SDC – 93 – 573
UM – HE – 93 – 29

UNIDAQ
Version 2.2

Software for UNIX-Based Data Acquisition

Technical Guide
Editor: R. Ball

September 22, 1993

Superconducting Super Collider Laboratory
University of Michigan
National Laboratory for High Energy Physics (KEK)
Tokyo Institute of Technology
Lawrence Berkeley Laboratory

```
************************************ ************************************
*                                                                      *
* Copyright (c) 1993                                                   *
* The Regents of The University of Michigan                            *
* Universities Research Association, Inc.                              *
* All Rights Reserved                                                  *
*                                                                      *
* Authors:                                                             *
*      R. Ball, C. Timmermans, UofM                                    *
*      Y. Takeuchi, T.I.T.                                             *
*      M. Nomachi, KEK                                                 *
*      A. Fry, C. Erbas, D. Brenner, SSCL                              *
*                                                                      *
************************************ ************************************
```

These programs are free software; you can redistribute them and/or modify them under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

These programs are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

Authors may be reached as follows:
bob.ball@umich.edu
fry@sscvx1.ssc.gov
timmer@mail.physics.lsa.umich.edu
nomachi@kekvax.kek.jp

1. INTRODUCTION

This document is part of an overview of the SDC Portable Data Acquisition System, UNIDAQ Version 2.2 . The full suite of documents includes an Installation Guide, a Technical Manual (this document), and a User's Guide. All three are available both in postscript and TEX (SLAC PHYZZX style) format in the doc sub-directory of the distributed software. The user is also referred to Fermilab PN457, the Murmur User's Guide (pn457.ps.Z in the directory containing the distribution tar files.)

The overall system software consists of a generic *template* structure, a *buffer manager*, and a set of *processes* such as a collector, analyzer, recorder, and runcontrol, which were developed using the template and the buffer manager. The processes running in UNIDAQ interact with each other by means of three entities: *command messages*, *event data* and *process data.* The interprocess message passing is handled by the template, the circulation of event data is controlled by the buffer manager (NOVA), and the process data are handled by the Status Path. The interaction of the processes is illustrated in Figure 1.

Two of the criteria considered in the design of the SDC Portable Data Acquisition System are *extendibility* and *scalability.* The template is extendible by providing a generic underlying structure on which the processes are built. The processes built on top of the template inherit all the characteristics provided by this generic structure, and if required, additional process-specific features can be added to a particular process without having to modify the kernel template. This facilitates the addition of new processes to the system. Similarly, both the template and the buffer manager are scalable up to distributed environments and higher event rates, and allow on-the-fly addition and deletion of processes.

In this report we describe the technical details of how the *template* works, the directory contents, and many other details which a more-than-casual user would need to understand the UNIDAQ workings. We start by giving the directory structure of the system (Figure 2).*

bin : contains the executable files corresponding to the system processes, such as collector, analyzer, recorder, runcontrol, xpc, logbook, etc.

murmur client : contains the murmur client process files distributed by FNAL plus the UNIDAQ data interface to them. Murmur is a distributed message logging package, with (possibly multiple) clients interacting with a single server process.

NOVA : contains all the source files, includes, and libraries, of NOVA (the buffer manager).

src : contains a set of subdirectories organized by program usage. Sub-directory `user` contains the data flow processes (collector, receiver, analyzer, recorder, dataview) which the user may need to modify. Sub-directory `control` contains the group of controlling programs of the system (runcontrol, xpc, logbook, operator), and sub-directory `etc` contains

* The documentation for NOVA can be obtained from KEK and for murmur from pn457.ps.Z in the anonymous ftp area /pub of bangkok.ssc.gov .

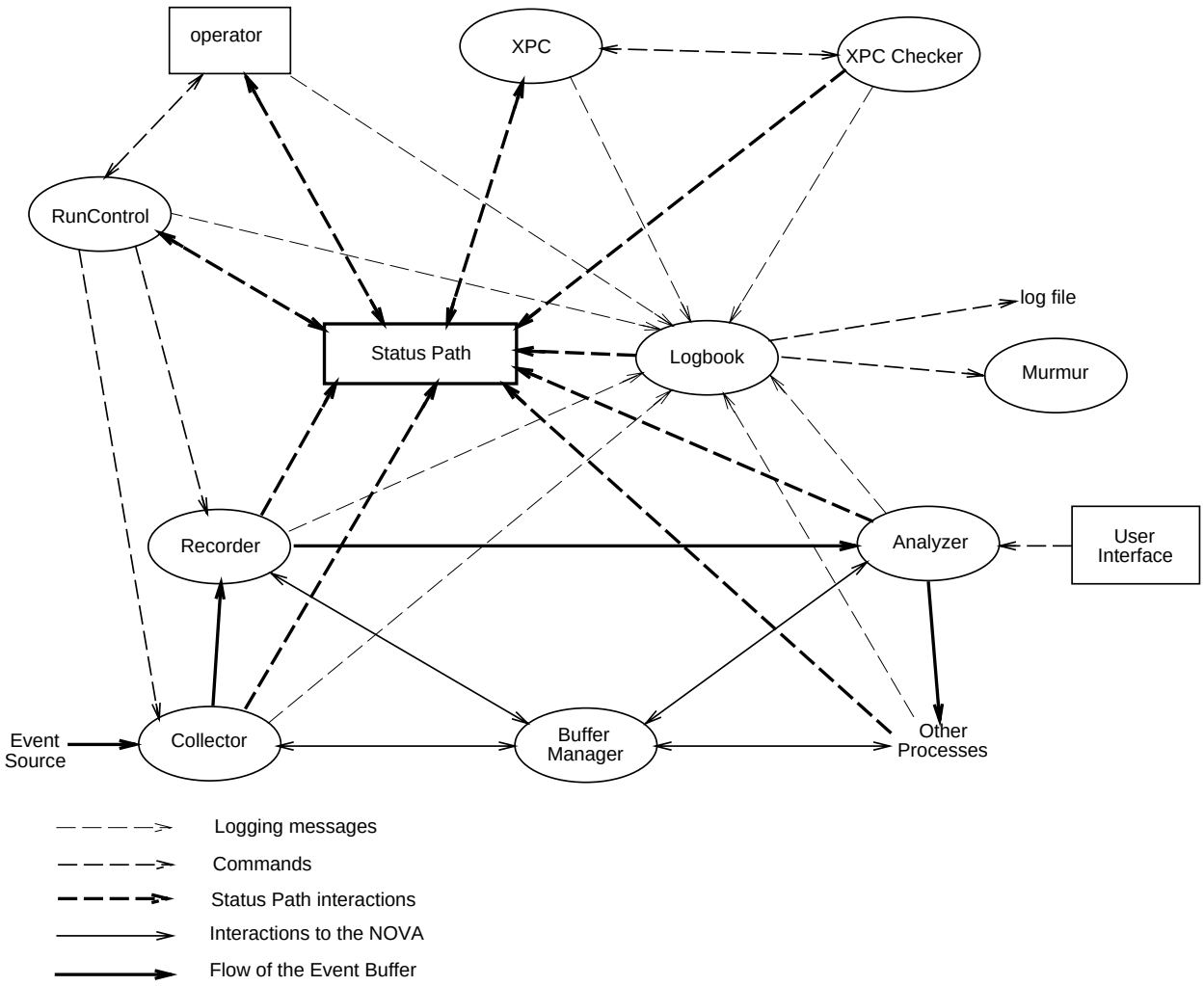

Figure 1. Interaction of the processes.

an assortment of tools and utilities. Each sub-subdirectory contains the source code and the related files specific to that process.

include : contains the include files of the template common to all the processes.

lib : contains the template libraries common to all the processes.

man : contains the man pages of the template procedures.

doc : contains the documentation of the system.

log : contains the log files.

conf : contains the files specific to a particular configuration of UNIDAQ. These include the Start_node files, runcontrol's command.list file, and the UNIDAQ startup file setup.csh .

examples : contains some example scripts and files to demonstrate how UNIDAQ works.

camvme : contains the libraries for access to CAMAC via the VME driver, written by

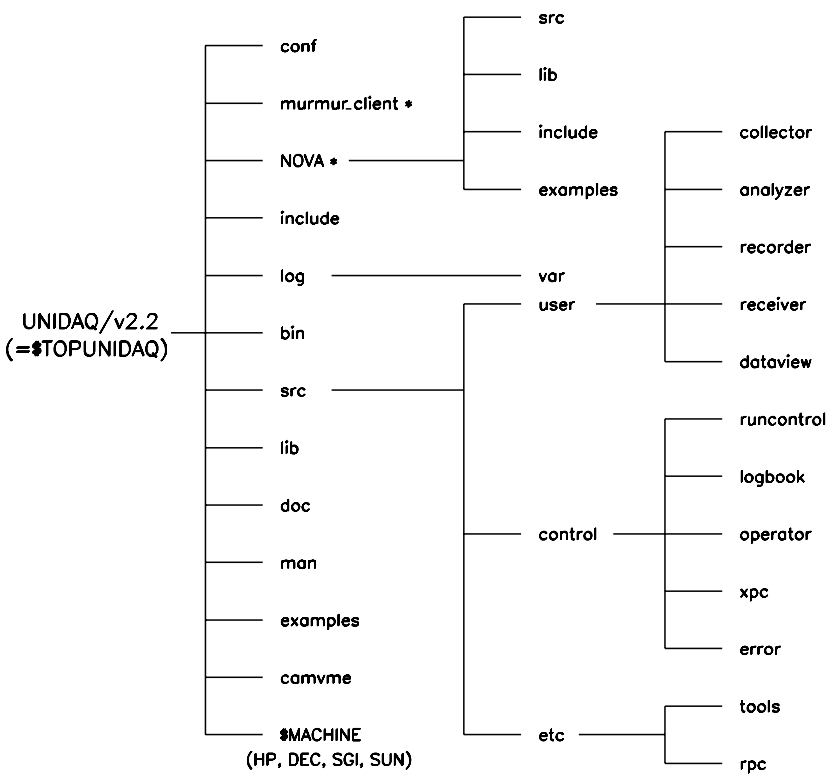

Figure 2. The directory structure of UNIDAQ 2.2 .

C. Timmermans.

$MACHINE : contains the machine specific directories for access to CAMAC. There is one such directory for each supported platform.

The libraries and source programs can be compiled and linked using the Makefiles provided in each subdirectory. The Makefile in the root directory calls the Makefiles in the subdirectories. The environment variables necessary for compilation as well as the differences between different machines, such as SUN Sparc, HP, SGI, and DEC are set using the setup.csh given in the root directory (actually a soft link into the conf directory) of UNIDAQ.

1.1 Suggested Reading

We suggest that, at a minimum, the user should read Chapters 1-5 of this manual. Later chapters detail the internal workings of UNIDAQ and how to personalize various processes.

1.2 Acknowledgements

The editor would like to thank all those who contributed text, figures and ideas to this document. These persons include, but are not limited to, R.Ball, C.Timmermans, A.Fry, D.Brenner, C.Erbas, M.Nomachi and B.Roe. E-mail help can be obtained on this package from:

MICH::BALL	bob.ball@umich.edu
MICH::TIMMER	timmer@mail.physics.lsa.umich.edu
SSCVX1::FRY	fry@sscvx1.ssc.gov
KEKVAX::NOMACHI	nomachi@kekvax.kek.jp

2. FILES REQUIRED FOR DISTRIBUTED UNIDAQ

Distributed UNIDAQ requires a $TOPUNIDAQ/conf/nodefile, which contains the list of nodes that will be used to run UNIDAQ. All of these nodes should be listed in your .netrc file in order to be able to start and stop processes on that specific node. The nodefile may look as follows:

```
mhpbob
mhptim.physics.lsa.umich.edu
```

It is possible to use only the local name, or the complete host.domain name of the machines. Using the complete host.domain name is preferred as it is necessary when leaving your local network.

The directory $TOPUNIDAQ/conf has to contain the scripts that are to be executed when starting up processes. These scripts are named "Start_node", where "node" corresponds exactly with the entry in the "nodefile". Make sure that the shell scripts can be executed (protection mod 755). The example nodefile shown above requires one to create the files: 'Start_mhpbob' and 'Start_mhptim.physics.lsa.umich.edu'. The UNIDAQ installation scripts create a nodefile and a start_node script corresponding to the name of the node on which the installation is performed. All distributed processes which do not run interactively are included in this Start_node script. To make the most efficient use of distributed UNIDAQ, the minimal Start_node script looks like:

```
1   #!/bin/csh
2   onintr -
3   source $TOPUNIDAQ/setup.csh
4   msg_server >& /dev/null &
5   xpc >& /dev/null &
```

1 : This is a C-shell script
2 : Ignore all interrupts
3 : Source the setup.csh.
4 : Start the message_server which is absolutely necessary for the IPC

5 : Start the XPC. The XPC updates the common table which may speed up communication between processes.

To make sure that the common tables within all cpu's contain all information, it is best (but not required) to start all processes (except *msg_server* and XPC) from the Start-script of the last node listed in the nodefile (In our example Start_mhptim.physics.lsa.umich.edu). The exception to this rule is nova, which needs to be started by the local Start_node script, just like the XPC and the *msg_server*. As an example, in addition to lines 1–5, one could add:

```
6       if ( -e /tmp/nova ) rm /tmp/nova
7       novad /tmp/nova >& /dev/null &
8       sleep 1
9       novand >& /dev/null &
10      ask xpc START xpc_checker@mhpbob
11      ask xpc START runco@mhpbob
12      ask xpc START logbook@mhpbob
13      ask xpc START collector
14      ask xpc START recorder
```

6 + 7 : Remove possible leftovers and start nova locally
8 : Wait to make sure the XPC and nova have started
9 : Start the network nova daemon
10 + 11 + 12 : Start processes at node mhpbob
13 + 14 : Start local processes

If the XPC is not running, you can just start the processes from the local Start-scripts (as in line 5). Note that the Start-scripts need to be located in the conf directory of the UNIDAQ-tree on the machine where the script is to be executed.

2.1 The `.netrc` File

A file called .netrc, located in your home directory, is strongly recommended if distributed UNIDAQ is run on more than one machine. This file must have 600 protection, and is formatted such that each machine entry has 3 lines of information, the "machine" name, the "login" account, and the "password" for the account, and each line begins with the quoted string shown. For example, a single entry may look like:

```
machine mhpbob
login ball
password some_junk
```

One such entry set is needed for each machine on which you will run processes, ie, those listed in the $TOPUNIDAQ/conf/nodefile file.

If the `.netrc` file does not exist or is incomplete, you will be prompted for login information by the Start program.

3. STARTING DISTRIBUTED UNIDAQ

After modifying the nodefile and updating the Start_scripts one can start UNIDAQ by typing:

```
1       source $TOPUNIDAQ/setup.csh
2       Start
```

1 : May not be needed if the sourcing of this file is done in the .cshrc file

2 : Invokes the Master Start

If the MURMUR_SERVER_ADD is not set, Start prompts for it:

```
The environmental variable MURMUR_SERVER_ADD does not exist.
This variable is filled with the IP-address of the murmur-server.
Do you wish to Exit, Continue without murmur or Set the variable (E,C,S)
(E,C,S)>>
```

After this the TOPUNIDAQ variable for each node is requested to account for different file systems.

```
Please give $TOPUNIDAQ at node mhpbob
[ <CR> = /usr/hploc/users/timmer/sdc/UNIDAQ/v2.2 ]
Started on mhpbob.physics.lsa.umich.edu
Please give $TOPUNIDAQ at node mhptim.physics.lsa.umich.edu
[ <CR> = /usr/hploc/users/timmer/sdc/UNIDAQ/v2.2 ]
Started on mhptim.physics.lsa.umich.edu
Started all nodes
```

After all local Start-scripts have been run Start prints the message 'Started all nodes'. Note that if your .netrc file is not set up properly Start will prompt for the login name and password on each remote and local node.

Starting only a single node can be done by logging into the node and typing:

```
Start_local
```

The Start_local script executes the Start_node script for the local node.

3.1 Starting Processes with Alternate Names

When a process starts, its name is normally entered into the common table (see Chapter 6) as the same as its image, eg, collector or recorder. However, it is possible to start a process and give it an alternate name. This is performed by adding the name you desire it to have following a "-n" switch, eg,

```
collector -n my_collector
```

will start the collector process, but it will be entered into the Common Table as "my_collector", and that is the name under which it will receive messages.

3.2 Receiving Event Buffers Across the Network

Those processes which receive event buffers can do so across the network. This is done by specifying "-m machine" on the command line, where "machine" is the Internet node name where *novad* and *novand* run. See Chapter 6 in the User's Guide for the practical details of doing this.

Note that processes which receive buffers in this way should have a low priority (<20) so that it will be acceptable for NOVA to skip them if a buffer is needed by the acquisition process.

3.3 Changing the Size of NOVA Buffers

Those processes which use event buffers do so with a default buffer size of 16kBytes. However, if this size is not sufficient, then it can be changed by command line option. The option is needed only on those processes accessing buffers across the net (-m option) and must match the value used when the *novad* process was started with a -b option. This is done for each process by specifying on the command line the option "-b size" where size is the buffer size to be used in bytes. For example, "-b 16384" is no different than the default buffer size already in use. Processes among the standard set which use event buffers are collector, analyzer, recorder, dataview and receiver. Some of the tools also use event buffers.

4. STOPPING DISTRIBUTED UNIDAQ

Typing Reset will start the Reset_local script on all nodes. This will stop all processes and remove shared memories, semaphores and message-queues. Again Reset will prompt for the TOPUNIDAQ variable.

```
Please give $TOPUNIDAQ at node mhpbob
[ <CR> = /usr/hploc/users/timmer/sdc/UNIDAQ/v2.2 ]
Please give $TOPUNIDAQ at node mhptim.physics.lsa.umich.edu
[ <CR> = /usr/hploc/users/timmer/sdc/UNIDAQ/v2.2 ]
```

If your `.netrc` file is incorrect you will also be prompted for login information on each such node.

Note that by typing 'Reset_local' while logged in on a single node within the distributed UNIDAQ environment, it is possible to reset just that node.

When executing Reset, the nodefile is read in, and one is queried for the $TOPUNIDAQ environmental variable for each node. Using rexec() the Reset program sends the appropriate commands to the remote nodes (and also to the local node). The commands are:

1. sourcing $TOPUNIDAQ/setup.csh
2. executing Reset_local

The Reset_local-script can be found in $TOPUNIDAQ/bin. It works on HP, SGI, SUN and DEC-station. First it executes ResetUnidaq. ResetUnidaq reads the common-table and sends the command "exit" to all processes listed in this table. Afterwards it tries to clean up the table. After executing ResetUnidaq, the Reset-script checks if novad, or any process containing the name of any of the files located in $TOPUNIDAQ/bin (with the exception of Reset and Reset_local) is running. If so that process is killed. Afterwards all shared memories, semaphores and message-queues belonging to the person executing the Reset_local are removed.

5. RUNNING DISTRIBUTED UNIDAQ IN A SINGLE CPU ENVIRONMENT

There are three ways of running UNIDAQ in a single CPU-system. The first option is to use the setup created during the installation of the software. The install-scripts create a nodefile containing only the node that you are running on. Furthermore, the installation creates a Start_node script for the local node, in which all distributed processes are started. After modifying this script to your needs, you can use Start and Reset as described before.

The second option is to run 'Start_local' to start UNIDAQ and 'Reset_local' to stop it. The nodefile in this case is not required at all. Since Start_local executes the Start_node script of the local node, the Start_node script can be modified to your needs.

The third option is to manually start all processes in some terminal window(s).

6. THE MESSAGE TEMPLATE

In this chapter we discuss the implementation details of the template. The processes running in a typical data acquisition system deal with two types of activities: (1) Issuing commands to other processes, and processing the commands received; (2) Waiting for event data, and processing (updating) event data. Commands can be issued to a process by a user (from a user interface such as the "ask" tool), or by another process using the Send_Message subroutine. Event data is received in terms of event buffers, which are circulated among processes by the buffer manager (NOVA).

6.1 System-Level commands and System-level Variables

The template provides the underlying structure necessary for the processing of command messages. It also provides a set of *commands*, and a set of *variables* to the user processes. These are called *system-level commands* and *system-level variables*, since they are defined in the template kernel. Any user process developed using the template inherits

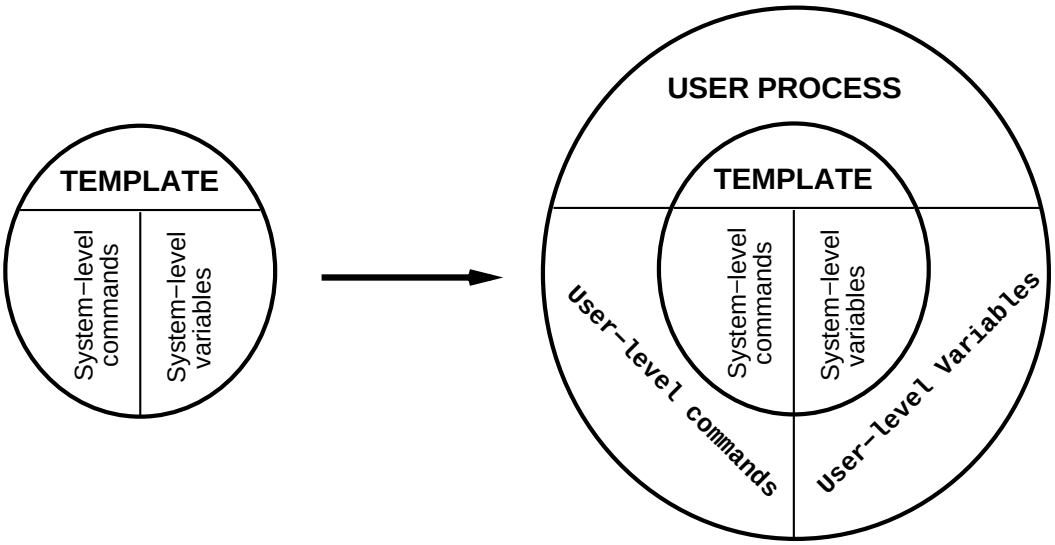

Figure 3. Developing user processes using the template.

the message passing mechanism, the system-level commands, and the system-level variables from the template. Further, the user processes are able to define their own *user-level* commands, and variables (Figure 3).

UNIDAQ 2.2 provides the following system-level commands, and variables;

System-level Commands :

ACKN, ASK, EXIT, GETPAR, MAIL, REPLY, SET, SETPAR, SHELL, SHOW, and TELL.

System-level Variables :

process_name, debug_level, status and proc_status.

6.1 a. System-Level Commands

The descriptions of the system level commands are given below: A command consists of an initial word referred to as a "verb" and arguments. Command verbs are not case sensitive in the template.

ASK <*process_name*> <*command*>, **ACKN** <*command*>, **REPLY** *Done*: These three commands are related to each other. Whenever a process receives an *ASK* command it sends an *ACKN* <*command*> to the initiating process. After receiving the <*command*>, the <*process_name*> executes the command and sents a *REPLY Done* to the initiating process, which replies with the *ACKN* command.

TELL <*process_name*> <*command*>: This command is similar to the *ASK* command. Whenever a process receives a *TELL* command, it sends the <*command*> to <*process_name*>. As opposed to the *ASK* command, this time the process does not expect a *REPLY* back.

EXIT: Whenever a process receives an *EXIT* command; it executes the exit handler, cleans up the message queue, shared memory, etc., and quits.

MAIL <*string*>: This command is used to send a <*string*> to another process. Whenever a process receives a *MAIL* command, it displays the <*string*> on its standard output.

SET <*variable*> = <*value*>: The *SET* command allows the user to assign a <*value*> to a certain <*variable*>. In order to be used in a *SET* command, the <*variable*> should be defined in either the system or user variable dictionary.

SHOW <*variable_name*>: This command is used to display the value of the <*variable_name*>. In order to be used in a *SHOW* command, the <*variable*> should be defined in either the system or user variable dictionary.

SHELL <*UNIX_command*>: Whenever a process receives a *SHELL* command, it passes the <*UNIX_command*> to the unix shell to be executed.

GETPAR <*variable_name_1*> ... <*variable_name_N*> and **SETPAR** <*assignment_stmt_1*> ... <*assignment_stmt_N*>: These two commands are related to each other. Whenever a process wants to get the value of a set of variables, it sends a *GETPAR* command. After receiving a *GETPAR* command, a process sends the values of the requested variables using the *SETPAR* command. In order to be used in a *GETPAR* or *SETPAR* command, the <*variable_name_s*> should be defined in either the system or user variable dictionary. *SETPAR* is only sent as a response to *GETPAR*. Both processes using *GETPAR* and *SETPAR* should have all the specified variables in their respective dictionaries.

6.1 b. System-Level Variables

The descriptions of the system-level variables are given below:

<*process_name*>: String. Initially this is the name of the process, either the name of the image as stored in the $TOPUNIDAQ/bin directory, or the argument of the "-n" command-line parameter when the process was started.

<*debug_level*>: Integer. Default value is zero. This variable is used to determine at what level debug print output should appear. Typical values used in the template are 10 and 20, where a debug_level $\geq$ 10 results in notification of errors in operation, and a debug_level $\geq$ 20 results in echoing of the actions of many routines of the template.

<*status*>: Integer. Default value is zero. This variable is not used.

<*proc_status*>: Integer. Default value is "ALIVE". This parameter is used to determine if the process is alive. From time to time the xpc will signal each process, resulting in an update of this variable.

6.2 User-level Commands and User-Level Variables

In addition to the system level commands and variables, the user is able to define process-specific user level commands and variables. As an example, the user-level commands and variables for the *collector* process in UNIDAQ 2.2 are given below:

User-level Commands:

BEGIN, PAUSE, RESUME, END, EXIT.

User-level Variables:

runnr, maxnevents, eventnr, eventsource, runstate, starttime, endtime, runtype, mode.

Note that the EXIT command occurs at two levels. The user-level commands have higher priority than the system-level commands. So, whenever the user issues a command which exists in both levels, then the user-level command is executed.

6.2 a. SETting Variables While Command Parsing

When executing a user-level command, system-level and user-level variables may optionally be set before the command is executed. This is done by including the "variable=value" syntax anywhere on the message line as if a SET command were being given. These strings are acted upon and stripped from the message before the message is passed to the processes command interpreter. For example, the message

BEGIN 100 comment_string eventsource=PSEUDO

when sent to the *collector* would set the variable "eventsource" to the value "PSEUDO" before passing the BEGIN command to the *collector*'s command processor.

Token replacement (see the next section) takes place before the search for SETtable variables. If "variable" still begins with a token after the first round of replacement, then this portion of the command string is passed along to the command processor with no further action taken.

6.3 Token Replacement in Commands

A command sent to a process may contain tokens. A token is a way of replacing the name of a system-level or user-level variable with its value at the time of command execution. This works by preceding the variable name with the & character. For example, the command

ask collector begin &runnr

would cause *collector* to first replace &runnr with the current value of that variable before executing the command.

Each process through which a message passes will replace one level of tokens. This allows constructs such as &&runnr to proceed to a second process before the value of runnr is actually inserted by that second process in the chain.

6.4 Handling Command Messages and Event Data: Main Loop of the Template

As noted in the introduction, the processes interact with each other by means of two entities: command messages and event data. Each process may perform the incoming commands and process the event data. The template main loop provides the necessary

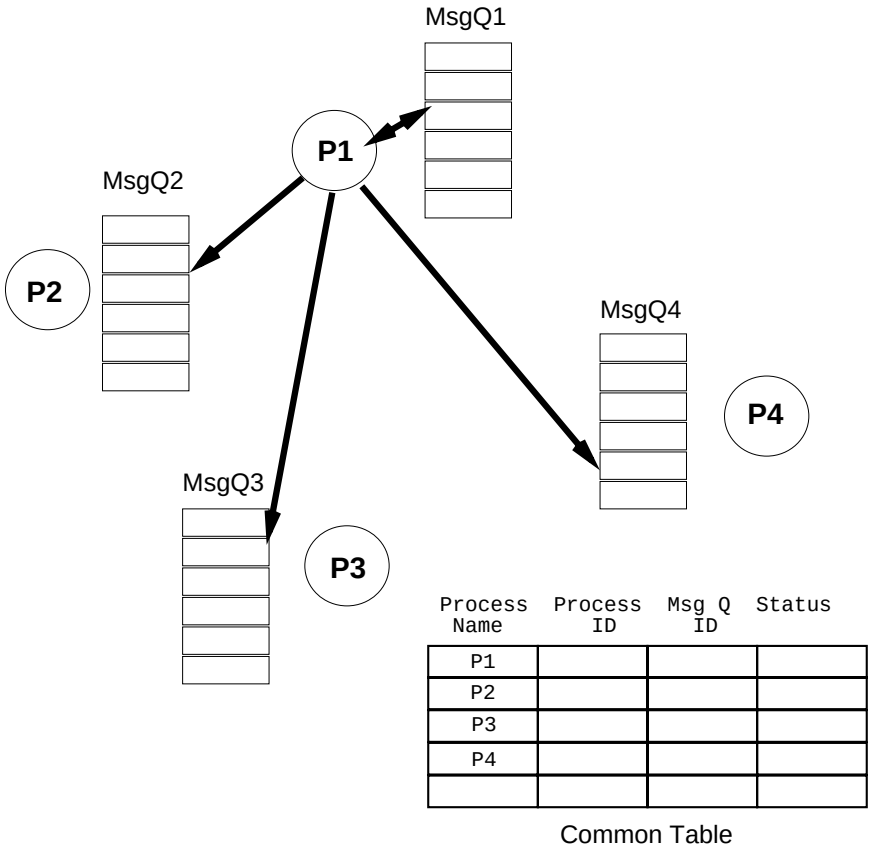

Figure 4. Interprocess message passing in the template.

mechanisms to handle these two activities without having to deal with the details of the underlying communication and synchronization structure. In this section, we discuss the implementation of the template main loop and the underlying communication structure.

The main loop structure of the template is given below.

While *(the process is alive)* {
    interrupt_flag = 0;
    If *there is a command* **then** process the command;
    If *there is an event* **then** process the event;
    Call the idle procedure;
}

All the procedures for handling the commands are provided by the template. The user is supposed to prepare user-level commands, as described in the previous subsection. The procedure for processing the events *(Process_Event())* is provided by the user, and a function pointer is passed to the Template_Main() procedure. Similarly, the idle procedure *(Process_Idle())* is also provided by the user. However, if the Process_Idle() is not provided by the user, then the default idle() procedure, which sleeps for one second (or until receiving a signal) is called. Process_Event() and Process_Idle() procedures are called once every loop cycle.

6.5 Message Passing Within a Single Processor

The message communication between processes within a single cpu is handled by message queues. Each process has a message queue. The message queue id for each process is stored in a *common table*, which can be accessed by all the processes in the system. Whenever a process P1 wants to send a message to process P2, it gets the message queue id of P2 from the common table and writes the message to the message queue of P2 (Figure 4). Further, it sends a SIGUSR2 signal to P2. In the program level, this can be done by calling the *Send_Message()* procedure. This procedure has two parameters: *process_name* and *message.* For example, if one process wants to send a link command to the analyzer, this can be done as follows:

```
strcpy(process_name, "analyzer");
strcpy(message, "link histogram1.f");
Send_Message(process_name, message);
```

The equivalent of this at the shell level is the command

```
ask analyzer link histogram1.f
```

6.6 Message Passing Across a Network

The same commands to send messages within a processor, both at the program level and at the shell level, can be used to send messages across the network to other processors. The call Send_Message can be used in two different ways:

a) Send_Message(proc@node,message)

b) Send_Message(proc,message)

The first call does not require the system to try and find out in which node process 'proc' is running, since that node is obviously known. The second call does require some algorithm to locate process 'proc'. The system-actions to both calls are listed below:

6.6 a. Case a

If proc1 calls routine Send_Message as: Send_Message(proc2@node2,message), the destination is prepended to the message. The resulting string is sent to the *msg_server* at node2. This server splits the string into the destination and the message. Afterwards it adds the message to the message queue of the destination process and signals the destination process that a new message has arrived. The message server sends a return code back to the sending process. Using this code the common table at node1 is updated.

6.6 b. Case b

First the location of proc2 needs to be found. To do this, the common table is first used to locate proc2. If that fails brute force is used, and all nodes are checked to find if the process is running on that node. Afterwards, if the node is known, the same actions as described above in case a are performed to send the message and update the common table. We will describe both ways to find the location in somewhat more detail.

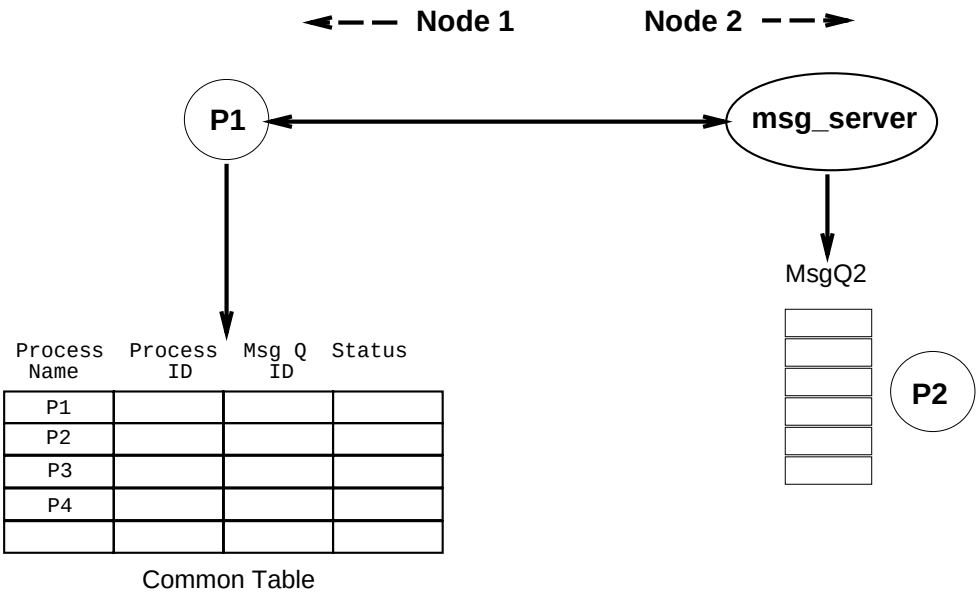

Figure 5. Interprocess message passing across the network.

1) The local common table is checked to see if process 'proc2' is listed in there. If the process is not listed in the common table or is listed to be dead, the brute force method will be used. If the process is listed to run locally, the message is sent locally (without the use of any servers). If the process is listed to run on a remote node, the message is sent in the same way as described above. If the reply from the remote *msg_server* indicates the process is not there, then the common table is updated and brute force is used to locate the node where 'proc2' is running.

2) Brute force: The nodefile is read and the server on each node is asked to check if the process is running there. If one server responds in an affirmative way, the message is sent as described above and the common table is updated.

6.7 Updating the common table

The XPC should be started at all nodes as well as the *msg_server* process. A change of state (starting/stopping) of a process on a node will cause the XPC to send a message to the XPC running on each of the other nodes. This results in an update of the common table on all nodes.

Even if the XPC is not running, the common table can hold information on processes on other nodes. As described above, by sending messages across the net the common table will be updated.

Note that there is no guarantee that the common table is accurate regarding all nodes and all processes. For example, when *msg_server*s or XPC's are not functioning, modifications are not passed to all nodes within the distributed UNIDAQ.

When a single node is Reset using Reset_local, it first sends an "EXIT" command to all processes. This will cause the XPC to send common-table updates across the net. Since it

takes a while before the XPC is actually killed, the common tables on all machines should have all changes.

6.8 Locations of Some Files Related to Message Passing

Template related procedures, including the main loop, message passing procedures, command parsers, signal handlers, etc., are defined in cmdlib.c, which is located in the lib directory. This file (cmdlib.c) also contains the procedures for the system level commands. The header descriptions for the system level commands and variables are in variable.h, located in the include directory. Further, the general header file template.h is also placed in the include directory.

7. STATUS PATH

The Status Path is a system of shared memories, semophores and rpc's that simplify the sharing of data among processes. In a distributed UNIDAQ environment, each machine holds the process data of the local processes in its shared memory. Sharing this data across machines is done with the UNIDAQ *msg_server*. The shared memory within each machine (see Figure 6) contains the following elements:

1. a buffer to store all data
2. a control structure that keeps track of which process is using which part of the buffer.
3. a control structure to keep track of empty buffersegments.

A "proc_info" block contains the name of the process associated with it and a pointer to the first "buf_info" block attached to this process. These "buf_info" blocks point back to the "proc_info" block and contain the first and last address of the "buffersegment" that it controls. These "buffersegments" consist of datasegments in which the data is stored. These elements are all detailed in the figure.

When a process starts up, it will either create the Status Path or link to it. Afterwards, it will locate space in the Status Path to store its user and system level variables. If space is not to be found within the shared memory, more memory will automatically be created. During this startup procedure, only this process will be able to modify the control structure of the status path. Other processes that try to do so will wait. This is controlled by the use of a semophore. After a process receives its chunk of memory, it will be the only process writing to it. All other processes will only be able to read from it.

7.1 Reading from and Storing Variables to the Status Path

There are two routines to read values from the Status Path, and two routines to write to the Status Path. Reading from the Status Path can be done using the following:

1. read_value_from_sp(char *procname, char *varname, short *type, char *value)
2. read_all_values_from_sp(char *procname, char *varname, int varsize, short *type, char *value, int valsize, int *n_var)

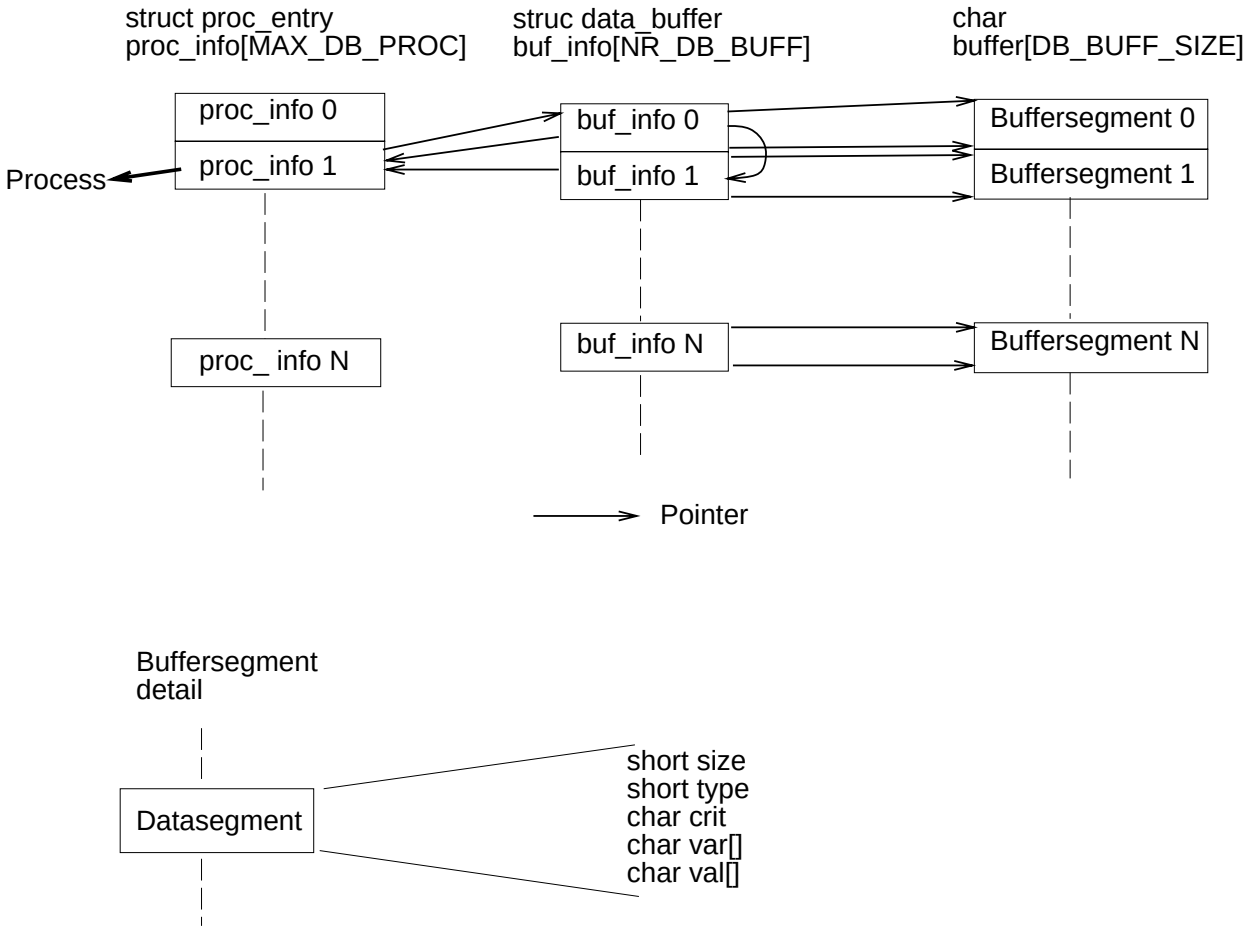

Figure 6. The Status Path shared memory structure.

In the first case, only one variable will be read. Input parameters to this routine are:

- char *procname, the name of the process one wants to read the variable from. This name can be given in the form "process@node" or simple "process", in which case the system will find the correct node.
- char *varname, the name of the variable one wants the value from.

Output parameters are:

- short *type, the type of the variable (INTEGER, REAL, STRING)
- char *value, the value of this variable

The second routine reads all variables stored for this process. The "varname" parameter in this case is an output parameter, containing a whole array of variable names. The maximum length of these variable names is the input parameter "varsize". This parameter determines the offset to the beginning of each variable name. The "type" and "value" parameters are output arrays containing lists of values. The "valsize" is again an input parameter, determining the maximum available size for each of the values. Finally, n_var is an output parameter, which gives the number of parameters the process has stored in the Status Path.

Correspondingly, it is possible to store values in the Status Path using:

3. set_value_in_sp(char *varname,char *value)
4. set_all_values_in_sp()

Again the first of these routines sets just one parameter in the Status Path. The second routine sets all variables in the Status Path to the same value as known by the template.

Note that the template does not automatically update the Status Path. When parameters change, the user should make sure that the Status Path is updated. An exception to this rule is that the "set" command also updates the status path.

7.2 Critical Variables

Some variables are critical for the experiment, and one wants to save those even when UNIDAQ stops. The routine "Crit_Var(char *name)" takes care of that. This routine needs to be called after Template_Init(), and before Template_Main() (see Figure 12 in Chapter 10) for each variable 'name' that needs to be saved and read back. This routine checks the existence of the file $TOPUNIDAQ/log/var/process_name. If that file exists it will read the appropriate variables from this file. When the process stops, it will write the variables back to disk. If the process gets killed, the XPC will write them back to disk.

Following all calls to Crit_Var(char *name), the user should also call read_value_from_sp (ownpname, char *name, &(short dummy), var_name) for each variable named in a Crit_var call, which copies the value from the Status Path into the local process location. "ownpname" is a literal, the name of this process, "char *name" is the same in each call, and "var_name" is the local variable as declared in the `var_dict_user` array corresponding to "*name".

8. EVENT BUFFERS

Collector interacts with the event source. When an event occurs, *collector* accesses the event source (for example, a CAMAC or a VME crate), reads the related signals from the channels, and stores the event data into a NOVA buffer. This part must be written by the user. This data is called an *event_data record*, and is circulated among other system processes which request event_data such as *recorder* and *analyzer*. Further, whenever the *collector* process receives a BEGIN command, it prepares a *begin_run record*, and ships it to the other processes. Similarly, a *pause_run, resume_run* or an *end_run record* is prepared for each PAUSE, RESUME or END command, respectively. Each record type is distinguished from the others by the record format. At the present time, the system contains 7 types of records:

1. Event data record
2. Begin run record
3. Pause run record

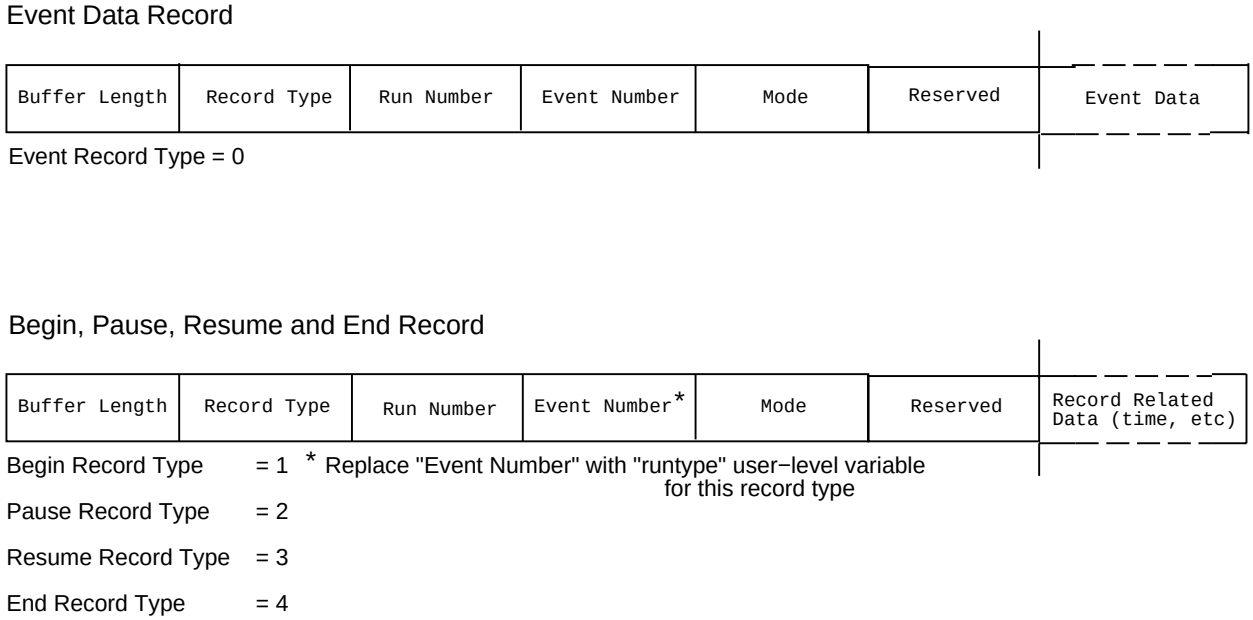

Figure 7. Record Type

4. Resume run record
5. End run record
6. Calibration record
7. Environment record

Among these record types, the calibration and the environment record types have not yet been implemented. The formats of the other record types are given in Figure 7. At this time, the "Event Related Data" of the Begin, Pause, Resume and End records consists of a character time-stamp and the comments supplied with the equivalent commands to the *collector* process, each NULL terminated and padded to a multiple of 4-bytes.

8.1 Number and Size of Event Buffers

By default, NOVA uses 16kByte event buffers, and UNIDAQ will request records up to max_length words (see Chapter 10) from each buffer until no more such can be filled, or a maximum timeout period has expired without an event, at which time the buffer is passed down the priority chain and another is obtained. This maximum buffer size can be changed using the "-b" command line switch when the process is started (see Section 3.3).

The value of max_length is set as *collector* starts by calling the user_init routine, which is coded in user_event.c of the collector directory. The default value is 150 words, which includes the event header words of figure 7.

The significance of the NOVA buffers is hard-coded as 70. This means that any process which calls for NOVA buffers with a priority greater than or equal to 30 will receive all data buffers. Processes receive data buffers in order, with those processes having a higher priority (set in the UNI_nova_open call) receiving the buffers first. *Analyzer*-style processes

should have priorities <= 29 so that, in effect, they will simply sample the buffers, and not slow down data acquisition by forcing NOVA to always give them all data buffers. Processes which take the longest time working on event buffers should have the lowest priority.

9. SIGNALS, SEMAPHORES AND OTHER SYSTEM RESOURCES

UNIDAQ uses a number of signals, semaphores, shared memories and message que resources in the performance of its tasks. In this chapter we summarize those used, and the reason for their use.

9.1 Signals

Five different signals are used by UNIDAQ. The particular signals in use were chosen to avoid conflict with PAW and X11, and yet to be commonly available on all UNIDAQ platforms.

Signals used by UNIDAQ	
Signal Name	Usage
SIGUSR2	This signal accompanies messages sent from process to process. It is used to wake the receiving process up from an idle state so that the process can read and act upon the message.
SIGQUIT	This signal is sent by the XPC to test the status of each process from time to time. The receiving process writes to its Status Path in response.
SIGALRM	When entering the Idle() routine, the template generates a SIGALRM after the specified timeout.
SIGTERM	This signal is sent by the XPC to set the interrupt_flag variable of a process. Setting this flag is a message to the process that a time consuming operation should be stopped. It is up to the programmer of each process to actually handle this stoppage.
SIGINT	This signal results in a call to the Exit() routine, which detaches the process from the Common Table and terminates execution.

9.2 Message Queues

Commands are transferred from process to process using message queues. As implemented each process has its own message queue from which it reads. Other processes which try to communicate with this process read the queue number from the Common Table and

then write their messages to the appropriate queue. This is detailed in Chapter 6. Message queues are not a fast way to transfer messages and there are some system dependent restrictions:

The restrictions are for the HP,DEC,SUN and SGI.

msgmax:	message max. size.	Default 8192 bytes (SGI 16384)
msgmnb:	max nr of bytes on a message queue.	Default 16,384 bytes (SUN 2048, SGI 32768)
msgmni:	max. number of msgq identifiers.	Default 50
msgtql:	number of message headers.	Default 40 (SUN 50)

9.3 Shared Memory

UNIDAQ communication depends on shared memory as much as it depends on message queues. Shared memory is not only used for the Status Path, but also for the Common Table so that processes have a specified place from which to find the information needed to communicate (node and message queue). Furthermore, another shared memory contains a header for this common table. The header has key 186, the Common Table has key 185. The Status Path may contain multiple shared memories. The keys of these shared memories start with 201.

The only process in need of its own shared memory is the XPC. Since this memory is used to save information for new invocations of the XPC it cannot be a simple IPC_PRIVATE shared memory, but a specific key needs to be provided. The key for the XPC shared memory is 49456.

NOVA uses an IPC_PRIVATE key, which the buffer manager process ports to connecting processes.

9.4 The RPC System

We are using SUN Remote Procedure Calls to extend the message template and Status Path principles across the network. This is not implemented as a generalization of the UNIX message queues, shared memory and semaphores, but as a generalization of the UNIDAQ routines. This means that, when executing a Status Path routine or Send_Message() on a remote machine to the local machine, that information is actually handled by the local *msg_server* process, and the resulting message from this routine call is then transferred to the local process via its message queue.

9.5 Semaphore Usage

Semaphores are used by several processes to coordinate the use of shared memories.

Semaphore Usage in UNIDAQ	
Where Used	Usage
Status Path	The Status Path is locked using a semaphore (key 200) when a process needs to write into the Status Path control structures. This ensures a correct handling of these control structures.
analyzer	A PRIVATE semaphore is used to synchronize/separate PAW commands from the user interface from those from other processes which are using the PAWC command.
collector	Creates and deletes a PRIVATE semaphore without actually using it.

10. MODIFYING THE COLLECTOR PROCESS

The user can add a new process to the system by following the steps given below. Each step is exemplified with the *collector* process. It is assumed that the $TOPUNIDAQ/src/user/collector directory does not yet exist in this example.

This example shows how to add both user-level commands and variables to a program. As such, it is a more complicated task than the average user will usually want to undertake. However, that complication aside, it exemplifies the procedure for creating ones own programs.

The example begins with copying two blank files, user.c and user.h, from the lib subdirectory of $TOPUNIDAQ, then modifying them to your needs. For a complete, well-documented sample of making such changes, see example.c and example.h in the examples subdirectory of $TOPUNIDAQ.

Step 1. Create a new subdirectory under $(TOPUNIDAQ)/src/user, and copy a blank template and its header from $(TOPUNIDAQ)/lib to the new subdirectory.

```
1> cd $TOPUNIDAQ/src/user
2> mkdir collector
3> cd collector
4> cp ../../../lib/user.c collector.c
5> cp ../../../lib/user.h collector.h
```

A blank template source contains the code given in Figure 8. The include of "user.h" should be changed to "collector.h" once the files are copied.

```
#include "user.h"          <== Change this line as instructed

main (argc, argv)
int argc;
char*argv[];
```

```
{
  if ( Template_Init(get_name_from_args(argc,argv)) < 0) {
    printf("Process of name %s already exists\n",
          get_name_from_args(argc,argv));
    printf("You can modify the process name by using the -n option\n");
    exit(0);
  }
   Template_Main((int (*)()) Process_Event, NULL, (void(*)()) Process_Idle);
}
```

Figure 8. A blank template source code (user.c).

A blank template code contains a main procedure, which calls *Template_Main()*. Whenever an event occurs, the template calls the Process_Event() procedure. So, the user code to process an event is written into this procedure.

A blank template header (user.h) is given in Figure 9.

The template header (user.h) is provided to allow the user to specify his own commands and variables. The structures *var_dict_usr* and *cmd_dict_usr* are used to define the user-level variables, and the user-level commands, respectively.

```
#include "generic.h"
struct var_dict_type  var_dict_usr[20];
struct cmd_dict_type  cmd_dict_usr[20];
int cmd_number_usr = 0;
int var_number_usr = 0;
```

Figure 9. A blank template header code (user.h).

Step 2. Specify the user-level variables in the header file. As an example, two user-level variables of the collector are defined in Figure 10. These variables will become accessible to SHOW and SET system-level commands.

```
 char eventsource[20] = "CAMAC";                          <1>
 int runstate = END_STATE;                                <2>
struct var_dict_type var_dict_usr[20] = {
        "eventsource",STRING,(int)eventsource,            <3>
        "runstate",INTEGER,(int)&runstate                 <4>
};
 int var_number_usr = 2;                                  <5>
```

Figure 10. Defining user-level variables.

The lines that are modified or added to the blank include file are italicized and numbered. The definitions of the user-level variables called eventsource and runstate are done in lines <1> and <2>. Then, these variables are added to the user-variable dictionary in lines <3> and <4>. The dictionary contains the following information for each variable:

<variable_name>, <variable_type>, <pointer_to_variable>.

<variable_name> is defined as a string constant, and the variable is referred to by this name. Variable type can be INTEGER, REAL, or STRING. If the <variable_type> is INTEGER or REAL, then <pointer_to_variable> is defined as follows:

(int) &<variable>.

Otherwise, if the <variable_type> is STRING, then the definition of the variable becomes

(int) <variable>.

Finally, the number of variables in the user dictionary is defined using var_number_usr as shown in line <5>.

Step 3. Specify the user-level commands in the header file. As an example, the user-level commands of the *collector* are defined in Figure 11.

```
int Begin();                                                    <1>
int Pause();                                                    <2>
int Resume();                                                   <3>
int End();                                                      <4>
struct cmd_dict_type cmd_dict_usr[20] = {
        "BEGIN",(int(*)()) Begin,                               <5>
        "PAUSE",(int(*)()) Pause,                               <6>
        "RESUME"(int(*)()) Resume,                              <7>
        "END",(int(*)()) End                                    <8>
};
int cmd_number_usr = 4;                                         <9>
```

Figure 11. Defining user-level commands in the header file.

In this example, there are four user-level commands defined in the header: BEGIN, PAUSE, RESUME, and END. The procedures corresponding to these commands are defined in lines <1>, <2>, <3>, and <4>. Each of these procedures will be coded in collector.c as shown in step 4. Lines <5>, <6>, <7>, and <8> illustrate how each entry in the command dictionary contains the following two items:

<procedure_name>, <function_pointer>.

<procedure_name> refers to the name of the command procedure and is defined as a string constant. <function_pointer> is a pointer to the command procedure.

Finally, the number of user_level commands are defined using the cmd_number_usr variable, as shown in line <9>.

Step 4. Write the source code of the procedures corresponding to the user-level commands. As an example, the procedure for the user-level commands of the *collector* process are given in Figure 12. One procedure should be written for each of the commands, BEGIN, PAUSE, RESUME, and END. Further, Process_Event() should also be defined in this step.

When the system starts running, the *collector* enters the Process_Event() procedure every loop cycle (see Chapter 6 for the description of the loop cycle). Further, whenever the *collector* receives a BEGIN, PAUSE, RESUME, or END command, it executes the Begin(), Pause(), Resume(), or End() procedure. Template passes two parameters to these procedures: *header,* which contains the name of the process sending the command, and *arg_string*, which corresponds to the parameters of the command. For example, if *runcontrol* sends a "begin 74859" command to the *collector*, the parameters of the Begin procedure take the following values: header = runcontrol, arg_string=74859.

```
main (argc,argv)
int argc;
char *argv[ ];
{
    Template_Init(get_name_from_args(argc,argv));
    Open the buffer manager;
    Declare critical variables;
    Rretrieve values of critical variables from the Status Path;
    Template_Main((int (*)()) Process_Event, NULL, (void (*)())Process_Idle);
}

int  Process_Event()
{
    Read the event data from the event_source;
    Get a buffer from the buffer manager;
    Fill the buffer with event data;
    Return the buffer to the buffer manager;
}

int  Begin(header, arg_string)
char *header, *arg_string;
{
    Initialize the event_source;
    Start the data collection process;
}

int  Pause (header, arg_string)
char *header, *arg_string;
{
    Pause the data collection process;
}

int  Resume (header, arg_string)
char *header, *arg_string;
{
    Resume the data collection process;
}

int  End (header, arg_string)
char *header, *arg_string;
{
    Stop the data collection process;
}
```

Figure 12. Procedures corresponding to the user-level commands of the collector.

Step 5. Write a Makefile to compile and link the new program. The make file should generate the executable file into the $TOPUNIDAQ/bin directory. An example Makefile for the *collector* is given in Figure 13. Further, the name of the new directory should be added to the SUBDIR and CLDIR list of the Makefile in the $TOPUNIDAQ/src/user directory, ie, the directory immediately above that in which you are working.

```
    MAINDIR = $(TOPUNIDAQ)
    CAMDIR = $(TOPUNIDAQ)/$(MACHINE)/vme/camac

    TLIB = -L$(MAINDIR)/lib -ltmplt
    CLIB = -L$(CAMDIR)/lib -lcamac
    NLIB = -L$(MAINDIR)/NOVA/lib -lnova -L$(MAINDIR)/NOVA/netlib -lnetnova

    LIBS = $(MAINDIR)/lib/libtmplt.a  $(MAINDIR)/NOVA/lib/libnova.a \
           $(MAINDIR)/NOVA/netlib/libnetnova.a

    INCLUDE = -I$(MAINDIR)/include  -I$(CAMDIR)/include \
           -I$(MAINDIR)/NOVA/include
    OBJS = collector.o camac_event.o pseudo_event.o user_event.o

    CFLAGS =
    DFLAGS = -D$(MACHINE)
    CC = cc $(CFLAGS) $(DFLAGS) $(INCLUDE)

.c.o    :
    $(CC) -c $<

all    : $(MAINDIR)/bin/collector
$(MAINDIR)/bin/collector: $(OBJS) collector.h $(LIBS)
    $(CC) $(OBJS) -o $@ $(TLIB) $(CLIB) $(NLIB)

clean    :
    -rm -f $(MAINDIR)/bin/collector
    -rm -f *.o
```

Figure 13. An example Makefile for the collector.

In this example, MAINDIR and CAMDIR point to the main directories of UNIDAQ and the CAMAC device driver. TLIB, CLIB, and NLIB are the library directors of the template, CAMAC, and NOVA, respectively.

Once the Makefile is ready, compile and link the new *collector* process by typing the command "make" at the shell prompt. Clean up by deleting old object files using the command "make clean".

10.1 Standard Subroutines Callable from Collector

There are eight subroutine which can be called from the *collector* process. The first six listed below are called by the standard coding of *collector* and the final two are available for use but not called by default. All are coded in the user_event.c file, which is the suggested source the user should modify for a particular hardware instance.

- user_begin (runtype) – Called to process a BEGIN command.
- user_resume (mode) – Called to process a RESUME command.
- user_event (mode, max_length, event_length, buf) – Called to process an event for data acquisition.
- user_pause (mode) – Called to process a PAUSE command.
- user_end (runtype) – Called to process an END command.
- user_init (max_length) – Called to set the maximum length (max_length) which a user_event call may fill.
- user_exit ( ) – Called to process whatever the user would like at program exit.
- put_record_ (data, length, type, event_number) – This routine may be called to insert an event buffer into the NOVA event stream. The data portion of the event is filled with the first "length" words from array "data". The record type (see figure 7) is filled with the value of argument "type" as the event number is also filled from argument "event_number".

"Mode" and "runtype" in the calls above are the user-level *collector* variables of those same names. "Event_length" is returned from user_event, and is the number of words placed in the event buffer. "Buf" is the Event Data portion of the Event Data Record (figure 7).

11. ANALYZER

Dynamic linking varies from machine to machine. Right now, it is not implemented on the DECstation, just on the HP, SGI and SUN.

11.1 Analyzer on the SUN, SGI and HP

When the *analyzer* receives the *LINK* command, it first compiles the file. On the SUN it uses *f77 -c filename* or *cc -c filename* depending on the file extension. On the HP it uses *f77 +ppu +z -c filename* or *cc +z -c filename* respectively. On the SGI it uses *f77 -c -G 0 filename* or *cc -c -cckr -G 0 filename* respectively. Then *analyzer* links the object file with the following system call: *ld -o [filename] -Bdynamic [filename].o* on the SUN, or *ld -o [filename] -b -E [filename].o* on the HP; this step is not necessary on the SGI. Finally it opens the linked file with the *dlopen* command on the SUN, or with the *shl_load* command on the HP, or the *ldopen* command on the SGI.

These commands are formalized in the file `analyzermake` found in the analyzer directory. It can be edited to change the options before the *analyzer* LINK or ADD commands are issued, but we do not recommend this course of action.

11.2 Analyzer on the DECstation

The DEC version of the analyzer requires one to write two Fortran routines:

ANA_INIT: This routine is called before entering the event loop.

ANA_EXEC: This routine is called at every event.

As an example, consider the following routines:

```
SUBROUTINE ANA_INIT()
COMMON /PAWC/HMEMORY(10000)

CALL HLIMAP(10000,'SDC1')
CALL HBOOK1(20, 'ADC0       $', 50, 0, 300., 0.)
CALL HBOOK1(30, 'ADC1       $', 50, 0, 300., 0.)
CALL HBOOK1(40, 'Dummy data $', 50, 0, 300., 0.)
RETURN
END
```

The HLIMAP call makes sure that the histograms are written in shared memory, of name SDC1. After this call the histograms are actually created.

```
SUBROUTINE ANA_EXEC(IDATA, IHEADER)
INTEGER IDATA(*), IHEADER(*)
COMMON /PAWC/HMEMORY(10000)

CALL HF1(20, FLOAT(IDATA(3)), 1.)
CALL HF1(30, FLOAT(IDATA(4)), 1.)
CALL HF1(40, FLOAT(IDATA(40)), 1.)
RETURN
END
```

The routines fill the histograms for each event. Note in this example that when a run is stopped, and a new run started, the data is added to the same histograms. Two arrays are passed to the routine; IDATA contains the event data only, and IHEADER contains the six word header information, which precedes the event data in the complete buffer. See figure 7 for details of the structure of the header.

After creating your own FORTRAN file, the Makefile must be modified to compile your routines and not the examples.

11.3 Never the Twain Shall Meet

At this time, dynamic linking of *analyzer* works on the HP, SGI and SUN platforms, but not on the DECstation. The shared memory version works on the DECstation, and should work on the SUN, but it will not work on the HP because of a compiler/HP-UX limitation. We have not tried it on the SGI.

12. DATAVIEW PROCESS

The *dataview* process is completely event driven. It responds to the system level UNIDAQ commands, but has no user level commands or variables. Its purpose is to display event parameters on the screen. File $TOPUNIDAQ/src/user/dataview/datawin.h contains the configuration for the *dataview* process. For each variable you want to show, two fields have to be filled in the dview[ ] array. The first field contains the text to be displayed in front of the variable, and the second field contains the name of the subroutine which converts the data into an appropriate string, *e.g.*,

```
"Variable text", (int(*)())routine_name,
```

These conversion routines are coded in file `dataconv.c` in the same directory. The routines have the following form:

```
routine_name ( header, data, txt )
int *header, *data;
char *txt;
{
}
```

Using the header and data from an event, this routine should fill the `txt` string with the value to be displayed. Examples are coded in the `dataconv.c` file.

13. RUN CONTROL PROCESS

The *runcontrol* process can be viewed as a command interpreter. It is the means of doing the (possibly) complicated list of actions associated with such innocuous commands as "Begin Run". The commands and their actions are defined in an input file. The name of this input file is listed in the file 'files.h' located in the $(TOPUNIDAQ)/src/control/runcontrol directory and that name is bound into the *runcontrol* image. This default input-list is named 'command.list' and is located in the `conf` directory.

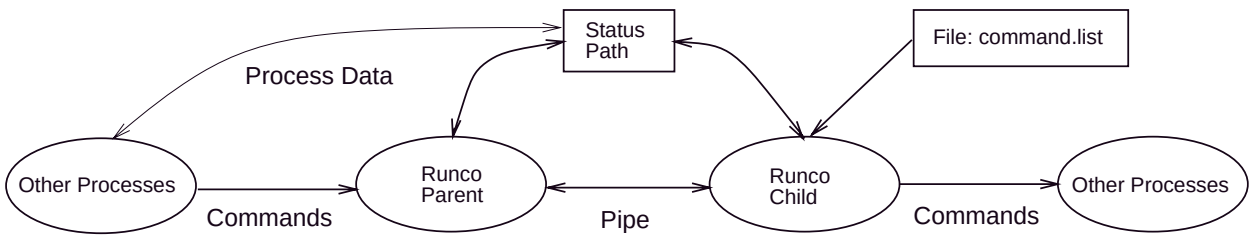

Figure 14. Runcontrol's Parent and Child Interaction.

If a command is given (which is not a system, user level, or shell command) a child process is created. This child process reads the input file and tries to execute the command. This command can consist of a set of operations that needs to be executed either in parallel or sequentially. If parallel processes are required the child tries to fork more child-processes. If it fails to do so the operations are performed sequentially.

The low level operations (such as send_process) are subroutines which are compiled and linked to the *runcontrol* executable. Communication between parent and child uses a pipe, to enable the parent to clean up when the child finishes, and the Status Path (Figure 14).

Besides responding to commands, the *runcontrol* parent process frequently (every five seconds) pulls current run parameters from the Status Path. This interval is determined by the "DATATIME" parameter, which can be found in the file `runco.h`.

13.1 The `command.list` file

Runcontrol reacts to commands received from other processes. These commands usually require *runcontrol* to do several things (e.g., during the start of a run several processes need to be signalled to start the run, parameters need to be set, etc.). Therefore, macros are developed. These macros are written in the file `command.list` and read by *runcontrol* every time *runcontrol* receives a command that might require it to perform multiple actions.

The definition of a macro starts with a #NAME, where NAME is the name of the macro. This might be followed by 'variable-name' in which case any occurrance of 'variable-name' in the macro definition will be substituted during execution. For example,

#BEGIN_RUN_WARM rnum

is a macro definition. This macro is executed when runcontrol receives a command such as:

BEGIN_RUN_WARM 10

in which case all occurrances of 'rnum' in the macro definition are replaced by '10'.

The next line of the macro-definition contains either:

- SEQUENTIAL{ or
- PARALLEL{

In the first case the commands are executed sequentially and in the last case in parallel.

Afterwards we get the macro-body containing subroutines and their arguments or references to macros. This body may also contain nested 'PARALLEL' and 'SEQUENTIAL' parts, in which case a group of commands will be treated as a separate macro with commands to be executed in parallel or sequential. This group of commands is ended by a '}'.

Finally, the definition of a macro is ended by a '}'.

13.1 a. The body of a macro

The body of a macro consists of:

- **subroutine calls:** subroutine-name arg1 ... argn
  The subroutine-name should be defined in 'commands.h'; this will be discussed afterwards. Furthermore the actual coding of this routine should be done in the file 'commands_lib.c' . The subroutines have only one character-string argument. This string contains all arguments listed in the 'command.list' file, arg1 ... argn, complete with embedded blanks.
- **macro-substitutions:** %macro-name arg
  The macro macro-name has to be defined elsewhere in the file 'command.list'. Any arguments should correspond to those expected by the macro.
- **Shell commands:** @command
  Using the "system()" routine, a command is executed. The return code of this call is not used.
- **IF-statements**
  IF (subroutine-call) [subroutine-call | macro-call] ELSE [subroutine-call | macro-call]
  If a subroutine returns the value '1', the first subroutine (or macro) call will be executed. If the subroutine returns 0 the statement following the ELSE is executed.
- nested **PARALLEL** and **SEQUENTIAL**-parts.
  Parts of a macro can (and will) be treated as a separate macro.

13.2 The `commands.h` file

The file 'commands.h', located in $TOPUNIDAQ/src/control/runcontrol, consists of a formal declaration of the subroutines that are called by the macros and a command table containing the pointers to subroutines and their names. This command table is needed by *runcontrol* to convert a string read out of 'command.list' to a subroutine-call.

All of the subroutines have to be declared as 'int name(char *s)' in order to fit in this scheme. The actual coding of the subroutines has to be done in the file commands_lib.c . These subroutines can call any standard routine or routine provided in the template library.

Entries in the command table have the following syntax:

```
"command-name", subroutine-pointer,
```

The command-name contains the name of the subroutine as used in the file 'commands.list' while the subroutine-pointer is the pointer to a subroutine coded in 'commands_lib.c', usually these entries appear as, *e.g.*,

```
"set_run_status", set_run_status,
```

In the next section we summarize the routines that are already coded in 'commands_lib.c' .

- **send_process** (referenced by "ask")
  The argument string consists of a process name to which a message has to be sent and the message itself. The subroutine sends the message to the process indicated.
- **send_logbook**
  The argument indicates which message needs to be sent to the logbook-process. Meaningful arguments are run_start, run_end, run_pause, and run_resume which mean that runcontrol is trying to modify the runstate, execute [cmd] which indicates that runcontrol executed a command, and incompatible [cmd] which indicates that runcontrol tried to execute a command which is incompatible with the runstate.
- **update_parameters**
  The arguments can be STORAGE or RUN. The subroutine sends the specified parameters to the *operator* process and waits until the person on shift modifies them or approves of them. The modifications are adopted by the *runcontrol* process.
- **write_begin_run**
  This subroutine writes a begin-run summary to a file. This file is read by the *logbook* process. The file name is coded in $TOPUNIDAQ/src/control/runcontrol/files.h as $TOPUNIDAQ/log/writerun.file .
- **write_end_run**
  This subroutine writes an end run summary to file. This file is read by the *logbook* process. The file name is coded in $TOPUNIDAQ/src/control/runcontrol/files.h as $TOPUNIDAQ/log/writerun.file .
- **check_run_status** <argument>
  The argument describes a run status. The actual run status is compared to the argument. Possible states, defined in file $TOPUNIDAQ/src/control/runcontrol/runco.h, are Idle, Initialize, Pause and Run, which correspond exactly to the values of END_STATE, BEGIN_STATE, PAUSE_STATE and RESUME_STATE.
- **macro_exit** (referenced by "exit")
  Stops executing the macro. If the macro was called from a parent macro, the parent macro is also stopped.
- **next_runnr**
  Adds one to the run number, unless run number is zero. If the run number is zero, it is not changed. The new run number is sent to both *collector* and *runcontrol* (remember, a child process of *runcontrol* is executing this command).
- **prompt_process**
  The arguments are a process name, and additional arguments that are sent to that process as arguments of a "prompt" command. *Runcontrol* expects a "re:prompt" command back (with parameters). The parameter string of this command is stored in the user variable "re_prompt", and can be used in following commands.

13.4 Additional Remarks

Remember that the commands listed in the commands.list file are executed by child processes. When the *runcontrol* user variables change value, the user variables of these child processes still have the old values. When communicating to other processes the most up to date values should be used.

Using the command.list file, variables can be sent to other processes as follows:

ask runco tell collector set runnr = &runnr

This command will send a message to *runcontrol*. *Runcontrol* will substitute the runnr token and send a string like "set runnr = 10001" to *collector*.

14. OPERATOR

The *operator* process is the interface between the user and the online system. When it starts four windows are presented on the user's workstation screen. These will look similar to those shown in figure 15.

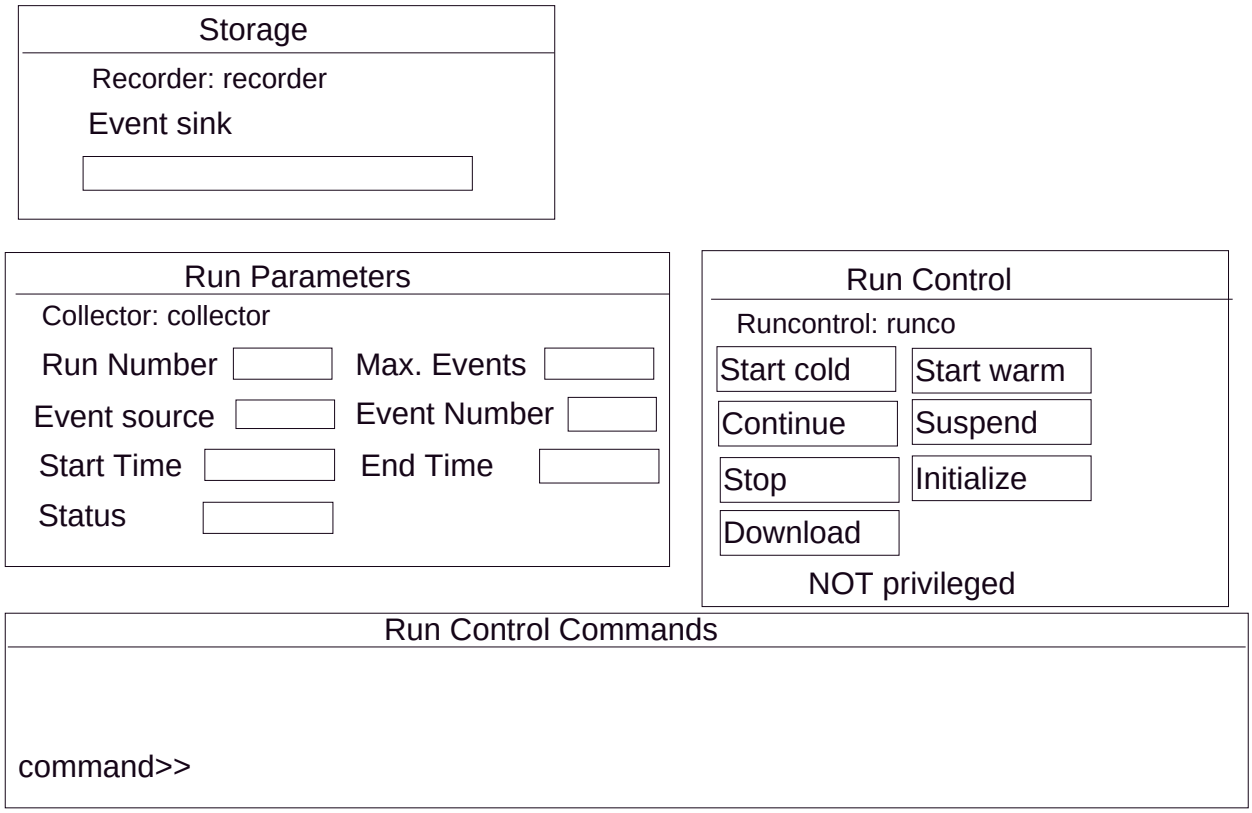

Figure 15. The Four Default *Operator* Process Windows.

14.1 Actions Taken to Pushed Buttons

The actions taken when an *operator* button is pushed depend on the input file of *runcontrol*, `command.list` . Below the actions of *runcontrol* to the commands listed in the User's Guide are summarized.

1. a) BEGIN_RUN_COLD
   First *runcontrol* executes INITIALIZE, followed by 'START_RUN'
   b) BEGIN_RUN_WARM
   First *runcontrol* updates the run number followed by 'START_RUN'.

   c) START_RUN
   First the eventsink, dest_dir, runnr, and maxnevents parameters are sent to *recorder*. The eventnr is set to zero for both *runcontrol* and *collector*. The eventsource, runnr and maxnevents are sent to the *collector*. Afterwards, the person on shift is prompted for a begin run comment. Both the begin run comment given by the person on shift and the runnr are sent as parameters of the BEGIN command to the *collector* process. Finally the RESUME command is sent to the *collector*.
2. STOP_RUN
   The command 'END' is sent to the *collector* and an end of run statement is sent to the *logbook*.
3. SUSPEND_RUN
   The command 'PAUSE' is sent to the *collector*.
4. CONTINUE_RUN
   The command 'RESUME' is sent to the *collector*.
5. DOWNLOAD
   To be implemented.
6. INITIALIZE
   Initialize the run parameters. This causes the run number to be raised by one. Furthermore, both run and storage parameters are sent to the person on shift for confirmation. Afterwards, the confirmed runnr, maxnevents, and eventsource are sent to the *collector*.

14.2 Reconfiguring Operator

Both the "run parameters" and the "run control" windows are completely configurable to the needs of an experiment. The standard *operator* configuration may serve as an example of how to do so.

14.2 a. The Run Parameters Window

The configuration of the run parameters window is determined by file $TOPUNIDAQ/src/control/operator/runpar.h . All system and user level variables of all processes within UNIDAQ can be displayed. For each variable that one wants to display, six fields need to be filled in a structure called `rp_but` which is coded in this file. Look at the following example entry from this file:

```
"Run number", "runnr", "collector", INT_DEC, (int(*)())NULL, YES,
```

The first field determines the text to be displayed in front of the parameter. The second field shows the name of the user or system level variable to be displayed. The third field shows the name of the process that maintains this variable. If this process name is "collector", "recorder", or "runcontrol", then this name will be translated according to the corresponding user level variable indicating the real process currently in use. The fourth field shows the type of variable. This can be "REAL", "STRING", or different types of integer representations. In this case, "INT_DEC" means a decimal integer representation. Other possibilities include "INT_HEX" indicating a hexadecimal representation and "INT_CONV" meaning there is a conversion routine coded for the variable (see below) with the name of the routine entered in the fifth field. The fifth field contains the name NULL otherwise. The sixth field shows if this parameter is to be user updated at the beginning of a run by the *operator* process (YES) or not updated (NO).

The conversion routine named in the fifth field is to be written in file `runp_conv.c` in the same directory. In general, such a routine looks like:

```
int routine ( val, txt, dir )
int *val, dir;
char *txt;
{
   if (dir == 0) {
      convert *val into txt.  Return 0/1 if success/not.
   }
   else {
      convert txt into *val.  Return 0/1 if success/not.
   }
}
```

14.2 b. The Run Control Window

File $TOPUNIDAQ/src/control/operator/runcon.h determines the setup of the run control window. One needs to fill the "rc_but[ ]" array. Each button requires two fields to be filled. The first field shows the text to be displayed on the button, and the second field contains a routine name to be executed when the button is clicked, *e.g.*,

```
"Start cold", (int(*)())act_cold ,
```

The named routine should be coded in file `runcon_act.c` in the same directory and may contain all kinds of actions. Examples are already coded in this file.

15. LOGBOOK

Three subroutines are provided to interface user processes to the *logbook*.

long add_err_string (char *s);
long send_err_string (unsigned long p1);
long send_err_mess (unsigned long p1, char *s);

These routines access *logbook* via the RoundUp command.

The idea is that strings of characters (ie, a message) can be sent for logging with up to 4 such strings sent at a time. Routine add_err_string has one argument, a message which is to be sent to the logbook. Zero to four such calls may be made, to be followed by a call to send_err_string which actually dispatches the messages to the *logbook* process. The argument to send_err_string is either one of the pre-defined message codes, or zero in the case where messages are simply to be logged. A call to send_err_mess combines one call to add_err_string with a call to send_err string. Pre-defined error codes are found in files in the directory $TOPUNIDAQ/include and the specific file name depends on the language of your program. The files are called sdc_pack_msg_[F.inc|c.h|cxx.h] for Fortran, C or C++ source code.

Four of these pre-defined error codes are for generalized error reporting and have no specific process name attached as a prefix. Any process can use these if the programmer is so inclined without the need of going through a murmur reconfiguration procedure (see below). A single message string describing the error should be sent with the error code, preferably containing the process name to help in identifying the origin of the message. These pre-defined error codes are "General_Success", "General_Info", "General_Warning", and "General_Error". A sample code segment in C using one of these error codes might contain the lines

```
#include "sdc_pack_msg_c.h"
  j = send_err_mess ( General_Warning, "Program catchall – Watch out!");
```

The "read" and "log" user-level commands make the *logbook* process accessible to the command shell via the `ask` tool. Using "log" a single message string will be logged both to the murmur "General Log" window and to the log.bookYYMMDD file. The "read" command copies the specified file into the log.bookYYMMDD file, but the content does not appear in the murmur window.

The location of the log.bookYYMMDD file depends on the initialization of the system as described in the Installation Guide. By default it is placed in $TOPUNIDAQ/log.

15.1 Error Codes

Murmur actually allows parameters of type Floating, Integer, and others in addition to Strings as arguments of its calls. However, we have implemented a restricted set which allows only character string arguments. A multi-step process must be followed if new error codes are to be added to those already known to UNIDAQ. The complete process is illustrated in the UNIDAQ Installation Guide. Basically, a message prefix reflecting the name of the calling program is chosen, a message mnemonic giving more hints as to what the message means to the calling program is chosen, and up to four strings further defining the message text can be used. This is formalized in the file $TOPUNIDAQ/murmur_client/error/inc/sdc_pack.msg and then the steps detailed in the installation guide are followed. *Logbook* will understand the new messages immediately, but the murmur server will not know of them until the next time it is started; instead it will simply route them to its "General Log" window, possibly without the message strings, independent of the message severity.

15.2 Compiling to Use Murmur

The *logbook* process can be configured to use murmur; the default is to not use it at all. To do this, edit the Makefile in $TOPUNIDAQ/src/control/logbook and remove from the CXXFLAGS switch the string " -DNO_MURMUR". Then type the commands "make clean" and "make" to recompile the image. In this case, the *logbook* will assume that murmur is now to be used and will begin to send all messages to the murmur server in addition to logging all messages to file.

16. XPC: Checking the Process Status

The XPC checks the status of all local processes. If there is a change in this status, the modification will be sent to xpc's running on other nodes. The XPC checks the process status every MSGTIME (60) seconds. The status is checked by sending a busy-signal to the process. The process reacts by updating the "proc_status" variable in the Status Path. If this updating took place, the XPC sends a message to the process to set its proc_status to "ALIVE". When the proc_status variable becomes "ALIVE" then the Common Table entry is also set to "ALIVE". If later signals are sent before the proc_status has been set to "ALIVE", then the Common Table entry for the process will be set to "BUSY". If a process does not respond to signalling, it is asumed to be dead. Its message queues will be removed, and the Status Path and Common Table will be updated by the XPC to show the process is "DEAD". Critical Variables of the process will be written to a disk file in $TOPUNIDAQ/log/var for later invocations of the process.

17. XPC CHECKER

When the *xpc_checker* has been started using the XPC by using the shell command:

ask xpc START YES xpc_checker

the *xpc_checker* will be restarted by the local XPC if it happens to die. If the *xpc_checker* is started by hand, you have to restart it yourself. If the node on which it is running crashes, you can restart the *xpc_checker* on any of the other nodes. It will not restart automatically.

18. MURMUR

18.1 The Murmur Server

The murmur server comes pre-built by Fermilab. It will run only on SUN Sparc and SGI platforms. Currently only the SUN version is distributed with UNIDAQ.

18.2 The Murmur Client

The *logbook* process is the only murmur client process in the UNIDAQ system. For this process to correctly access the murmur server, the environmental variable MURMUR_SERVER_ADD must be set. This address is established during the installation phase of UNIDAQ. By default *logbook* is built without calls to the murmur server, but this can be changed as explained in the chapter on *logbook*.

18.3 Starting the Murmur Server

The IP address of the machine where the murmur server runs is the value which is assigned to the environmental variable MURMUR_SERVER_ADD. This address is requested during the UNIDAQ installation phase and must be known by the *logbook* process for it to correctly access the murmur server.

Access to the command `murmur_Start` requires that the $TOPUNIDAQ/setup.csh file has been sourced.

18.4 Stopping the Murmur Server

Access to the command `murmur_Stop` requires that the $TOPUNIDAQ/setup.csh file has been sourced. When the murmur server runs it uses a number of semaphores and some shared memory. As a result, if other UNIDAQ processes are running on the same node as the murmur server, and either Reset or Reset_local is issued on the node, then you must be sure to shut down the murmur server as well or the node reset procedure will not be able to completely clean up the semaphores and shared memories, and it will loop forever. The correct order then is:

murmur_Stop
Reset_local or Reset or Reset_node-name

18.5 Further Murmur Restrictions

If the murmur server stops, but *logbook* is still running as a murmur client, *logbook* will go through a standard IP timeout of several minutes before disconnecting from the server and continuing without it. To reconnect, *logbook* must be stopped and then restarted only after the server has been restarted.

The three murmur display windows do not have to appear on the same workstation, but can be split to several machines. See the Installation Guide for details.

If a murmur display window is accidentally closed or disappears for any other reason, the only way to get it back is to stop and restart the murmur server.

Do not forget, the machines on which murmur display windows appear must have "xhost" access set for the machine where the server runs. For example, if the murmur server runs on `pablo`, with display to `mhpbob`, then `mhpbob` must allow X-displays from `pablo`. The command

xhost +pablo

issued to the shell on `mhpbob` will accomplish this.

19. RETURN CODES OF SOME UNIDAQ LIBRARY CALLS

The UNIDAQ libraries have been affected by the transition to the distributed environment. The following return codes have been modified by creating the distributed environment:

int Template_Init(char *process_name)

-10: Cannot decode $TOPUNIDAQ
-2: Process of the same name runs remotely
-1: Process of the same name already runs locally
0: Successful

Note: Only if the return is successful will the entry be added to the Common Table, thereby allowing other processes to communicate with this process.

int Send_Message(char *process_name, char *message)

-12: Process does not exist within distributed UNIDAQ
-11: Cannot open $TOPUNIDAQ/conf/nodefile
-10: Cannot decode $TOPUNIDAQ
-5: Cannot link to remote server
-4: Remote server does not respond
-3: Process has no entry in the common table
-2: Nodename in table does not correspond to local nodename

-1: Process is dead
0: Successful

long add_err_string(char *err_mess)
0: Successful return
1: Error, message could not be packed for sending to *logbook*

long send_err_string(unsigned long err_code)
0: Successful return
1: Error, pre-packed messages could not be sent to *logbook*

long send_err_mess(unsigned long err_code, char *err_mess)
0: Successful return
1: Error, message could not be packed for sending to *logbook*

int link_to_sp()
-2: Cannot link to shared memory
-1: Cannot link to semaphore
0: Succesful

int add_proc_to_sp(int size)
-5: requested size too big for status path
-4: Process is already listed in status path
-3: No space in current path, cannot link to new path
-2: Should never occur
-1: Status Path is not linked
0: Succesful

int add_variable_to_sp(char *varname, short type, short length, char *value)
-4: Cannot add another buffersegment to process
-3: No space left to add this variable
-2: Variable is already in status path
-1: Status Path is not linked
0: Succesful

int set_value_in_sp(char *varname, char *value)
-3: There is not enough space available in datasegment
-2: Variable is not known
-1: Status Path is not linked
0: Succesful

int read_value_from_sp(char *procname, char *varname, short *type, char *value)
-12: Process does not exist within distributed UNIDAQ
-11: Cannot open $TOPUNIDAQ/conf/nodefile
-10: Cannot decode $TOPUNIDAQ
-5: Cannot link to remote server
-4: Remote server does not respond
-3: Process does not exist
-2: variable does not exist

-1: Status Path is not linked
0: Succesful

int set_all_values_in_sp()

-1: Status Path is not linked
0: Succesful

int read_all_values_from_sp(char *procname, char *varname, int varsize, short *type, char *value, int valsize, int *n_var)

-12: Process does not exist within distributed UNIDAQ
-11: Cannot open $TOPUNIDAQ/conf/nodefile
-10: Cannot decode $TOPUNIDAQ
-5: Cannot link to remote server
-4: Remote server does not respond
-2: Process does not exist
-1: Status Path is not linked
0: Succesful

int remove_proc_from_sp(char *procname)

-3: Cannot remove process
-2: Process is not listed in status path
-1: Status Path is not linked
0: Succesful

int unlink_sp()

0: Succesful

int Crit_Var(char *variable_name)

-1: Variable not listed in status path
0: Succesful

CONTENTS

1. INTRODUCTION 2

   1.1 Suggested Reading 4

   1.2 Acknowledgements 5

2. FILES REQUIRED FOR DISTRIBUTED UNIDAQ 5

   2.1 The `.netrc` File 6

3. STARTING DISTRIBUTED UNIDAQ 7

   3.1 Starting Processes with Alternate Names 7

   3.2 Receiving Event Buffers Across the Network 8

   3.3 Changing the Size of NOVA Buffers 8

4. STOPPING DISTRIBUTED UNIDAQ 8

5. RUNNING DISTRIBUTED UNIDAQ IN A SINGLE CPU ENVIRONMENT 9

6. THE MESSAGE TEMPLATE 9

   6.1 System-Level commands and System-level Variables 9

      6.1 a. System-Level Commands 10

      6.1 b. System-Level Variables 11

   6.2 User-level Commands and User-Level Variables 11

      6.2 a. SETting Variables While Command Parsing 12

   6.3 Token Replacement in Commands 12

   6.4 Handling Command Messages and Event Data: Main Loop of the Template 12

   6.5 Message Passing Within a Single Processor 14

   6.6 Message Passing Across a Network 14

      6.6 a. Case a 14

      6.6 b. Case b 14

   6.7 Updating the common table 15

   6.8 Locations of Some Files Related to Message Passing 16

7. STATUS PATH 16

7.1 Reading from and Storing Variables to the Status Path 16
7.2 Critical Variables 18
8. EVENT BUFFERS 18
8.1 Number and Size of Event Buffers 19
9. SIGNALS, SEMAPHORES AND OTHER SYSTEM RESOURCES 20
9.1 Signals 20
9.2 Message Queues 20
9.3 Shared Memory 21
9.4 The RPC System 21
9.5 Semaphore Usage 21
10. MODIFYING THE COLLECTOR PROCESS 22
10.1 Standard Subroutines Callable from Collector 26
11. ANALYZER 27
11.1 Analyzer on the SUN, SGI and HP 27
11.2 Analyzer on the DECstation 28
11.3 Never the Twain Shall Meet 28
12. DATAVIEW PROCESS 29
13. RUN CONTROL PROCESS 29
13.1 The `command.list` file 30
13.1 a. The body of a macro 30
13.2 The `commands.h` file 31
13.3 The commands/subroutines already present in `commands_lib.C` 32
13.4 Additional Remarks 33
14. OPERATOR 33
14.1 Actions Taken to Pushed Buttons 34
14.2 Reconfiguring Operator 34

14.2 a. The Run Parameters Window 34
14.2 b. The Run Control Window 35
15. LOGBOOK 36
15.1 Error Codes 37
15.2 Compiling to Use Murmur 37
16. XPC: Checking the Process Status 37
17. XPC CHECKER 38
18. MURMUR 38
18.1 The Murmur Server 38
18.2 The Murmur Client 38
18.3 Starting the Murmur Server 38
18.4 Stopping the Murmur Server 38
18.5 Further Murmur Restrictions 39
19. RETURN CODES OF SOME UNIDAQ LIBRARY CALLS 39