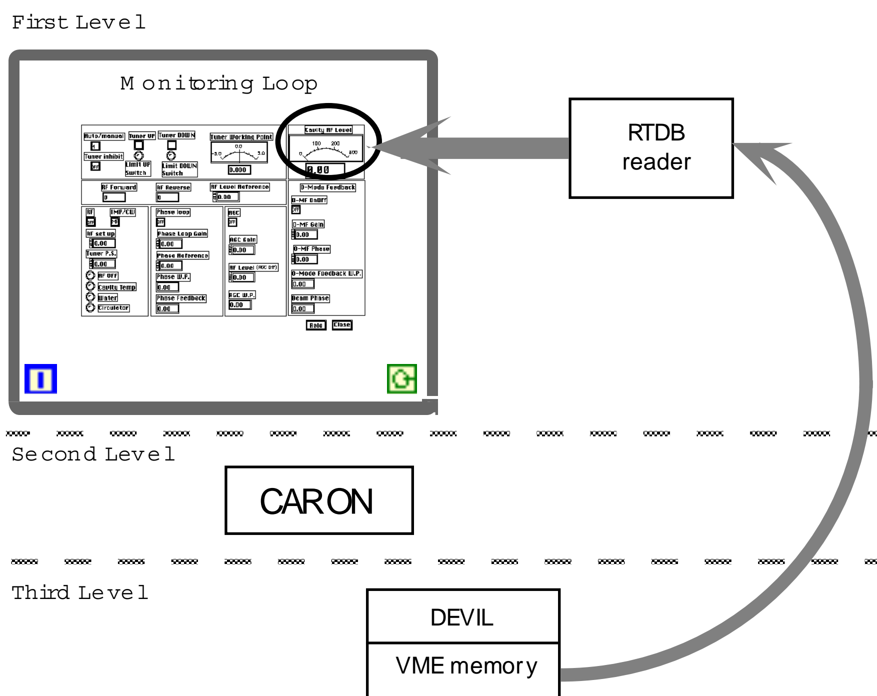

Fig. 4 - The Graphic Interface and the Common Algorithmic Kernel.

The visualization of RTDB element descriptive records.

4.2. The Second Level Software

The second level is formed by a single CPU, CARON. It is the heart of the control system, because it is the central node for the data flow, routing requests coming from the first level to the peripheral CPUs. It also relays to the Consoles the problem notification generated by DEVILs.

4.2.1. The CARON's Program

CARON runs a program that is made of some initialization tasks and a never-ending Main Loop.

The initialization tasks are needed to configure the hardware (communication boards) and for setting specific registers for memory mapping. At this phase the global memory map is filled in the boards' registers, so that CARON can address all VME memory in DANTE.

After configuring the hardware, CARON initializes and loads the starting values of those RTDB data structure that must be stored in its VME memory:

- Consoles' ALIVE Status Array
- DEVILs' ALIVE Status Array
- Log Areas (Command, Error, Warning Logs)
- ConsoleToCaron Mailboxes

Finally CARON begins the never-ending loop. The loop is configured as two symmetric blocks plus the ALIVE check call. At each loop iteration a single Console and a single DEVIL are checked in order to detect incoming messages. Two cyclic indexes keep memory of the current console and the current DEVIL. The sequence of instructions at each loop can be summarized as follows:

```
If the Console is ALIVE Then
        read the Console's Mailbox
        If the Mailbox was not Empty Then
                decode the message
                If the message is a valid command Then
                        forward the message to the proper DEVIL
                Else
                        send error to the Console
                Endif
        Endif
Endif

If the DEVIL is ALIVE Then
        read the DEVIL's Mailbox
        If the Mailbox was not Empty Then
                decode the message
                If the message is a valid error/warning Then
                        forward the message to the proper Console(s)
                Else
                        send error to every Console
                Endif
        Endif
Endif

If ALIVE timeout has occurred Then
        call ALIVE check routine
Endif
```

Figure 5 shows a detailed flowchart of the CARON's program main loop.

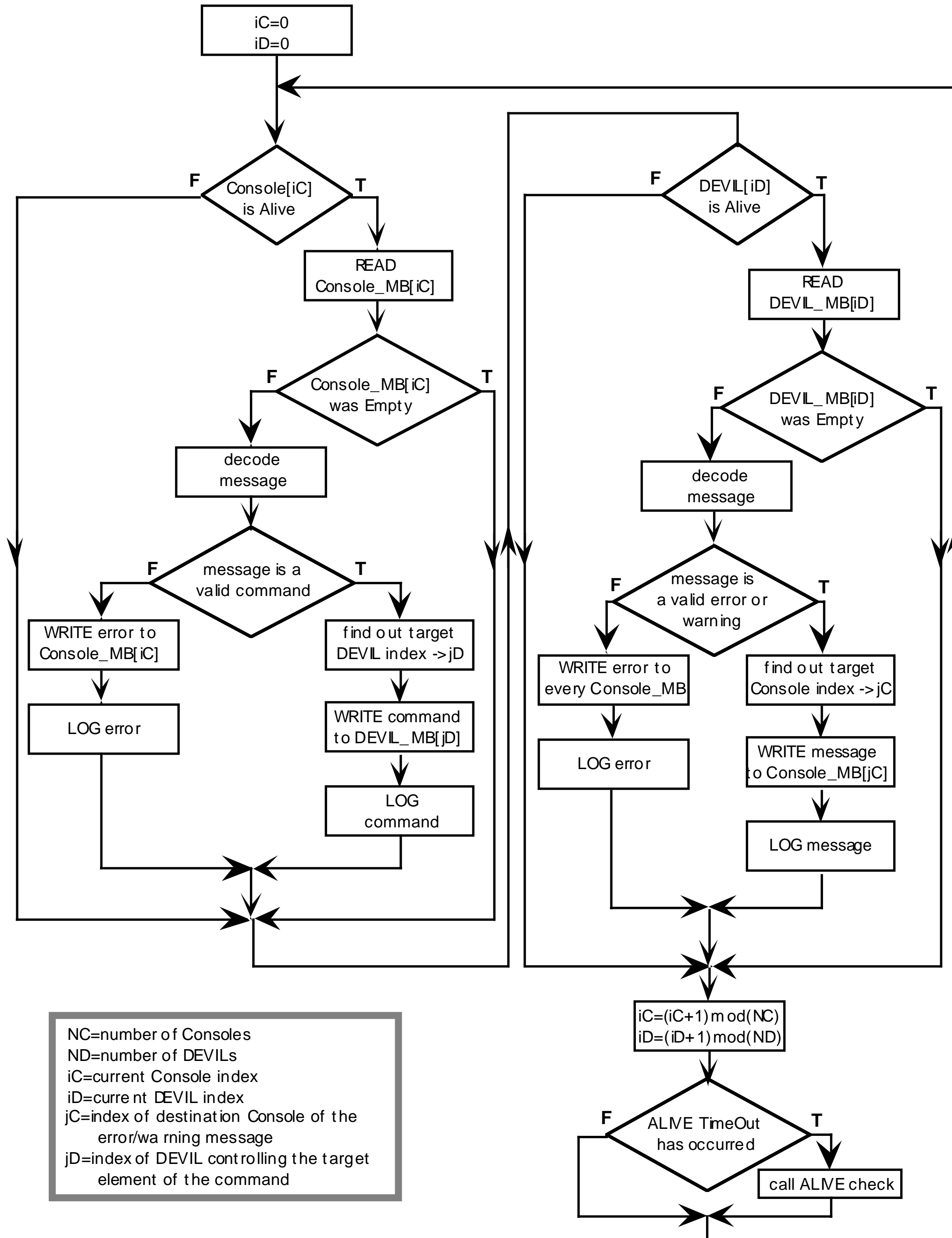

Fig. 5 - The CARON's program main loop.

The ALIVE check routine at each call determines the status of a pair of CPUs in turn, a Console and a DEVIL, by comparing the value of the ALIVE Counter at time *t* (the present call time) and *t-1* (the previous call time): if they are equal it means that the CPU is not working, because it has not increased the ALIVE Counter in the period between two subsequent calls. The ALIVE Counters are memory locations held in CARON's VME memory (Consoles' ALIVE Counters) or in each DEVIL's VME memory (DEVIL's ALIVE Counter).

All the routines that read/write data from/to VME memory, handle the error condition due to failure on the VME bus (BusError). In order to get rid of those errors due to rare events, we always reserve a second try when accessing VME memory before assuming that a BusError condition has been detected. In this case an error message is promptly sent to the consoles. Then the CPU, which owns that VME memory, is put in a faulty state, the BusError status. This is done to prevent CARON from further attempts of reading/writing that VME memory, avoiding to get an error message each time. Thus a previous check on the owner CPU status is always executed whenever a VME memory access is needed.

When a BusError condition is detected in the VME memory of CARON itself, the program stops executing after sending the last error message to all the Consoles through the Ethernet network. Possible paths for getting a BusError in CARON's VME memory are:

- accessing Consoles' Mailboxes (read/write)
- updating the ALIVE Status Arrays (write)
- reading the Consoles' ALIVE Counters (read)
- logging command/error/warning messages (write)

4.2.2. Startup of the ALIVE Mechanism

As all the CPUs boot and start running asynchronously, we had to face the problem of guaranteeing a reliable startup mechanism of the ALIVE mechanism, without incurring in BusError condition due to a DEVIL's VME memory not yet initialized.

We have defined four possible codes to qualify the CPU status:

Status	Description
NotInit	CPU has never initialized its VME memory
Dead	CPU has stopped
Alive	CPU is running
BusError	CPU's VME memory is in BusError condition

When CARON program starts, all the DEVILs are put in NotInit status, while Consoles are considered in Dead status. This permits to solve gracefully synchronization problem at booting time among CARON and the first and third level CPUs. In fact CARON can check from the beginning the Consoles' ALIVE Counters without having BusError condition due to VME memory not yet initialized, because it is reading its own VME memory, that was previously configured. On the other hand, CARON needs to read external VME memory (DEVILs' ALIVE Counters), but does not know if the owner CPU has already performed memory initialization tasks; so our solution is to go on reading the DEVILs' ALIVE Counters located in DEVIL VME memory, even though it gets a BusError condition. When CARON reads the ALIVE Counter value successfully for the first time, the DEVIL is put in Dead status. Then, when CARON detects that the ALIVE Counter begins to be increased, the CPU jumps into the Alive status and CARON sends a warning message (*CPUstart*) to all the Consoles. If a BusError condition occurs while CPU is Dead or Alive, the CPU is put in BusError (that is out of order!) and no longer examined. In fact it is very likely that a serious problem has occurred in its VME memory or in the crate VME bus or in the communication cables, and human intervention is required to solve the accident. Figure 6 shows a scheme of all the possible transitions between the different states.

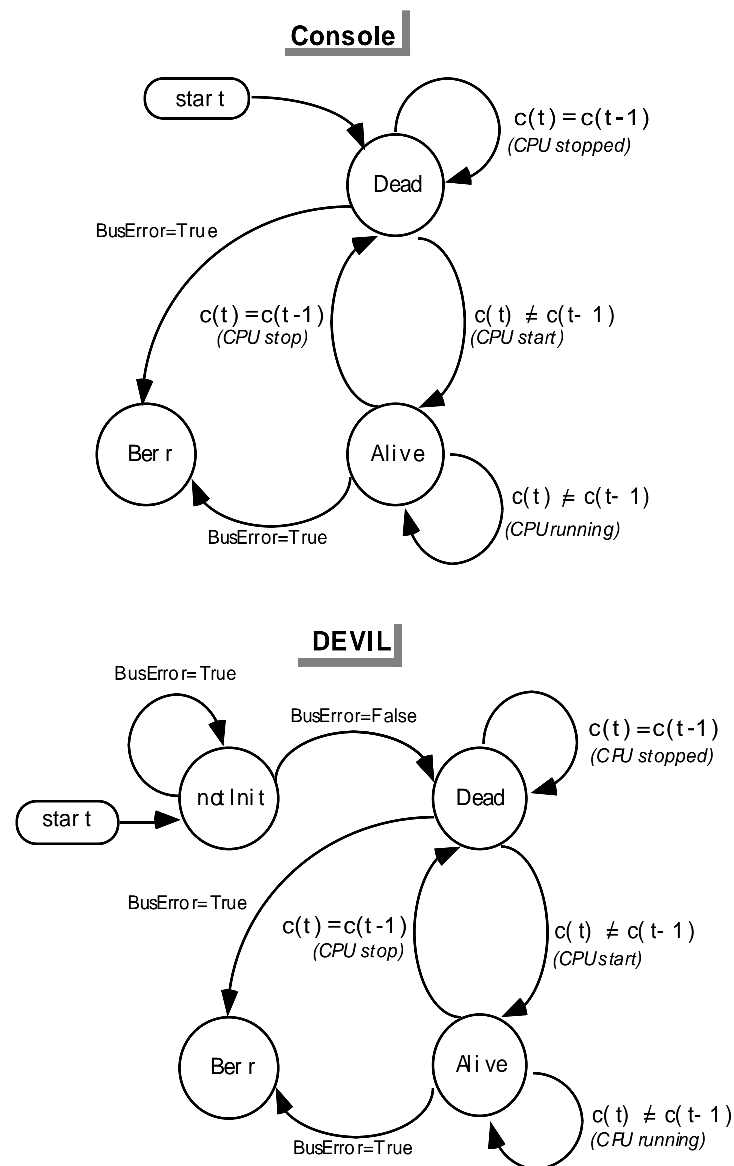

Fig. 6 - The CPU state transition scheme.

4.2.3. Command Relaying

CARON is responsible for delivering the command messages coming from the Consoles to the proper DEVIL. It uses a table to detect which is the DEVIL controlling the element that is the target of the command. If the element is not found in CARON global element list an error message is generated and sent to the console issuing the command. CARON does not perform any check for command validation because this task is under responsibility of the DEVIL (see section 5 for message format and naming convention).

CARON writes the command message in the DEVIL's Mailbox. It is worth noticing that the mailbox mechanism implemented to exchange messages between different levels is not affected by deadlock problems due to simultaneous shared memory access. In fact the process of writing a message in a Mailbox by the sender never interferes with the process of reading a message by the receiver. Both of them work in the same memory area but on different addresses [4].

4.3. The Third Level Software

The Third Level programs are formed by a common kernel, the DEVIL's Main Loop, and a pair of customized routines for each class of controlled elements, the Command and the Control Routines.

4.3.1. The DEVIL's Main Loop

Before describing in detail the sequence of tasks of the DEVIL's program, it is necessary to mention the logical organization of the elements into classes. A controlled element is a physical device installed on DAΦNE that has to be monitored and driven in a remote way by the Control System. There could be some exceptions to this definition, because there are some elements in the Control System that do not correspond to any physical apparatus, but they identify logical groups of physical devices. However this case does not influence the following explanation.

Each controlled element is qualified by a class, a physical location and a numeric identifier. A class is represented by a class name, which is a three characters code: for instance, kickers belong to KCK class, while RF cavities to RFS class. The physical location is encoded into a pair of chars: TL stands for Transfer Line, SR for Storage Rings, and so on. Finally the element numeric identifier is a three digits decimal number (starting from 000), that distinguishes similar elements within their class.

The DEVIL's program starts initializing the hardware (VME memory, interface boards in the crate) and then runs the Main Loop. The Main Loop is a never-ending repetitive task, which performs three main actions at each iteration: reading the mailbox, executing a command (if any), and controlling an element, plus the ALIVE Counter updating (a simple writing of the loop counter value in a well-defined VME memory location). The following pseudo-code highlights the sequence of operations:

```
read the Mailbox
If the Mailbox was not Empty Then
        decode the message
        If the message is a valid command Then
                    append the command to the Progress/Wait Queue
        Else
                    send an error to the Console issuing the command
                    (via CARON)
        Endif
Endif

If the ProgressQueue is not Empty Then
        get next command from the ProgressQueue execute the command
        If the command execution has been completed Then
                    remove the command from the ProgressQueue
                    If any command from the WaitQueue is available Then
                                move the command from the WaitQueue to the
                                ProgressQueue
                    Endif
        Endif
Endif

control the next element

update the ALIVE Counter
```

Figure 7 shows a detailed flowchart of the DEVIL's Main Loop.

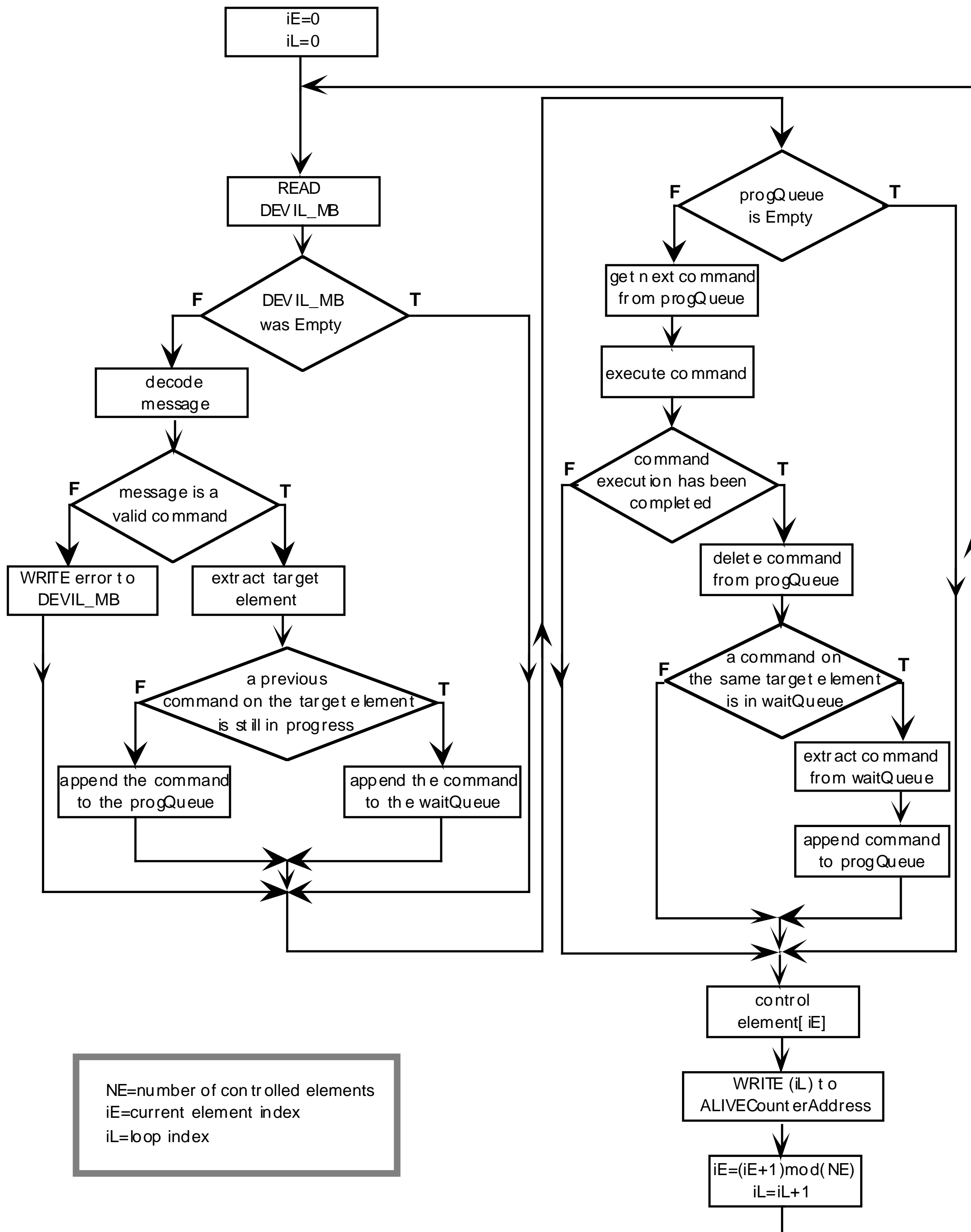

Fig. 7 - The DEVIL's Main Loop.

4.3.2. The Command Queues

Commands come to the DEVIL in the form of messages. They are translated from string format to a set of scalar values and indexes (identifying the console issuing the command, the class and the identifier of the element involved, the type of command) followed by a variable length string field used for specific parameters. All these fields are packed into a record. This is done because commands are not immediately executed, but they are inserted as packed records in one of two command queues, the ProgressQueue or the WaitQueue. The ProgressQueue holds those commands that are ready to be executed or are being executed, while the WaitQueue contains the rest of delivered commands (Fig. 8). The ProgressQueue and the WaitQueue adhere to the following rules:

- in the Progress Queue there cannot be two commands with the same target element
- when a command has been completely executed, it has to be removed from the Progress Queue
- every time a command is removed from the Progress Queue, another command on the same target element (if any) can be transferred from the Wait Queue to the Progress Queue.

The two command queues have a finite maximum dimension, which can be configured as a static parameter. They are implemented as double linked list to speed up the process of appending and extracting items dynamically.

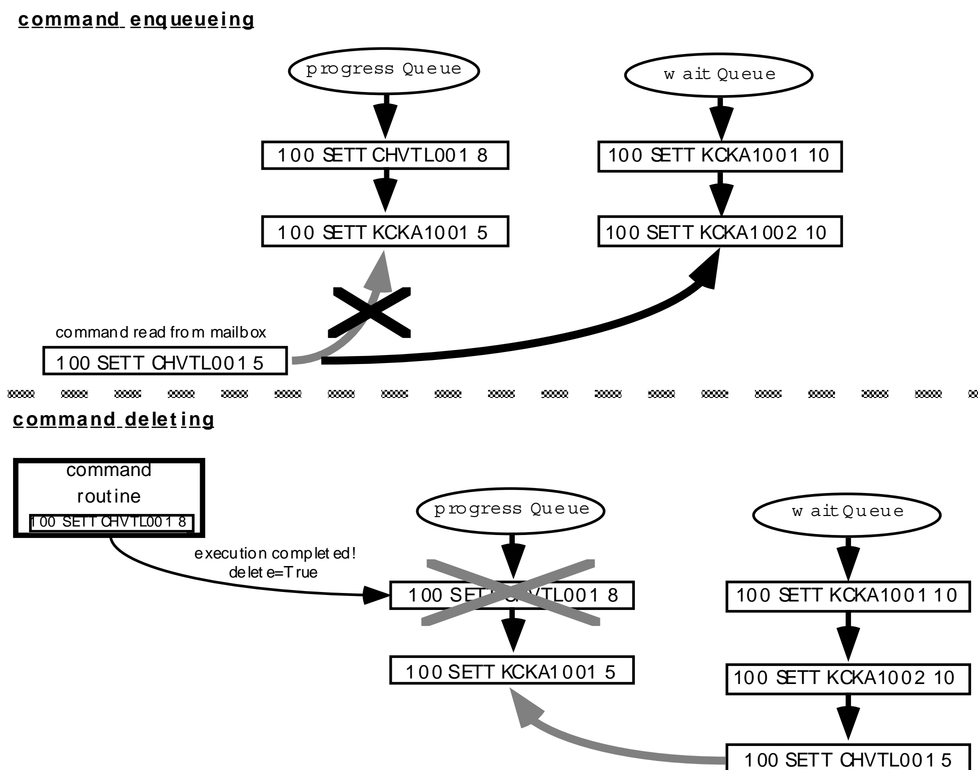

Fig. 8 - The ProgressQueue and the WaitQueue.

The procedure of inserting a new command and the procedure of deleting a command completely executed.

4.3.3. Element Reservation

To prevent competition problems in the use of an element between different Consoles, we have conceived the *element reservation* mechanism. When a command coming from a Console is being executed on an element, this element is automatically reserved to that Console. It means that only the reserving Console can go on issuing commands on the element, while the others are inhibited (they will get an error message from the DEVIL controlling the element, if they try to send commands). To make the element available again, a specific command, RELEASE, has to be issued from the reserving Console. The RELEASE command is however automatically sent when the operator stops working on that element by quitting the graphic interface window on the screen of the Console. The DEVIL carries out element reservation by setting the Console name in the dynamic data of the element descriptive record stored in the RTDB.

4.3.4. Control and Command Routines

There are a Control Routine and a Command Routine for each class. As the first three chars of the element name represent its class code, the DEVIL's program is able to run the proper Command/Control routine for that element. It is difficult to draw a general scheme for a common algorithm, because they are very specialized and strictly depending on the peculiar features of the controlled element. In every case the Command routine is roughly formed by a set of cases corresponding to the set of command types, the *services*, that it can support, while the Control routine is a global reading of the registers of the hardware interfacing the element to retrieve, at each call, the values of the dynamic data describing the element.

5. Message Format

All messages in the system are divided into three categories:

- commands
- warnings
- errors

Commands are defined as the codification into a proper string of actions which must be executed by a controlled device, like moving the current value of a magnet.

Warnings match those kinds of events that are noticeable, but do not require any human immediate decision or intervention. They are informative messages, like the start of a CPU.

Errors are notification of problems involving human decision and intervention.

Command messages can only flow from the first level to the third level through CARON, while errors and warnings go in the opposite direction, from the third level and second level to Consoles. Commands are generated at the first level by working with interactive control panels or by requests coming from the High Level Software Applications.

The following table shows the exact format and syntax of the three kinds of messages, together with their paths. The length of the fields are enclosed within parenthesis (an asterisk (*) means variable length), while optional fields are surrounded by brackets.

Type	Path	Format
Command	Console->CARON CARON->DEVIL	<fromCPU(3)> <service(4)> <element(8)>[<parms(*)>]
Error	DEVIL->CARON	'ERRO' <fromCPU(3)> <toCPU(3)> <errCd(16)> <errLc(16)>[<parms(*)>]
Error	CARON->Console	<ts(17)> • 'ERRO' <fromCPU(3)> <errCd(16)> <errLc(16)>[<parms(*)>]
Warning	DEVIL->CARON	'WARN' <fromCPU(3)> <toCPU(3)> <errCd(16)> <errLc(16)>[<parms(*)>]
Warning	CARON->Console	<ts(17)> • 'WARN' <fromCPU(3)> <errCd(16)> <errLc(16)>[<parms(*)>]

We have chosen to arrange messages in a fixed format (with an exception for the last optional parameters field) to simplify string parsing and to use smart techniques for searching specific fields. Figure 9 shows two examples: a command and an error message.

(a)
100 SETT KCKA1001 10
issuing Console
service
element
optionalparameter

(b)
ERRO 300 100 serviceNotFound DEVILdecodeCmd {100 SWTC KCKA1001 ON}
constant string
source CPU
destination CPU
error code
error location
optional parameters

Fig. 9 - Example of messages.

A command message (a) and an error message (b) with the indication of their fields.

A command may require the specification of one or more additional parameters to be fully characterized. To leave maximum flexibility in implementing command routines for heterogeneous devices, we have provided an optional variable length field, the *command parameters*, at the end of every command message.

CARON is the time reference in the System, in the sense that all those events that need to be recorded in a chronological table for logging or debugging purposes, are time-stamped by CARON (like messages delivered to Consoles and the sequence of system logs in the Service Area).

5.1. Naming Convention

We have adopted the convention of giving a name to every object in the System, like CPU and controlled elements.

Every CPU in the System is identified by a three digits name:

- Consoles 100, 101, 102,...
- CARON, Logger DEVIL 200, 201
- DEVILs 300, 301, 302,...

The name of the Console issuing the command is inserted in the command string for two reasons:

1) when the DEVIL reserves the target element before executing the command, it has to know which Console issued the command;

2) in case of an error condition, the DEVIL has to send back an error message to the Console that issued the command.

Each element has a unique name (8 chars) and belongs to a class (Fig. 10). Element names are stored internally and in the RTDB as double float (size of DBL = 8 bytes) to reduce time spent in searching and during data exchange between routines (scalars can be managed faster than strings in LabVIEW). Each class (encoded by 3 chars) has a set of services (4 chars) associated to itself, which are the types of commands that the class can support (Fig. 11).

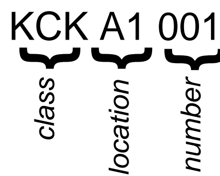

Fig. 10 - Element name format.

class	***services***								
RFS	RELE	SETT	SWTC						
KCK	RELE	SETT	MODE	DELY	POWR	TRIG	RMTE	ONLN	BYPS

Fig. 11 - Element Classes and Services.

5.2. Error Codes and Error Locations

To make the understanding of system problems and failures easy, a consistent error handling mechanism has been carefully studied and implemented.

Error types are encoded in a 16 chars string, internally cast to complex double number (size of CDB = 16 bytes) to speed up system performance. Error codes are part of the error message, in conjunction with the Error Location. Error Locations have the same format of error codes and represent the identification of the piece of program or the specific action that was being executed when error occurred. They have been introduced to give a more precise hint to make problem resolution faster. In fact there are several routines which are called in many parts of the System Software. An error generated during their execution is a poor help if there is no way to know where that routine was called. At the first level a complete table of all the system error and location codes will be accessed by a utility program to show an extensive description of the problem with an indication of some possible action to solve it. Excerpts from the Error Code Table and the Error Location Table are shown below.

Table of Error Codes

Error Code	*Error Description*	*Parameters*	*Source File*
ALIVEUpdateBerr	Bus error when writing the alive status array to VME RAM.	VME write address	aliveUpdate.vi
serviceNotFound	The service is not implemented for the specified element.	{Command string}	commandDecode3.vi
CPUstop	Console/DEVIL CPU has stopped running.	CPU name	Born&Dead.vi
badElementName	Command coming from console contains an unknown element name.	{Command string}	sendCommand2>3.vi
MBFull	The mailbox has not enough space for storing the message.	Mailbox address, {message string}	writeMB.c
....	....	....	....

Table of Error Locations

Error Location	*Location Description*	*Source File*
CARONreadDEVILMB	CARON is reading a message from DEVIL mailbox.	CARON.vi
CARONBorn&Dead	CARON is checking CPU alive status.	Born&Dead.vi
DEVILdecodeCmd	DEVIL is decoding a command string.	commandDecode3.vi
....	....	....

6. Conclusions

The DANTE System Software has been completely reviewed and standardized in all those parts that do not depend directly on specific features of controlled devices.

The First Level Software architecture has been decided and implemented in its basic blocks. Because of the peculiarity of each graphic interface related to a specific controlled element, only a set of general criteria and statements of common software writing style has been established.

CARON's programs can be considered in their definitive version, apart from new features that will be required in the future.

The DEVILs' software has been frozen for those aspects that concern the common basic algorithm driving the sequence of tasks performed and for the interface towards the specialized low level routines (flexible mechanism to embody new pieces of code, standard format for input/output parameters).

A strong emphasis has been dedicated in clearly defining message categories and in establishing simple but exhaustive format and syntax for all of them.

A common error naming convention and definition has been adopted to facilitate debugging and system failure recovering tasks.

REFERENCES

[1] G. Di Pirro, C. Milardi, A. Stecchi, L. Trasatti, *The DANTE Control System*, DAΦNE TECHNICAL NOTE, C-6, INFN-LNF, Accelerator Division, Frascati, June 10, 1992.

[2] M. Verola, *DANTE CONTROL SYSTEM DATA FLOW*, DAΦNE TECHNICAL NOTE, C-9, INFN-LNF, Accelerator Division, Frascati, February 14, 1994.

[3] LABView® National Instrument Corporation, 6504 Bridge Point Parkway, Austin, TX 78730-5039.

[4] G. Di Pirro, M.Masciarelli, M. Verola, *The DANTE Mailbox system*, DAΦNE TECHNICAL NOTE, C-12, INFN-LNF, Accelerator Division, Frascati, March 31, 1994.