The SPS RF Control System
Report & Reply, Archives

Y. Pilchen

Edited and updated
by Frode Weierud

August 13, 2002
Abstract

The replacement of the old Control System for the SPS RF (NORD 100 computers and NODAL programs) is now complete. The new Control System has been installed during the shut-down of winter 1994, and is now extended to all SPS RF equipment.

This technical note describes how the new system deals with Reports, Replies and Archives.
Chapter 1

SPS RF Control System

1.1 Introduction

Four different processes have been implemented in this application. *rptrply* and *hprr* allow the communication between the user interface (MMI) and the equipment. *actif* and *archive* store the settings written in those equipments so that you can easily consult them and save new settings in archives.

This report is divided in five chapters.

- Chapter 1.2 describes the four processes in the entire SPS RF control system.
- Chapter 2 deals with general points concerning the programming environment. It describes the division of the whole application into directories and explains how the binary files are created.
- Chapter 3 describes report and reply processes both in the DSC (*rptrply*) and on the HP workstations (*hprr*).
- Chapter 4 describes archive processes (*actif* and *archive*). In these last two chapters, the use of main globals is given and every major procedure is fully detailed.
- Chapter 5 describes important practical points.
1.2 SPS RF Control system

1.2.1 Three different layers

Three different layers are involved in the control application.

G64

G64 racks control various types of equipments directly.

Device Stub Controller (DSC)

Each set of equipment is gathered in a system such as RF_Synchro, TWC_200_Low_level . . . A DSC is able to control several systems but one system is controlled by a single DSC. Each DSC runs kernel, rptrply, bufferman, and every exec linked to the systems that the DSC controls.

HP workstations

Users are able to use the MMI interface from any HP terminal connected to the network. The two processes bulles and hprr which go together can be duplicated several times on the same workstation. Archive administration is managed by the two processes actif and archive only. They run on rfsrv1 which is the operational workstation for the RF.

1.2.2 RPC services

In the DSC, we have been compelled to use shared memory with a private buffer manager. This buffer manager allows communication between kernel, the different execs and rptrply. The use of buffers will be described in section 3.2.4. Communication between the other processes is done via RPC. Figure 1.5 gives an overview of the whole system.

hprr

hprr is the report and reply process linked to each session of bulles. It allows bulles to ask for reports on a specific DSC and will receive every update from rptrply before giving it to bulles. Eight different services are defined for this server (figure 1.1):

- seven for the communication between bulles and hprr (1 to 1)
- one for the communication between rptrply and hprr (n to 1).

The details of these services are given in section 3.1.
\textit{hprr services}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{hprr_services_diagram.png}
\caption{hprr services}
\end{figure}

\textbf{hprr services}

\textit{rptrply}

\textit{rptrply} is the report and reply process in the DSC. It is linked to the \textit{kernel}, to the \textit{buffer manager}, and to all the \textit{execs} in the same DSC. There is only one \textit{rptrply} per DSC. Eight different services are defined for this server (figure 1.2):

- three for the communication between \textit{kernel} and \textit{rptrply} (1 to 1)
- four for the communication between \textit{hprr} and \textit{rptrply} (n to 1)
- one for the communication between \textit{actif} and \textit{rptrply} (1 to 1).

The details of these services are given in section 3.2.

\textbf{actif}

There is only one process \textit{actif} for the whole RF system. It runs on the operational workstation \textit{rfsrv1}. It writes, in shared memory, the new setting sent by \textit{rptrply} before \textit{archive} reads it. Three different services are defined for this server (figure 1.3):
CHAPTER 1. SPS RF CONTROL SYSTEM

Figure 1.2: rptrply services

Figure 1.3: actif services
There is only one process *archive* for the whole system. It runs on the operational workstation rfsrv1. It reads, in shared memory, the current settings or the archived settings. This server allows *bulles* to save an archive, modify it and reload an entire system. Five different services are defined for this server (figure 1.4):

- five for the communication between *bulles* and *archive* (n to 1).

The details of these services are given in section 4.2.
Figure 1.5: Global System
Chapter 2

Organization of directories

Two different types of platforms are used for the application: HP workstation with the UNIX operating system, and DSCs with LynxOS.

2.1 On the workstations

/user/spsrfop/sps_rf_op is the directory for the four operational processes on the workstations (hprr, actif, archive and bulles).

```
sps_rf_op
  | COM        | IDF
  |       | NC_SRC
  |       | NC_INCL
  | UTIL        | INCLUDE
  |       | SH_SRC
  |       | SH_INCL
  | ACTIF
  | ARCHIVE
  | HPRR
  | DOC        | R_AND_R
  |       | bulles
  | bulles
```

Figure 2.1: Organization of directories on the workstations

Each process has its own sub directory, there is a sub directory for communication between pro-
cesses (/COM), and one sub directory shared by most processes (/UTIL). See figure 2.1.

2.1.1  COM directory

Directory /COM contains the files that define the servers. As most processes communicate with each other, definitions of all the services are gathered in /COM/IDF. You find there: communic.h, archive.hostname, actif.nc, archive.nc, hprr.nc, rptrply.nc, and rfs.nc (used only by bulles).

File communic.h contains the definitions of types used in the *.nc files. Those types can be used in usual C programs. For my applications, I have not used those types, but have redefined them in /UTIL/INCLUDE/types.h.

File archive.hostname defines the name of actif and archive hostname. It is included inglobals.h. (If that name is changed, you must recompile the whole application).

Files *.nc define the services: host name, port number, timeout . . . The network compiler creates files *.defs.h that are stored in /COM/NC_INCL, and files *.support.c, *.server.c and *.client.c that are stored in /COM/NC_SRC.

2.1.2  UTIL directory

Include files  Each source file includes shared include files that are gathered in directory /UTIL/INCLUDE.

cntl.h contains standard definitions, error definitions and string length constants, path definitions and read format.

constants.h contains constants used in the different sources.

globals.h contains global variables used in the different sources.

md5Globals.h contains definitions exclusively used by MD5 (it has not been written by us, but is included in the MD5 package).

types.h contains types definitions (the same as in communic.h plus other types used in the different sources).

Shared sources  Some procedures are used by different processes. So instead of copying each procedure in the corresponding directory, I gathered them in a shared directory. These sources are stored in /UTIL/SH_SRC, prototypes are stored in /UTIL/SH_INCL.

display.c: display types BCommand, KCommand, RrData, ExecTarget especially for debugging.

free.c: free memory allocated in store.c for the different types: BCommand (command defined for bulles with only target number), KCommand (command defined for kernel with full description of targets), RdData, ExecTarget. Those procedures are also called to free sequences of RPC call. (NC frees the main structure but not the sequences inside the structure). It is very important to check if the length of the sequence is greater than zero and that the pointer is not NULL before
freeing it. An initialization of the pointer to NULL and of the length to zero is also necessary for safety.

`readdatabase.c`: procedures declared here are used by both `actif` and `archive`. They are used to open files, read them and store data in shared structures such as System, Node, Target...

`search.c`: procedures declared here are used by both `actif` and `archive`. They check if the command is correct and calculate the target location to store data.

`store.c`: procedures declared here copy stuctures (BCommand, KCommand, RrData, Exec-Target) from local variables to global variables and from RPC variables to local variables. Since memory is allocated, it is necessary to free the structure after its use. (See `free.c`)

### 2.1.3 Process directory

The directory associated to each process is divided into 5 sub directories: `/SRC`, `/OBJ`, `/INCL`, `/BIN`, `/DATA`. Figure 2.2 shows the directory associated to `archive`.

```
ARCHIVE
<table>
<thead>
<tr>
<th>BIN</th>
<th>Makefile</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>archive</td>
</tr>
<tr>
<td>SRC</td>
<td></td>
</tr>
<tr>
<td>INCL</td>
<td></td>
</tr>
<tr>
<td>OBJ</td>
<td></td>
</tr>
</tbody>
</table>

DATA
| ARCH1
|   RF_Synchro
|       TWC_200_Low_level
|       ARCH40
```

Figure 2.2: Organization of the archive directory

**Source files** The source for each process is divided into 4 files whose names begin with the first 3 letters (`xyz`) of the process i.e. `act`, `arc`, `hpr`.

- `xyzInitialization.c` defines some procedures to intialize global variables and read flat tables.
- `xyzIdfProcedures.c` defines the services declared in the corresponding IDF file.
- `xyzProcedures.c` defines procedures used in the previous files and in the main.
- `<process>.c` contains the main, calls the initialization routine, exports and/or imports services before waiting for RPC calls.
**Include files**  The prototypes of procedures declared in `xyzInitialization.c`, `xyzIdfProcedures.c` and `xyzProcedures.c` have the same name with h extension and are stored in `/INCL`.

**Object and binary files**  Each process has its own makefile in `/BIN`. The gnu compiler creates object files associated with sources from `/SRC`, `/COM/NC_SRC` and `/UTIL/SH_SRC` and store them in `/OBJ`. Each process creates its own object files, even for shared sources, so that we can compile and link the processes on different machines. If we would have shared object files, `actif` could have been linked with object files created by `hprr`'s Make with a different compiler. The result of the link is in `/BIN` and must be started in this sub directory.

Each process needs data files coming from the data base.

**hprr data**  `hprr` needs only one file named `hprSystems.system`. This file is stored in directory `/user/spsrfop/HPRR/DATA`.

**hprSystems.system:**

<table>
<thead>
<tr>
<th>system name</th>
<th>date</th>
<th>pca name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurements_BA3</td>
<td>07−10−94</td>
<td>rfscf1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**actif data**  `actif` needs several flat tables to initialize its structures. The number of systems is defined in the file `actSystems.system`.

**actSystems.system:**

```
<table>
<thead>
<tr>
<th>system name</th>
<th>#node</th>
<th>#child</th>
<th>#target</th>
<th>#runtime</th>
<th>#setting</th>
<th>#step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurements_BA3</td>
<td>36</td>
<td>35</td>
<td>37</td>
<td>29</td>
<td>16</td>
<td>46</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

continued below (extra columns) . . .

<table>
<thead>
<tr>
<th>date</th>
<th>pca name</th>
</tr>
</thead>
<tbody>
<tr>
<td>07−10−94</td>
<td>rfscf1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

#node is the number of lines in the table `<systemName>.node`.

**<systemName>.node:**

```
<table>
<thead>
<tr>
<th>idx</th>
<th>node name</th>
<th>#target</th>
<th>target idx</th>
<th>#child</th>
<th>child idx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Measurements_BA3</td>
<td>1</td>
<td>16</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>200MHZ_COMPONENT</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>−1</td>
</tr>
<tr>
<td>2</td>
<td>BL_ACQUISITION</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>−1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
Target index is the index of the first target of the node in table `<systemName>.target`. And `#target` is the number of lines in this table corresponding to that node. Child index is the index of the first child of the node in table `<systemName>.child`. And `#child` is the number of corresponding lines in this table.

`<systemName>.child:`

<table>
<thead>
<tr>
<th>idx</th>
<th>parent node name</th>
<th>child node name</th>
<th>child node idx</th>
<th>path increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BUNCH_LENGTH</td>
<td>BL_BUNCH_NUMBER</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>BUNCH_LENGTH</td>
<td>BL_HARMONIC_NUMBER</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>BUNCH_LENGTH</td>
<td>BL_ACQUISITION</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Child node index is the index of the child node name in table `<systemName>.node`. Path increment is used to compute the unique value assigned to each path in the graph, and then to identify its runtime name.

`<systemName>.target:`

<table>
<thead>
<tr>
<th>idx</th>
<th>node name</th>
<th>target description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>200MHZ_COMPONENT</td>
<td>dummy</td>
</tr>
<tr>
<td>1</td>
<td>BL_ACQUISITION</td>
<td>Bunch$1</td>
</tr>
<tr>
<td>2</td>
<td>BL_ACQUISITION</td>
<td>Bunch$2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(Target description does not support blanks. They are be replaced by $).

`#runtime` is the number of lines in the table `<systemName>.rtsettings`.

`<systemName>.rtsettings:`

<table>
<thead>
<tr>
<th>path idx</th>
<th>runtime name</th>
<th>setting idx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>COMP_200</td>
<td>−1</td>
</tr>
<tr>
<td>1</td>
<td>BL_ACQ</td>
<td>−1</td>
</tr>
<tr>
<td>2</td>
<td>BL_BU</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Setting index is the index in table `<systemName>.setsize`. And `#setting` is the number of corresponding lines in the table `<systemName>.setsize`

`<systemName>.setsize:`

<table>
<thead>
<tr>
<th>idx</th>
<th>runtime name</th>
<th>leaf name</th>
<th>target comb.</th>
<th>Parameters (max nbr)</th>
<th>Data (max nbr)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>long float string</td>
<td>long float string</td>
</tr>
<tr>
<td>0</td>
<td>TEST</td>
<td>TEST_LEAF_NAME</td>
<td>4 1 3 0</td>
<td>1 2 1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BL_BU</td>
<td>BL_BUNCH_NUMBER</td>
<td>1 0 0 8</td>
<td>0 0 0</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

`1Files .setsize, .setting, and .Data show data for the test runTimeName TEST`
Target comb. is the product of the number of targets of each node involved in the command with that runtime name. And #step is the number of lines in the table `<systemName>`\textunderscore cmdstep.

**<systemName>\textunderscore cmdstep:**

<table>
<thead>
<tr>
<th>idx</th>
<th>runtime name</th>
<th>setting idx</th>
<th>node description</th>
<th>node idx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BL_BU</td>
<td>0</td>
<td>Measurements$\text{BA3}$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>BL_BU</td>
<td>0</td>
<td>Bunch$\text{length}$</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>BL_BU</td>
<td>0</td>
<td>Bunch$\text{number$$(B)}$</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Setting index is the index in table `<systemName>`\textunderscore setsize. Node index is the index in table `<systemName>`\textunderscore node. Node description does not support blanks (they must be replaced by $\)$. Nodes are ordered by runtime name and by depth for the same runtime name. (Depth zero at the root of the graph).

Once actif has read those files, it initializes data segments for long, float, string data and parameters, retrieves the archive data from the disk, and stores them into its memory, by reading, for each system, the file `<systemName>`\textunderscore settings in `/ACTIF/DATA/<systemName>`.

**<systemName>\textunderscore settings:**

<table>
<thead>
<tr>
<th>idx</th>
<th>runtime name</th>
<th>Data (actual nbr)</th>
<th>Parameters (actual nbr)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>long</td>
<td>float</td>
</tr>
<tr>
<td>0</td>
<td>TEST</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>TEST</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>TEST</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>TEST</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

For one runtime name, there is one line per target combination. You can note that target combination #3 has no float parameter and target combination #4 has 2, for an allowed maximum of 3 (see table .setsize).

Finally, actif reads the files `<runTimeName>`\textunderscore Data in the same directory, to store data for each target of each runtime name.

**TEST.Data:**

| 197144110 | 200090807 | 200123456 | 200000300 | 4 long data |
| 2.05      | 2.68      | 1.93      | 0         | 3.89 | 0 | 1.54 | 2.01 | 8 float data (2 per tg) |
| void      | hello$\text{boys}$ | this$\text{is$no$joke}$ | just$\text{an$example}$ | 4 string data |
| 4387      | 3568      | 8756      | 6878      | 4 long parameters |
| 0.2       | 0.4       | 0.4       | 0.5       | 0.5   | 0   | 0.6  | 0.5  | 0 | 12 float parameters |
| no line for string param. |
CHAPTER 2. ORGANIZATION OF DIRECTORIES

archive data  A specific sub directory is created for each archive (/ARCH1 to /ARCH40). Process archive needs exactly the same files as actif for each archive except for actSystems.

.arcSystems.details:

<table>
<thead>
<tr>
<th>system name</th>
<th>#node</th>
<th>#child</th>
<th>#target</th>
<th>#runtime</th>
<th>#setting</th>
<th>#step</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF_Synchro</td>
<td>10</td>
<td>9</td>
<td>30</td>
<td>8</td>
<td>7</td>
<td>15</td>
<td>18 - 08 - 94</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

continued below (extra columns) . . .

<table>
<thead>
<tr>
<th>pca name</th>
<th>key</th>
<th>archive description</th>
<th>owner</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfscf1</td>
<td>1036</td>
<td>RF$Synchro$p,+,$e,$op.$Sept94</td>
<td>baudre</td>
<td><a href="mailto:baudre@hpslz7.cern.ch">baudre@hpslz7.cern.ch</a></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

continued below (extra columns) . . .

<table>
<thead>
<tr>
<th>date of archive</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 - 09 - 94</td>
</tr>
</tbody>
</table>

key.last contains the key of the next archive to be saved. This key is a code of the archive index, so you can not get or save data if you do not know the key of the concerned archive.

2.2 On the DSCs

Directory /user/spsrfop/RF_PCA/SOURCE/R_AND_R is the directory for the operational process on LynxOS (rptrply). The operational sources are in the sub directory OPERA while the development sources are in the sub directory DEVELOP. (This branching is not shown on figure 2.3). The binary file is moved to the directory /user/spsrfop/RF_PCA/SYSTEMS/<HOSTNAME>/EXEC where HOSTNAME equals RFSCF1, RFSCF2 or RFSBA2. This is the directory from where the binary is started. See figure 2.3.

Process rptrply has its own sub directory. There is a sub directory for communication between processes (/COM), and one sub directory shared by most processes (/UTIL).

2.2.1 COM directory

Directory /COM contains the files that define the servers. As most processes communicate with each other, definitions of all the services are gathered in /COM/IDF. You find there: communic.h, archive.hostname, actif.nc, hprr.nc, and rptrply.nc.
File communic.h contains the definitions of types used in the *.nc files. Those types can be used in usual C programs. For my applications, I have not used those types, but have redefined them in /UTIL/INCLUDE/Mytypes.h.

File archive.hostname defines the name of actif and archive hostname. It is included in globals.h. (If that name is changed, you must recompile the whole application)

Files *.nc define the services: host name, port number, timeout ... The network compiler creates files *_defs.h that are stored in /COM/NC_INCL, and files *_support.c, *_server.c and *_client.c that are stored in /COM/NC_SRC.

2.2.2 UTIL directory

Include files  Each source file includes shared include files that are gathered in directory /UTIL/INCLUDE.

cntl.h contains standard definitions, error definitions and string length constants, path definitions and read format.

constants.h contains constants used in the different sources.

globals.h contains global variables used in the different sources.

types.h or Mytypes.h on Lynx contains types definitions (the same as in communic.h plus other types used in the different sources.)
Shared sources  store.c, display.c and free.c are exactly the same as the ones on the workstation.

### 2.2.3 Process directory

The directory associated to each process is divided into five sub directories: /SRC, /OBJ, /INCL, /BIN, /DATA.

**Source files** The source for the process is divided into 4 files whose names begin with the first 3 letters of the process i.e. rpt.

- **rptInitialization.c** defines some procedures to initialize global variables and read flat tables.
- **rptIdfProcedures.c** defines the services declared in the corresponding IDF file.
- **rptProcedures.c** defines procedures used in the previous files and in the main.
- **rptrply.c** contains the main, calls the initialization routine, exports and/or imports services before waiting for RPC calls.

**Include files** The prototypes of procedures declared in rptInitialization.c, rptIdfProcedures.c and rptProcedures.c have the same names with h extension and are stored in /INCL.

**Object and binary files** rptrply has its own makefile in /BIN. The gnu compiler creates object files associated with sources from /SRC, /COM/NC_SRC and /UTIL/SH_SRC and store them in /OBJ. The process creates its own object files, even for shared sources. The result of the link is in /BIN. It is copied to ~spsrfop/RF_PCA/SYSTEMS/<HOSTNAME>/EXEC (where HOSTNAME = RFSCF1, RFSCF2, RFSBA2 ...) and must be started in this sub directory.

**Data files**

rptrply  Process rptrply needs only one file named rpt<hostname>.system. This file is stored in directory ../GRAPH_DATA relatively to /user/spsrfop/RF_PCA/SYSTEMS/<HOSTNAME>/EXEC where binary files are started.

**rpt<hostname>.system:**

<table>
<thead>
<tr>
<th>system name</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurements_BA3</td>
<td>21 - 10 - 94</td>
</tr>
<tr>
<td>RF_Synchro</td>
<td>05 - 10 - 94</td>
</tr>
<tr>
<td>TWC_200_Low_level</td>
<td>06 - 10 - 94</td>
</tr>
<tr>
<td>TWC_200_Power</td>
<td>10 - 10 - 94</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Chapter 3

Report & Reply processes

3.1 hprr

3.1.1 Introduction

bulles forks hprr with the three arguments:

1. ghRRBPort: this port number is chosen at random by bulles. If this number already exists for another session of hprr, the RPC call mmiFirst will warn bulles which will try to fork hprr with a new port number. After eight unsuccessful attempts process bulles stops.

2. HPRServerSite: host name where server hprr is started (not very useful actually).

3. ghFatherPid: hprr needs the process identifier of bulles to send signals to it (SIGUSR1 and SIGUSR2).

After the initialization phase, hprr exports its service (with port number ghRRBPort) and waits in the NC loop. It exits from this loop every 30 seconds (FATHER_ALIVE_TIMEOUT defined in constant.h) to check if bulles is still alive. If bulles is dead, hprr stops reports and replies on every DSC and kills itself, otherwise it comes back to the loop and waits for RPC.

3.1.2 Initialization, globals and types

In this section, we define some important structures for the process and the use of those structures as global or local variables.

Initialization   Initialization is made with the procedure initialization() which calls other procedures gathered in hprInitialization.c.

ghPCA_List is defined as an array of PCA_LIST:
CHAPTER 3. REPORT & REPLY PROCESSES

typedef struct _PCA_LIST {
    int rptSid;
    String pcaName;
    ushort report;
    ushort reply;
} PCA_LIST;

This global array contains informations concerning the DSC(s). rptSid is the service entry identifier returned by the import of each rptrply service. report is a number indicating on how many systems, on this DSC, the user has asked reports (report varies between 0 and ghSystemNbr). reply is a flag indicating whether the user has asked reply on this DSC.

ghSYSTEM_List is defined as an array of SYSTEM_LIST:

typedef struct _SYSTEM_LIST {
    String systName;
    ushort report;
    ushort pcaIndex;
} SYSTEM_LIST;

For each system, we store in this global array whether there is a report or not and the index in ghPCA_List of the DSC controlling this system.

Those two arrays are initialized in the procedure fillPcaSystemList by reading the file hprSystems.system. This procedure also initializes ghSystemNbr (number of systems) and ghPcaNbr (number of DSCs).

We use two global pointers that contain all the information needed for mmiUpdate.

The two pointers ghLastUpDate and ghmmiUpDateList have the following type:

typedef struct _BCommand_List {
    BCommand *cmd;
    BNodeSeq *execTarget;
    RrData *rrData;
    struct _BCommand_List *next;
} BCommand_List;

ghLastUpDate only contains the message code and family (in field Message of the RrData structure) that warn bulles that it is the last update. ghmmiUpDateList is the linked list containing all the updates received from rptrply. Both pointers are initialized in procedure initialization.

Other types and globals  ghUSERInfo is defined as a ClientId:

typedef struct _clientid {
    ushort uid;
    ushort gid;
    ulong host;
    ushort port;
    String hostName;
} ClientId;
This global is used to identify the user of bulles. With the NC, one can easily derive this info as soon as hprr receives a RPC call from bulles. This information will be sent to rptrply to identify every user. uid, gid, host and hostName can be the same if one user starts the application several times on the same station. The difference is the port of bulles. In fact (port, host) is unique and the rest is superfluous.

ghUserIdentified is a flag indicating whether the user of bulles has been identified or not.

### 3.1.3 RPC services

**Communication between bulles and hprr**

Eight different services are defined between bulles and hprr. We can divide those services into four subsets.

- **bulles forks hprr**

  Process bulles forks hprr and gives it a port number as a server. The service mmiFirst tests whether any other hprr has got the same port number on the same machine. If this port number already exists for hprr, the new hprr will abort because export will not succeed and bulles will send mmiFirst to the previous hprr. This hprr will respond that it is not the first time it responds to this service (non zero value in the errorCode field of the Error returned by mmiFirst()). bulles then kills the hprr just forked, and tries to fork a new hprr with a different port number (maximum of eight trials).

- **bulles asks report and reply via hprr to rptrply**

  Four services allow bulles to ask reports or replies and to stop them on every system.

  1. **mmiStartReply**: when a user selects a system with the MMI, bulles automatically sends a mmiStartReply to hprr. This service sets the reply flag to one for the DSC containing this system and sends the command startReply to the right rptrply.

  2. **mmiStopReply**: if the user selects another system which is on another DSC, bulles automatically sends a mmiStopReply for the previous DSC. hprr receives this call, unsets the reply flag for this DSC and sends a stopReply to the right rptrply.

  3. **mmiStartReport**: the user can ask reports on a system with the MMI. bulles sends a mmiStartReport to hprr. This service increments the report number by one for the DSC containing this system and sets the report flag to one for the system. It sends the command startReport to the right rptrply.

  4. **mmiStopReport**: when the user stops reports on a system with the MMI, bulles sends a mmiStopReport to hprr. This service decrements the report number by one for the DSC containing this system and sets the report flag to zero for the system. It sends the command stopReport to the right rptrply.

- **bulles reads the updates from hprr**

  When hprr receives a rrUpdate from rptrply it stores it in the linked list ghmmiUpDateList and sends signal SIGUSR1 to bulles. As soon as possible, bulles sends the call mmiUpdate to read the entire linked list containing results of commands. hprr just sends back the
command, the rrData and the execTargets and frees this set corresponding to the previous
call from its list. (See section 3.1.4 for more details about this procedure).

- **bulles asks hprr to re-establish the communication with rptrply**

  If *kernel* disappears, and is later restarted (reboot of a PCA) while *bulles* and *hprr* remained
  operational, the communication with *rptrply* is lost because the new *rptrply* does not know
  the (random) port number of *hprr*. Upon its next attempt to send a command to the *kernel*,
  *bulles* realizes the situation, performs a call *loginPca* to the *kernel*, and sends a call
  **mmiReConnect** to *hprr*. Upon reception, *hprr* establishes a link with *rptrply*, sends a
  **startReply** and one (or several) **startReport** to it.

**Communication between rptrply and hprr**  Process *rptrply* treats the buffers coming from the
execs and sends, to *hprr*, the set: command, rrData and execTarget for a reply or a report. Process
*hprr* copies this set in its linked list (**addMmiUpDateList**) and sends a SIGUSR1 to *bulles* to warn
it that an update is available. Alarms are also sent via this channel.

### 3.1.4 Other procedures

1. **addMmiUpDateList**: this function is called in **rrUpdate**. It allocates memory in the linked
   list **ghmmiUpDateList** for the set: command, rrData and execTarget. **ghmmiUpDateList**
   points to the first element of the linked list. When a set is added, it is added at the end of
   the list. The last element is always NULL. So when a new set is added, we jump to the
   element with NULL, we replace NULL by the new set which then receives NULL as its
   next element. The structures received via RPC (command, rrData and execTarget) must be
   freed at the end of the call because NC does not free the sequences inside those structures.

   We explain here in more details the procedure **mmiUpdate**: When *hprr* receives a **rrUpdate** it stores the data set at the end of the list and sends a signal
to *bulles*. When *bulles* sends an update request, we send back to him the first element of the
linked list, and **ghmmiUpDateList** points to the next element of the list. The first element
(that has been sent) must be freed, but we must wait until this element has been sent back
to *bulles* before freeing it. So we have decided to free it during the next call to **mmiUpdate**.
At the begining of the procedure we free the element pointed to by **toFreeList**, and then
**toFreeList** points to the next element (same as **ghmmiUpDateList**). The element pointed
to by **ghmmiUpDateList** is stored in structures to be sent to *bulles* and **ghmmiUpDateList**
points to the next element. If **ghmmiUpDateList** points to NULL, we send **ghLastUpDate**
to *bulles*. See figure 3.1.

2. **isMyFatherAlive**: every 30 seconds *hprr* exits from the NC loop and tests if *bulles* is still
   alive. If not, *hprr* stops reports and replies on the concerned DSC(s) and kills itself.
3.2 \texttt{rptrply}

3.2.1 Introduction

Process \textit{kernel} forks \texttt{rptrply} with no argument. After the initialization phase, \texttt{rptrply} exports its services, imports the server \textit{actif}, and waits in the NC loop. It exits from this loop every second (BUFFER\_TREATMENT\_TIMEOUT defined in constant.h) to treat a maximum of 80 (MAX\_BUFFER\_TO\_TREAT) buffers coming from execs and waiting in a queue. After the treatment of those buffers, it comes back to the loop and waits for RPC.

3.2.2 Initialization, globals and types

Initialization Initialization is made with the procedure initialization() which calls other procedures gathered in rptInitialization.c.

Procedure fillSystemList reads the file rpt<hostname>.system, fills the grSYSTEM\_List array with the name of systems and sets the number of systems (grSystemNbr). In procedure initialization, we open the buffer manager client, initialize buffers for functions RcvSBUFFER and BufType, and initialize null globals (gNoBCommand, gNoKCommand, gAlarmRrData, gNoRrData, gNoExecTarget) that will be sent by \texttt{rrUpdate} in case of alarm.

Other types and globals grUAptr is a linked list of type User\_Array:

```c
typedef struct _User_Array {
    long clientId;  /* Client ID */
    long userKey;   /* User key */
    int hprrSid;    /* Sid for rptrply->hprr connexion */
    int report;     /* Report number, from 0 (no report) */
} grUAptr;
```
Field `userKey` is filled during `login` received from `kernel`. Field `userInfo` is filled during `login` or `startReport` (if the user is not logged on this DSC). The first time `rptrply` receives `startReply` or `startReport`, it imports service `hprr` of this user and stores the SID in `hprrSid`. `report` gives the number of reports being requested on this DSC and `reply` is a flag indicating whether a reply has been asked or not. (If the user is logged on this DSC, reply is set to one).

`grSys_Rpt_Arr[MAX_SYSTEM_NBR]` is a linked list of type `Report_List`:

```c
typedef struct _Report_List {  
    long userKey;                /* user ident. */  
    ClientId userInfo;          /* user info. */  
    long rrAttribute;           /* report attribute. */  
    int hprrSid;                /* report SID. */  
    struct _Report_List *next;  /* next of list. */  
} Report_List;  
```

This linked list contains, for each system, informations about users who have asked reports. `userKey` is not really necessary since we only use `userInfo` to identify the user (and remove it from the list after `stopReport`).

`grRCAptr` is a linked list of type `Remote_Command_Array`:

```c
typedef struct _Remote_Command_Array {  
    KCommand *cmd;                 /* command. */  
    long userKey;                  /* user ident. */  
    ClientId userInfo;            /* user info. */  
    int klc;                       /* kernel link counter. */  
    short macro;                   /* macro flag (always 0 till now). */  
    time_t time_started;          /* time of command reception. */  
    time_t time_completed;        /* time of last exec EOMSequence. */  
    short rca_p;                  /* index of command in RCA. */  
    struct _Remote_Command_Array *next;  /* next of list. */  
} Remote_Command_Array;  
```

Remote_Command_Array contains the command received from `kernel`, the identity of the sender, the number of execs involved in the command (klc), a macro flag, the time when it has started and completed, the rca pointer which allows the identification of the command when `rptrply` treats a buffer.

`grSCAptr` is a linked list of type `Setting_Command_Array`:

```c
typedef struct _Setting_Command_Array {  
    KCommand *cmd;                 /* command. */  
    String execName;              /* command exec name. */  
    struct _Setting_Command_Array *next;  /* next of list. */  
} Setting_Command_Array;  
```
Setting_Command_Array contains all the commands, terminated with a write in a leaf, stored during the time when the connection between rptrply and actif was broken (or when actif was dead).

3.2.3 RPC services

Communication between kernel and rptrply

1. pcaLogin: when the user selects a system, it will automatically send a login to the kernel in the DSC containing this system. Process kernel sends then a pcaLogin to rptrply. This happens only for the first system selected. If the user decides to work on another DSC, he will still be logged on the former DSC. (But he will have stopped reply on it). In this procedure, grUAptr is filled with the userKey and userInfo.

2. pcaLogout: removes user information from grUAptr. This procedure is called only when a user quits bulles properly. Otherwise the user remains in this array forever.

3. pcaCommand: process kernel receives a command from bulles, checks if it is correct and sends it to rptrply with this procedure. Process rptrply stores it in grRCAptr, computes a new rca pointer and sends it back to kernel.

Communication between hprr and rptrply

1. startReply: during the call mmiStartReply, hprr sends a startReply to the right DSC. Process rptrply imports server hprr and stores the corresponding hprrsid in grUAptr. (If this import has already been done by startReport it just skips). It also sets the reply flag to one.

2. stopReply: during the call mmiStopReply, hprr sends a stopReply to the right DSC. This procedure sets the reply flag to -1 (meaning that the user has already started a reply once), but does not free the connection with hprr even if there are no report. If the field rrbPort is equal to the special key gLOGOUT, it will remove user from grUAptr (used when bulles is dead and hprr still alive).

3. startReport: during the call mmiStartReport, hprr sends a startReport to the right DSC. Process rptrply imports server hprr and stores the corresponding hprrsid in grUAptr. (If this import has already been done by startReply it just skips). It also increments the report number by one in grUAptr and stores user information in grSys_Rpt_Arr for the concerned system.

4. stopReport: during the call mmiStopReport, hprr sends a stopReport to the right DSC. This procedure decrements the report number by one in grUAptr and removes user information in grSys_Rpt_Arr for the concerned system.
CHAPTER 3. REPORT & REPLY PROCESSES

Communication between *actif* and *rptrply*  If the communication between *rptrply* and *actif* is interrupted, or if *actif* is dead, putData will not succeed and an alarm warns all the users that there is a problem with *actif*. This produces a timeout and to avoid timeout on subsequent calls, we set the global grActifLives to zero so that every write is stored in grSCAptr instead of being sent to *actif*. When the communication is restored, *rptrply* must empty grSCAptr and sets grActifLives to one. *actifRefreshes* is sent by *actif* when it starts and every five minutes (SAVE_DATA_TIMEOUT) thereafter. It just sets grActifLives to one and warns the users that the communication is restored (via an alarm).

3.2.4 Buffer treatment

Each time *rptrply* exits from the NC loop, it looks if there are buffers, coming from the execs, to treat. The first procedure called is BufType that gives the type of the different access descriptors.

We treat only M_MESS, M_FBERR, M_AERR and LF_DATA. To store the index of access descriptors to treat, we use the procedure fillAcsDscIndexArray. This procedure fills the array grAcsDscIndexArray (its size is MAX_ACS_DSC, i.e. 10) with the index, in the received buffer, of access descriptors to treat. It also initializes the global grAcsDscToTreat. We can thus do a for loop on access descriptor to treat.

Procedure findLeafDataNbr returns the number of LF_DATA access descriptors for the buffer and initializes grLeafDataNbr. This allows memory allocation for the sequence grRrData to be sent to *hprr*. If grLeafDataNbr is equal to two, that means that there are two different target sequences with two different sets of data.

The array tagIdArray (size MAX_NODE_IN_CMD) contains the tagId not null contained in the node sequence of the command. The procedure fillTagIdArray fills this array and initializes the global grTagIdNbr. This allows quick comparison between the tagId coming from the execs (stored in acs_tag->tag_ptr) and the ones of the command.

Before treating the buffers, we check if somebody is interested in what is in it. The procedures setY (for reply) and setR (for report) compare the cla (command level attribute) coming from exec to the rrA (report reply attribute) chosen by the user. If nobody is interested (information from exec is too detailed), we do not treat the access descriptors.

We now do a for loop on access descriptors to treat.

1. MESS: Each buffer contains one and only one message. We just copy the string from str_ptr to grRrData.message.msgText. A special code is given if it is the last message of the command. If the code is END_OF_MASTER_SEQUENCE, we set the grActiFile flag to one (the command will be sent to *actif* if the command is a write). Since we have only one field msgCode, we do not overwrite it with END_OF_MASTER_SEQUENCE when a special code is already used (this is the case when something wrong happens with LF_DATA).

2. AERR and FBERR: If an error happens, we can send it to *hprr* via the message. Since we only have one message field per buffer, we must send first the error message and then the message of the buffer (generally sequence aborted). The procedure sendErrorToIssuer
sends an update to the issuer of the command with the message text and the message code. The other access descriptors are then treated the same way as usual.

3. **LF_DATA**: a buffer can contain from zero to several LF_DATA corresponding to different target combinations. Special cases are explained here depending on the structure of the tag_ptr.

   **General case**: The usual tag_ptr contains four tagId with one number for each tagId. The structure of the tag_ptr is:
   
   No IDENT : 1; TX : 2; No IDENT : 1; No IDENT : 1;
   
   No IDENT means there is no tagId.
   
   Procedure getTagIdVal returns the tagId and the TagVal of the tag_ptr. The first call to this procedure returns NO_IDENT and 1. We do not treat NO_IDENT. The second call returns TX and 2. We check if TX is a tagId of the command (we just have to look in tagIdArray). If we find this tagId, the procedure findTagNo finds the tagNo corresponding to the tagVal 2. We copy this value in grOutknodeseq.

   **Example**: Let's assume we have sent the command:
   
   — **NodeName**: TWC_200_Power
   — **Targets**: 0 0
   — **NodeName**: TX
   — **Targets**: 0 LN_3 TX 1 1 LN_3 TX 2 2 LN_4 TX 1
   — **NodeName**: TX_AGST
   — **Targets**: 0 0
   
   We receive the following tag_ptrs for exec LN_3:
   
   No IDENT : 1; TX : 1; No IDENT : 1; No IDENT : 1;
   
   No IDENT : 1; TX : 2; No IDENT : 1; No IDENT : 1;
   
   When we treat the first tag_ptr, grOutknodeseq will be
   
   — **NodeName**: TWC_200_Power
   — **Targets**: 0 0
   — **NodeName**: TX
   — **Targets**: 0 LN_3 TX 1
   — **NodeName**: TX_AGST
   — **Targets**: 0 0
   
   We copy the first set of data (concerning tagNo 0) in grRrData.data.data[0].
   
   When we treat the second tag_ptr, grOutknodeseq will be
   
   — **NodeName**: TWC_200_Power
   — **Targets**: 0 0
   — **NodeName**: TX
   — **Targets**: 0 LN_3 TX 1 1 LN_3 TX 2
   — **NodeName**: TX_AGST
   — **Targets**: 0 0
   
   Since we have only one node sequence per buffer, we copy 1 LN_3 TX 2 after 0 LN_3 TX 1 for the same node.
   
   We copy the second set of data (concerning tagNo1) in grRrData.data.data[1].
   
   When all the tag_ptrs have been treated, we copy grOutknodeseq in grExecTarget that will
be sent to hprr.
When all the tag_ptrs have been treated, we send the set grRrData grExecTarget and grB-
command to the concerned hprr(s) and free those globals. If a write ends successfully, we
send the grKcommand to actif so that it stores the new settings in its shared memory. Then,
we release the buffer.

Case where LF_TFR is used:
The tagId LF_TFR in tag_ptr is not treated. We assume that it is the same as NO_IDENT.

Case where user option is used:
Instead of having only four tagId and tagVal, it is possible to have five tagId and multiple
tagVal for the fifth tagId (corresponding to user option). In this special case, we can not
treat tagVal as usual to find tagNo. We just copy every target corresponding to the exec
from the input command to grExecTarget. We copy the special code TgMappedtoUsopt +
#node in msgCode (#node is the number of the node tagged to user option, the fifth tagId
gives this node number easily) while msgFamily is RPTRPLY_FAMILY. The msgCode and
msgFamily must not be overwritten (because they are interpreted by bulles).

Case where the procedure findTagNo does not find the right tagNo according to tagId and
tagVal:
In this case, we set the grUnexpectedTagId flag to one and copy the special code UNEX-
PECTED_TAGID in msgCode while msgFamily receives RPTRPLY_FAMILY. The msg-
Code and msgFamily must not be overwritten (because they are interpreted by bulles). We
copy a default target sequence in grExecTarget.
Chapter 4

Actif & Archives.

4.1 actif

4.1.1 Introduction

After the initialization phase, actif exports its service and waits in the NC loop. It exits from this loop every 5 minutes (RPTRPLY_REFRESH_TIMEOUT defined in constant.h) to send, to every DSC, a RPC to warn them that actif is alive and that they can send putData to them. It comes back to the loop and waits for RPC.

4.1.2 Types

During the initialization phase, we fill the different structures defined for each system. This section fully describes those structures and the way data are stored in shared memory segments. Refer to figure 4.1.

gSystTable is defined as a SystList:

```c
typedef struct _SystList {
    ushort num; /* number of systems */
    System *list; /* pointer to the system table */
} SystList;
```

with System:

```c
typedef struct _system {
    String name; /* system name */
    ushort nodeNum; /* number of nodes */
    Node *nodeTable; /* pointer to the node table */
    ushort childNum; /* number of children */
    Child *childTable; /* pointer to the child table */
    ushort targetNum; /* number of targets */
    Target *targetTable; /* pointer to the target table */
} System;
```
```c
ushort rTimeNum; /* number of run time */
RunTime *rTimeTable; /* pointer to the run time table */
ushort setNum; /* number of settings */
Setting *settingTable; /* pointer to the setting table */
long *longDataSegment; /* pointer to the first long in */
    /* the long Data Segment */
double *floatDataSegment; /* pointer to the first float in */
    /* the float Data Segment */
LongString *stringDataSegment; /* pointer to the first string */
    /* in the string Data Segment */
long *longParamSegment; /* pointer to the first long in */
    /* the long Param Segment */
double *floatParamSegment; /* pointer to the first float in */
    /* the float Param Segment */
String *stringParamSegment; /* pointer to the first string */
    /* in the string Param Segment */
int *seqLength; /* pointer to the first length */
    /* in the seqLength segment */
ushort stepNum; /* number of step */
Steps *stepTable; /* pointer to the step table */
LineDesc date; /* creation date */
String pcaServer; /* name of the pca */
}
```

Figure 4.1: System Structure
This structure is filled by the procedure fillActifgSystem by reading file actSystems.system. When we read this file we do not know the number of systems. So, we allocate memory dynamically for each new system. Once this file is read, we know the number of lines for each of the following files: <system>.node, <system>.child, <system>.target, <system>.cmdstep, <system>.rtsettings, and <system>.setsize. Memory is then allocated for each of the following structures: Node, Child, Target, RunTime, Setting and Steps.

```c
typedef struct _node {
    char name[NAME_LEN]; /* node identifier */
    short targetNum; /* number of targets */
    short targetIndex; /* index in the target table */
    short childNum; /* number of children nodes */
    short childIndex; /* index in the children table */
} Node;
```

This structure is filled by fillNodes.

```c
typedef struct _child {
    char name[NAME_LEN]; /* child node identifier */
    short nodeIndex; /* index in the node table */
    short pathInc; /* path increment used to */
                    /* generate unique path ID */
} Child;
```

This structure is filled by fillChildren. With Node and Child, we can compute the path index that identify the runtime names.

```c
typedef struct _target {
    char nodeName[NAME_LEN]; /* node identifier */
    char targetDesc[NAME_LEN]; /* target description */
} Target;
```

This structure is filled by fillTargets. This structure is used when we want to transform raw Data file into readable files (.readable).

```c
typedef struct _runtime {
    char rtName[NAME_LEN]; /* runtime identifier */
    short setIdx; /* index in setting table */
} RunTime;
```

This structure is filled by fillRTNames.

```c
typedef struct _setting {
    short setIdx;
    char rtName[NAME_LEN]; /* runtime identifier */
    short tagNbr; /* target combinaison number */
    short lDMaxLength;
    int lDLengthStart; /* sequence of length for each */
```

CHAPTER 4. ACTIF & ARCHIVES.

```c
/* target. lDLength.length */
/* = tagNbr */
int lDStartAddress;
short fDMaxLength;
int fDLengthStart;
int fDStartAddress;
short sDMaxLength;
int sDLengthStart;
int sDStartAddress;
short lPMaxLength;
int lPLengthStart;
int lPStartAddress;
short fPMaxLength;
int fPLengthStart;
int fPStartAddress;
short sPMaxLength;
int sPLengthStart;
int sPStartAddress;
}
```

This structure is filled by fillSetSize. This is the most important structure. It allows one to find
data location in shared memory segments. There is one such structure per runtime name.

```c
type struct steps {

    char rtName[NAME_LEN]; /* runtime identifier */
    short rtNameIdx; /* index in runtime table */
    char nodeDesc[NAME_LEN]; /* node description */
    short nodeIdx; /* index in node table */
}
```

This structure is filled by fillCmdStep.

4.1.3 Organization of data in the shared memory segments

For each system, we have created one shared memory segment per data type (long, float, string
data and long, float, string params). The size of those segments is fixed and defined by constants.
Those constants, defined in `cntl.h`, are 1000 for longs, 500 for floats and 100 for strings. As
the system evolved these constants was adjusted and redefined. We now have 20,000 long data,
10,000 float data, and 100 data strings. For parameters we have: 500 longs, 100 floats, and 100
strings.

At present (August 2002), the listed system stores the following data: Measurements.BA3 –
125 longs, RF_Synchro – 54 longs, Synchro_Diagnostic – 19 longs and two strings, TRVS_Dam-
per_settings – 384 longs and 216 floats, and TWC_200_Low_level – 28 longs and eight floats.

Data are stored in memory segments in alphabetical order of their runtime name. (In the order of
runtime names as they appear in `<system>.setsize`). By reading `<system>.setsize` (see page 11),
we know, for each runtime name, the number of target combinations and the maximum number
of data of each type. With those informations we can compute the starting address of data concerning each runtime name. The maximum number of data is not necessarily used for each target combination (for instance extraction pulses for $e^-$ may need four delays, and for $e^+$ it may need six delays for a maximum of eight). So we have to know, for each target combination of each runtime name, the effective number of data. Those numbers are stored in the shared memory segment seqLength of the structure System. We also have to compute a starting address in this segment. All the addresses are stored in Setting and are computed by the procedure fillSetSize.

4.1.4 Example

Figure 4.2 shows the organization of data for the system RF_Synchro.

The first two runtime names with settings are B_SEL_WR, with one target combination, and EXTR_PUL with the three target combinations ($e^+$, $e^-$ and MD). B_SEL_WR may store up to 15 longs, but you can see in seqLength that there are only four significant longs for this setting. When archive gets data from shared memory, it will only take the four longs (4360 2050 895 3205). The 11 other longs may be not null because they are the result of a previous putData with more than four longs. EXTR_PUL may store up to eight longs for each target combination. lDLengthStart and lDStartAddress are the address for the first target combination. We easily find the starting address for each target combination:

address of MD in seqLength
\[ = \text{lDLengthStart} + \text{#targetcombinations} - 1 \]
\[ = 1 + 3 - 1 \]
\[ = 3 \]

address of MD in longDataSegment
\[ = \text{lDStartAddress} + ((\text{#targetcombinations} - 1) \times \text{lDMaxLength}) \]
\[ = 15 + ((3 - 1) \times 8) \]
\[ = 31 \]

4.1.5 RPC services

Communication between archive and actif

1. actifDisable: when process archive is reading data in shared memory or is saving this shared memory as an archive, it empeaches actif from writing in the shared memory. actif just sets a gaWriteWait flag to 1. If actif receives a setting when it is disabled, it stores the command in the linked list gaCTTptr.

2. actifEnable: once archive has finished with the reading or saving, it enables actif to let it write in shared memory again. actif unsets the gaWriteWait flag and treats all the commands stored in gaCTTptr.
Figure 4.2: Organization of long Data for RF_Synchro
Communication between rptrply and actif  putData: when rptrply has treated a write command successfully, it sends this KCommand (command with full description of targets), with the execName concerned, to actif via the procedure putData. In this procedure, we copy the KCommand to a BCommand (command with only the target numbers). We only copy the targets linked to the exec named. This command is then treated by procedure treatPutData explained further. If the gaWriteWait flag is equal to 1, actif stores the command in gaCTTptr instead of treating it.

4.1.6 Other procedures

The treatment of a command to put data in shared memory (putData) or to get data from shared memory or from archives (getData and getReloadData) is quite similar. So we fully describe here the procedure treatPutData used by putData. The procedure treatGetData used in archive by getData will be briefly described further.

TreatPutData  We first test if the first node of the command is a root of a system. The rootname gives us the index (gASysIndex) of system in gSystTable[0].list. We check if the command is correct according to the tables node and child. We also compute the path index (gASetIndex) that identifies the index of the setting in gSystTable[0].list[gASysIndex].settingTable.

We have a sequence of targets for each node. Each combination of targets has a special number stored in a local array called tag, tgNbr. The following example explains how we compute this special number.

Suppose that we have received the following node sequence and target list:

- NODE1 (top node = root) with target list: 0 1 3 (and let the maximum number of targets for this node be maxNod1 = 5)
- NODE2 with target list: 1 3 (and let the maximum number of targets for this node be maxNod2 = 4)
- NODE3 (bottom node) with target list: 0 1 2 (and let the maximum number of targets for this node be maxNod3 = 3).

The number of target combinations for this runtime name is:

$$\text{maxNod}_1 \cdot \text{maxNod}_2 \cdot \text{maxNod}_3 = 5 \cdot 4 \cdot 3 = 60.$$

The tag number, tgNbr, of a target combination \((tg_1, tg_2, \ldots, tg_N)\) is given by (index 1 refers to the node at the top of the path):

$$\text{tgNbr of } (tg_1, tg_2, \ldots, tg_N) = \sum_{i=1}^{N} (tg_i \cdot \prod_{j=i}^{N} \frac{\text{maxNod}_j}{\text{maxNod}(i)})$$

(4.1)

Applying equation 4.1, we get:
For target combination (0, 1, 0) \[tgNbr = 0 \cdot \frac{5 \cdot 4 \cdot 3}{3} + 1 \cdot \frac{4 \cdot 3}{3} + 0 \cdot \frac{3}{3} = 3\]

(0, 1, 1) \[tgNbr = 0 \cdot \frac{5 \cdot 4 \cdot 3}{3} + 1 \cdot \frac{4 \cdot 3}{3} + 1 \cdot \frac{3}{3} = 4\]

(0, 1, 2) \[tgNbr = 0 \cdot \frac{5 \cdot 4 \cdot 3}{3} + 1 \cdot \frac{4 \cdot 3}{3} + 2 \cdot \frac{3}{3} = 5\]

(0, 3, 0) \[tgNbr = 0 \cdot \frac{5 \cdot 4 \cdot 3}{3} + 3 \cdot \frac{4 \cdot 3}{3} + 0 \cdot \frac{3}{3} = 9\]

(0, 3, 1) \[tgNbr = 0 \cdot \frac{5 \cdot 4 \cdot 3}{3} + 3 \cdot \frac{4 \cdot 3}{3} + 1 \cdot \frac{3}{3} = 10\]

(0, 3, 2) \[tgNbr = 0 \cdot \frac{5 \cdot 4 \cdot 3}{3} + 3 \cdot \frac{4 \cdot 3}{3} + 2 \cdot \frac{3}{3} = 11\]

(3, 1, 0) \[tgNbr = 3 \cdot \frac{5 \cdot 4 \cdot 3}{3} + 1 \cdot \frac{4 \cdot 3}{3} + 0 \cdot \frac{3}{3} = 39\]

(3, 1, 1) \[tgNbr = 3 \cdot \frac{5 \cdot 4 \cdot 3}{3} + 1 \cdot \frac{4 \cdot 3}{3} + 1 \cdot \frac{3}{3} = 40\]

(3, 1, 2) \[tgNbr = 3 \cdot \frac{5 \cdot 4 \cdot 3}{3} + 1 \cdot \frac{4 \cdot 3}{3} + 2 \cdot \frac{3}{3} = 41\]

(3, 3, 0) \[tgNbr = 3 \cdot \frac{5 \cdot 4 \cdot 3}{3} + 3 \cdot \frac{4 \cdot 3}{3} + 0 \cdot \frac{3}{3} = 39\]

(3, 3, 1) \[tgNbr = 3 \cdot \frac{5 \cdot 4 \cdot 3}{3} + 3 \cdot \frac{4 \cdot 3}{3} + 1 \cdot \frac{3}{3} = 40\]

(3, 3, 2) \[tgNbr = 3 \cdot \frac{5 \cdot 4 \cdot 3}{3} + 3 \cdot \frac{4 \cdot 3}{3} + 2 \cdot \frac{3}{3} = 41\]

\(
\vdots
\)

Once we have those tag numbers, we can store the data of the command into the right places in shared memory. Let us assume that we have only 3 long data to store for each target combination, let the starting address of this runtime in longDataSegment be \(lDStartAddress = 123\), let the starting address of this runtime in seqLength be \(lDLengthStart = 12\) and let the maximum number of longData for this setting be \(lDMaxLength = 8\).

Consider, for example, the first target combination (0,1,0):
The number of long data stored is given by

\[seqLength[lDLengthStart + tgNbr] = 3\]

the first long data of command will be stored in longDataSegment at address

\(lDStartAddress + tgNbr \cdot lDMaxLength + 0\)

the second long data will be stored at address

\(lDStartAddress + tgNbr \cdot lDMaxLength + 1\)

and the third long data will be stored at address

\(lDStartAddress + tgNbr \cdot lDMaxLength + 2\)

Thus, for the first target:

\[seqLength[12 + 3] = 3\]

the first long data of the command is copied onto longDataSegment[123 + 3 \cdot 8 + 0]
the second long data is copied onto longDataSegment[123 + 3 \cdot 8 + 1]
and the third long data is copied onto longDataSegment[123 + 3 \cdot 8 + 2].

Similarly, for the second target:

\[seqLength[12 + 4] = 3\]

\(longDataSegment[123 + 4 \cdot 8 + 0]\) receives the first long data,
\(longDataSegment[123 + 4 \cdot 8 + 1]\) receives the second long,
and \(longDataSegment[123 + 4 \cdot 8 + 2]\) receives the third long data of the command.
4.2  archive

4.2.1 Introduction

After the initialization phase, *archive* exports its service and waits in the NC loop. It exits from this loop every 5 minutes (SAVE_DATA_TIMEOUT defined in constant.h) to save data stored in shared memory onto the disk. It then comes back to the loop and waits for RPC.

4.2.2 Types and globals

All the main types used here have already been detailed in section 4.1. The global gSystTable is an array of 41 SystList (see page 26). The first element of this array is for the actif file and the fourty other ones are archives (20 for operation and 20 for rfs). See figure 4.4.

![Figure 4.3: gArchiveDetails](image)

<table>
<thead>
<tr>
<th>RF_Synchro</th>
<th>Archive sequence of 40 archive records</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurements_BA3</td>
<td></td>
</tr>
<tr>
<td>TWC_200_Low_level</td>
<td></td>
</tr>
<tr>
<td>NO_IDENT</td>
<td></td>
</tr>
<tr>
<td>NO_IDENT</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.3: gArchiveDetails

gArchiveDetails (figure 4.3) is an array of MAX_SYSTEM_NBR ArchiveDetails:

```c
typedef struct _archiveDetails {
    int index;
    String systemName;
    ArchiveSeq archiveSeq;
} ArchiveDetails;
```

with ArchiveSeq:

```c
typedef struct _archiveSeq {
    ushort length;
    ArchiveRec *value;
} ArchiveSeq;
```
Figure 4.4: Global Systems Table
Where value is the structure ArchiveRec:

```c
typedef struct _archiveRec {
    long key;
    LineDesc description;
    LineDesc date;
    LineDesc address;
    LineDesc userName;
} ArchiveRec;
```

This array is first initialized to NO_IDENT for every systemName. The archiveSeq has a length of 40 (for the 40 archives), each record is initialized to -1 for key and to Not Used for description, date, address and userName (procedure archiveSeqInitialization). This array is then filled by reading files arcSystems.details in every directory ARCHIVE/DATA/ARCHidx (with 0 < idx < 40). Procedure fillArchivegSystem opens the files arcSystems.details and stores information concerning archives records in local variables. It calls fillArchiveSeq that fills gArchiveDetails with those local variables.

If ARCHIVE/DATA/ARCH13/arcSystems.details is the only archive file containing TWC_200_low_level, information concerning archive will be found at the 13th element of the sequence, the 39 others will remain not used.

### 4.2.3 RPC services

All the services defined are used by bulles

1. **getData**: the user selects a sequence of nodes and for each node selects one target. This corresponds to a specific runtime name and a specific target combination. Process archive retrieves the data corresponding to this target combination, copies it into the structure ActiveData and sends it back to bulles. Those data can be picked up from the shared memory (actif data) or from one of the 40 memory segments of archives. This selection is made via an index (from 0 to 40) and a key (unique for each archive). If the user selects multiple targets, he can not get data since data can be different for each target. To get data, you must put the index of the archive and its key as an input parameter. Those 2 arguments can be initialized (in bulles) thanks to the next procedure.

2. **archiveConsult**: in order to read data from an archive or write data into it, the user must select the archive he wants to work with. This procedure returns the content of gArchiveDetails: description, date, address and userName and key. The user clicks on a special archive, and stores locally its index and key. He uses them for the four other services (the key for actif file is 0).

3. **saveArchive**: the user selects an input archive (it can also be the actif file) and an output archive (one that has not been used yet or the archive he wants to overwrite, if it is already used). The procedure checks if the keys are correct, if capability is sufficient, and then begins copying data from an archive to the other.

If the output archive was not used:
• copy all the files <system>.node, child, target ... from input to output archive,
• copy in new directory <system> all the <runTime> files,
• initialize the new archive by reading those files,
• add a new line in arcSystems.details,
• compute a new key for the output archive.

If the output archive was occupied:

• copy all the files ARCHoutIdx/<system>.* and ARCHoutIdx/<system>/*.* to tmp/ARCHoutIdx,
• copy all the files <system>.node, child, target ... from input to output archive,
• copy in new directory <system> all the <runTime> files,
• send an email to the previous owner to warn him that his archive has been overwritten,
• free memory for the old gSystTable[outIdx].list[systIdx],
• initialize the new archive by reading the new files,
• overwrite the line in arcSystems.details corresponding to the system,
• compute a new key for the output archive.

The first thing this procedure does is to disable actif to prevent it from writing into shared memory (even if the input archive is not the actif file). Process actif is enabled at the end of the procedure.

4. putArchiveData: this is almost the same procedure as putData for actif. The user first chooses an archive to write in, selects a sequence of nodes and a sequence of targets. Process archive writes the same data set, for every target selected, in the memory segment. The owner of the archive is warned, by email, that it has been modified. The capability of the user is checked (a user opera can not write on a rfs archive).

5. getReloadData: when bulles sends a command to the kernel, the user’s capability is checked, and the command can eventually be rejected if this capability is too low (this will not be detailed here). It is however desired to let a user reload a setting from an archive, even if he does not have the capability to send the command modifying this setting directly. We thus need to recognize between data sent by a user, and data coming directly from the archive with a security that guarantees that the user has not modified them. This security is achieved by a digital signature generated by the secure hash function MD5.

getReloadData is exactly the same as getdata. We just add what we call a digest which implements the above mentioned digital signature. The entire input command plus the output data are filtered by MD5 in order to generate this digest. This digest is compared by the kernel with the one it computes, thereby guaranteeing that the user has not modified the data.
4.3 Utilities

You can find 2 different utilities in the directory used for archive.

4.3.1 Writefile

The binary writefile is used to write data stored in `<systemName>/<runtimeName>.Data` in files `<systemName>.readable` with a readable format for only one archive (the index of the archive is the argument of writefile). It is used each time an archive is overwritten in the procedure `saveArchive`, so that a readable file is available in /tmp directory. It can also be used by anybody who wants to read the data files for any archive or for the actif (index is 0). This process shares procedures with `archive` (to read files `<systemName>.node`, child ... and files `<runtimeName>.Data`). The file `<systemName>.cmdstep` contains the different nodes constituting a command and its runtime name. For each runtime name, it writes the node sequence, for each target combination, it writes the target description and then writes the data (and parameters) corresponding to the target combination.

4.3.2 Purge

Since actif creates shared memory segments, we have to destroy them, before restarting the application. The binary `purge.exec` destroys the correct shared memory segments. (It uses the keys defined in constants.h). The number of systems in actif is the argument. The script `purge` has to be run from ARCHIVE/BIN. It kills the processes actif and archive and runs purge.exec.
Chapter 5

Practical details

5.1 How to start actif and archive

Before starting the two processes actif and archive, make sure that the shared memory segments of the previous session have been destroyed. If necessary, run purge from ARCHIVE/BIN. Then you have to start actif first, from ACTIF/BIN. If you want to start it in background, you must redirect the output to /dev/null, so you type:

RF_Actif >& /dev/null &.

After about 30 seconds, you can start archive from ARCHIVE/BIN by typing:

RF_Archive >& /dev/null &.

The two processes work on rfsrv1. If you want to change the host, you must change the name in COM/IDF/archive.hostname and recompile actif, archive, bulles.exec and rptrply.

The names of processes for the operational system are (as mentioned above): RF_Actif and RF_Archive. They are defined in the makefile and in the shell script purge. The names for the development system are actif and archive.

5.2 How to add a system to actif

When a new system is added to the application, you must add some files useful for the initialization phase.

On the DSC you just have to add a line with the system name in rpt<hostname>.system.

On the HP workstation:

1. hprr: add a line in hprSystems.system with the system name and the DSC controlling it.
2. *actif*: add a line in actSystems.system with the system name, the DSC controlling that system, and all the information about files `<systemName>.*`. You must add the file `<systemName>.child` and similarly all the files with the extensions .cmdstep, .node, .rsettings, .setsize, and .target in the directory ACTIF/DATA. You must create the sub directory `<systemName>` and in this sub directory add file `<systemName>.settings`. All those files can be found in the directory containing the flat tables extracted from the Oracle database: ~/spsrfop/graph_db/transfer/to_ACTIF/data, on rfsrv1. The files `<runtimeName>.Data` do not need to be initialized.

3. *archive*: you can not add a system in an archive directly. You must add it to *actif* and then save it (via bulles) as an archive.

### 5.3 How to modify a system in *actif*

When a system is modified (new commands added for example), it is best to remove all tables concerning this system first, then to add the new tables describing that system, as explained in the above section:

On the DSC the line with the system name in rpt `<hostname>.system` should already be there.

On the HP workstation:

1. *hprr*: the line in hprSystems.system with the system name and the DSC controlling it should already be there.

2. *actif*: add a line in actSystems.system with the system name, the DSC controlling that system, and all the information about the new files `<systemName>.*`. You must add files `<systemName>.child` .cmdstep .node .rsettings .setsize .target in the directory ACTIF/DATA. For an existing system, the sub directory `<systemName>` has already been created. Delete every file (all files `<runtimeName>.Data`) in it, and add in this sub directory, the new file `<systemName>.settings`.

3. *archive*: you can not modify a system in an archive directly.

Of course, the above procedure will **empty the full content of the active file**. In order to preserve it, you should:

1. First copy the active file into an archive.
2. Follow the above procedure.
3. Restart the processes *actif* and *archive*.
4. Reload the hardware with the archive just saved.
By doing this, all commands that were existing in the old system will be reloaded into the hardware, and then saved into the new active file. Be very careful however if you modify the ordering of the targets of an existing command, or remove some targets of an existing command, or add targets in the middle of the list of an existing command. Targets are recognized by their index only. What will be reloaded from the old archive will then not correspond to the targets selected in the MMI program.

5.4 Differences between operational and development applications

Some important points make the differences between operational and development applications.

1. Port numbers: for *rptrply* we have used the same port number: 1959. For *actif*, the operational port number is 1962 while the development port is 1969. For *archive*, the operational port number is 1963 while the development port is 1971. Those port numbers are defined in nc files (*COM/IDF*). Make sure that they are the right ones on the HP (*SPEX*ff) and on the DSC (*SOURCE/R_AND_R*) before compiling each process.

2. Shared memory: Operational and development *actif* must create different shared memory segments. They are defined by the key and by a unique file.

For operation we use:

- The file UTIL/INCLUDE/constants.h (in files actInitialization.c, arcInitialization.c and purge.c).
- The KEY 19999 (defined in constants.h) for seqLength segment (used in procedure fillSMAArray). For each system the segment is created with the key KEY + system Index.
- The key 1100 for data segments (used in procedure fillSetSize that you find in readDatabase.c).
  - long data segment is created with the key 1100 * (1 + system Index) + 1
  - float data segment is created with the key 1100 * (1 + system Index) + 2
  - string data segment is created with the key 1100 * (1 + system Index) + 3
  - long param segment is created with the key 1100 * (1 + system Index) + 4
  - float param segment is created with the key 1100 * (1 + system Index) + 5
  - string param segment is created with the key 1100 * (1 + system Index) + 6

For development we use:

- The file UTIL/INCLUDE/cntl.h (in files actInitialization.c, arcInitialization.c and purge.c).
- The KEY 9999 (defined in constants.h) for seqLength segment (used in procedure fillSMAArray). For each system the segment is created with the key KEY + system Index.
• The key 1000 for data segments (used in procedure fillSetSize that you find in read-database.c).
  – long data segment is created with the key 1000 * (1 + system Index) + 1
  – float data segment is created with the key 1000 * (1 + system Index) + 2
  – string data segment is created with the key 1000 * (1 + system Index) + 3
  – long param segment is created with the key 1000 * (1 + system Index) + 4
  – float param segment is created with the key 1000 * (1 + system Index) + 5
  – string param segment is created with the key 1000 * (1 + system Index) + 6

3. *actif* and *archive* names: since the operational and development applications work on rfsrv1, we must use different names.

  • For operation, the names are *RF_Actif* and *RF_Archive*.
  • For development, the names are *actif* and *archive*.

Those names are defined in the Makefiles and in the script purge.
### Contents

1 SPS RF Control System .......................... 1
   1.1 Introduction ................................ 1
   1.2 SPS RF Control system ........................ 2
      1.2.1 Three different layers .................... 2
      1.2.2 RPC services ............................. 2

2 Organization of directories .................... 7
   2.1 On the workstations .......................... 7
      2.1.1 COM directory ............................ 8
      2.1.2 UTIL directory ............................ 8
      2.1.3 Process directory .......................... 9
   2.2 On the DSCs .................................. 13
      2.2.1 COM directory ............................ 13
      2.2.2 UTIL directory ............................ 14
      2.2.3 Process directory .......................... 15

3 Report & Reply processes ........................ 16
   3.1 hpr ......................................... 16
      3.1.1 Introduction .............................. 16
      3.1.2 Initialization, globals and types .......... 16
      3.1.3 RPC services ............................. 18
      3.1.4 Other procedures .......................... 19
   3.2 rptrply ...................................... 20
      3.2.1 Introduction .............................. 20
      3.2.2 Initialization, globals and types .......... 20
4 Actif & Archives.  
4.1 actif  
4.1.1 Introduction  
4.1.2 Types  
4.1.3 Organization of data in the shared memory segments  
4.1.4 Example  
4.1.5 RPC services  
4.1.6 Other procedures  
4.2 archive  
4.2.1 Introduction  
4.2.2 Types and globals  
4.2.3 RPC services  
4.3 Utilities  
4.3.1 Writefile  
4.3.2 Purge  

5 Practical details  
5.1 How to start actif and archive  
5.2 How to add a system to actif  
5.3 How to modify a system in actif  
5.4 Differences between operational and development applications