PS/CO/Note 92-21
25.09.1992

CONTROL SYSTEM SOFTWARE FOR THE VACUUM SYSTEM

Date : July 1992

Author : Anker Rosenstedt

1 Contents

2 Control software for the vacuum system

2.1 Specifications

2.1.1 Types of equipment and Equipment Modules

The different types of equipment that it is necessary to control explicitly from the application programs are listed below. Some equipment is indirectly controlled within a pumping group/station, and are not directly controllable.

To make the 'image' seen from the control room more clear, all the vacuum equipment is divided in three logical groups, which contains : pumps, gauges and valves. There is an Equipment Module for each group of equipment, meaning that all pumps are controlled from the *vpump* Equipment Module etc. In this way, the very identical (both name and function) property functions for the three different groups of equipment are separated. There is absolutely no technical reason for making this separation, it's entirely a matter of logic and convenience as seen from the operator.

Equipment Module	Type of equipment	Abbreviation
VPUMP :	PUMPING GROUPS	(P_GROUP)
	ION PUMPS	(P_ION)
	SUBLIMATION PUMPS	(P_SUBL)
	PUMPING STATIONS	(P_STAT)
VGAUG :	PIRANI/PENNING GAUGES	(G_PIRAN)
	ION GAUGES	(G_ION)
VVALV :	VALVES	(V_VALVE)

There has previously in the definition process been a suggestion of having a *sector* Equipment Module. This, however, due to problems of how exactly to define a sector and questions about if it's at all necessary, is therefore not part of the present specification.

2.1.2 The datatable

Although the datatable is logically located inside the Equipment Modules, it has some specifications connected with it which are not only implementation dependent. Apart from the control and acquisition data, the DT also contains some read only information for each single piece of equipment. These columns must be initialized by the operator for each new peace of equipment that is defined. The list of the columns that the DT contains is general for all the vacuum Equipment Modules :

1. Control data containing the up to date settings.
2. Acquisition data containing last received acquisition values.
3. Treatment (TRM), which is a *read only* int[2] containing the Equipment Type :
   1. TRM[0] : Equipment Type.
   2. TRM[1] : Equipment Subtype.
4. Serial Number (SNE) or Physical Equipment Number, which is a *read only* integer.

The exact structure of the first two items on the list is implementation dependent.

The Treatment column is an array of two integers which holds the Equipment Type and the Equipment Subtype. The Equipment Type numbers are defined as follows :

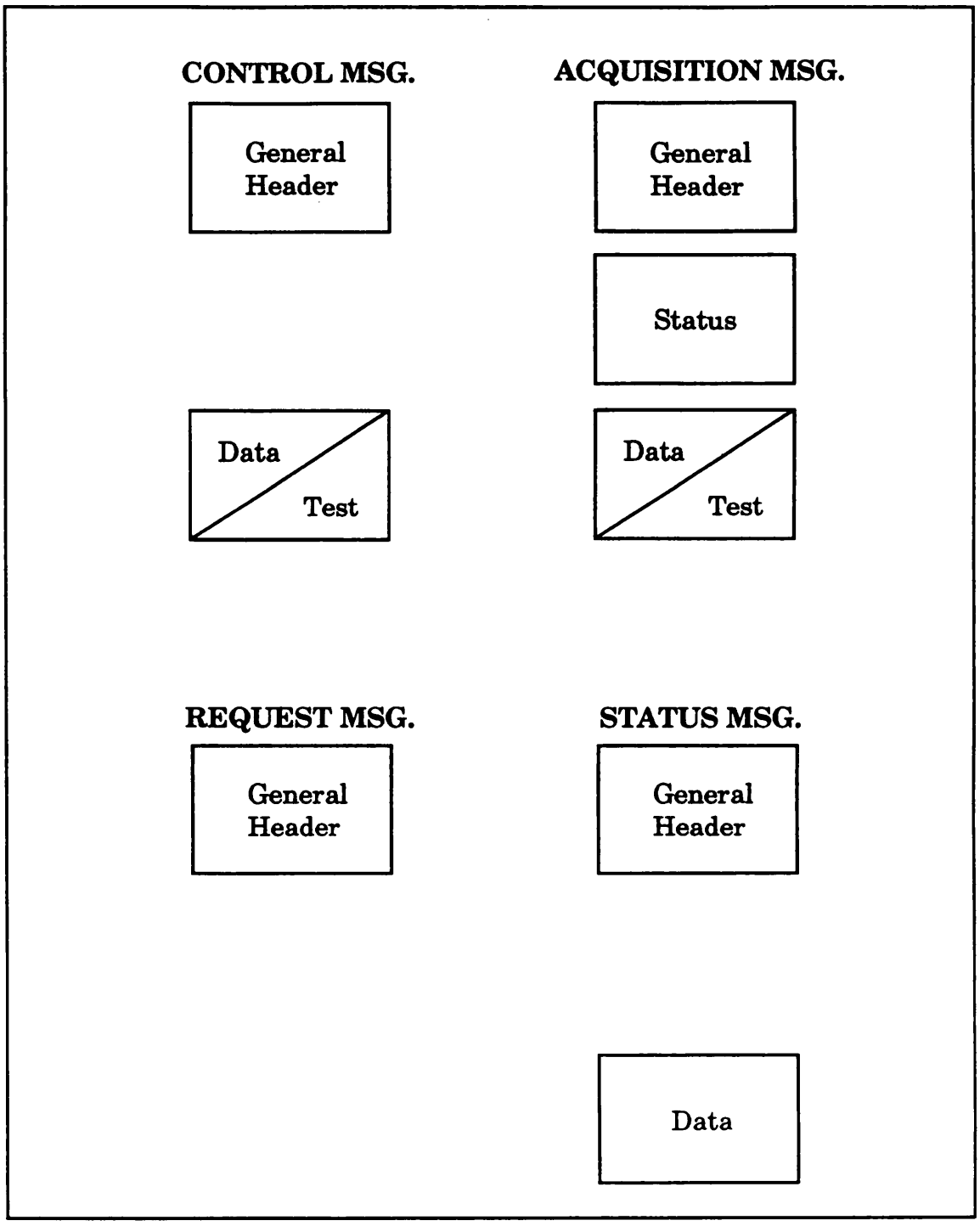
P_GROUP	1	G_PIRAN	5
P_ION	2	G_ION	6
P_SUBL	3	V_VALVE	7
P_STAT	4		

The Equipment Subtype is used to identify f. ex. different manufacturers of the same type of equipment. The Equipment Module is not dependent on this number.

The Serial Number or Physical Equipment Number is a single integer which identifies the vacuum equipment with local numbers. The Equipment Module is not dependent on this number either.

2.1.3 The messages

All messages are made up of a few general parts, which should hopefully also become general in use for other types of equipment. For vacuum there are four different messages, which are composed as shown below :



The *general header* block contains information on which number (equipment number) and type of equipment is to be accessed, what the rest of the message contains, the time the message was sent, and also a field for specialist action. All messages have a *general header* as the first part, and since it is from here that the type of message is determined (the meaning of the message is contained in the message itself), the *general header* must be identical for all messages. That is, the fields must be in the same order, have the same length etc., so that the receiver of the message is always able to read it, no matter what the rest of the message contains. The *general header* should be so general that, in the future, it can be used for all types of messages controlling all types of equipment.

The *status* block is simply some general status information regarding the equipment that is controlled. The format/meaning of the status information is specific to the vacuum equipment, and not necessarily general for several types of equipment.

The *data/test* block holds either the data that is to be send/received, or the space can be used for test. When used for test there is no defined meaning of the contents, it's entirely up to the specific process. The *test* part is only used by the special test property functions. The following pages will therefore only concern the messages with the *data/test* block used for data.

To simplify the code and make it easier to read (more general), the same *normal* control and acquisition message are used for *all* the different Equipment Types and all the Equipment Modules. I.e. the messages contain exactly the same fields no matter what type of equipment they are used for. From a programmers point of view this is a major shortcut from the alternative : Having a separate message structure for each of the equipment types. This would result in an enormous amount of code just for declaring the message structures (20 pages or so), and the program code gets comparable complicated and tedious to read because everything is a special case, and nothing is in any way general.

The drawback is of course that for some of the types of equipment, the messages will contain some information not needed, or certain fields in the message will be undefined. This could eventually be a little confusing and not so ideal. However, it is such a great advantage only having to deal with one type of message that this has been decided.

On the following pages is an outline of all the messages used for vacuum, with the fields they contain, the type of the field and in some cases an allowed value range. These are the specifications agreed on with the vacuum group. A short description of the meaning and use of the different fields in the messages follows in the next chapter.

The following types are used in the messages :

byte : 8 bit unsigned integer.

byte_s : 8 bit signed integer.

int16 : 16 bit unsigned integer.

int32 : 32 bit unsigned integer.

real : Float.

time : A structure containing time in seconds since bootup (1970) and usec.

event : A structure containing puls id. Not in use for vacuum since no PPM.

Because the *ccv* and *aqn* fields have different types depending on the type of equipment that is controlled, they are given a union type (*mul_type*), i.e. a type which at any time can hold any one of the above types : *byte, int16, int32* or *real*. This makes coding very simple and efficient.

The *normal* CONTROL and ACQUISITION messages :

	CONTROL MESSAGE	
	NAME	TYPE
HEADER	lin1	int16
	lin2	int16
	lin3	int16
	amm	int16
	pulsid	event
	date	time
	specialist	int16
DATA	ccsact_chng	byte_s
	ccsact	byte
	ccv_chng	byte_s
	ccv	mul_type
	ccv1_chng	byte_s
	ccv1	real

	ACQUISITION MESSAGE	
	NAME	TYPE
HEADER	lin1	int16
	lin2	int16
	lin3	int16
	amm	int16
	pulsid	event
	date	time
	specialist	int16
STAT	phys_status	byte
	saqn	byte
	aspect	byte
	qualif	byte
	busy_time	int32
DATA	aqn	mul_type
	aqn1	real
	aqn2	real

There is no need to show the **REQUEST control message**, as it is just the header part of the normal control message.

The special **STATUS acquisition message** exists for warning and fault reporting. First an *exception* is defined as (structure in C) :

exception	
list	int32
date_last	time
date_imp	time

list contains a bit for each warning/rfault/ufault/interl.,

date_last is the time of the last warn/rfault/... , and

date_imp is the time of the most important warn/rfault/...

The STATUS acquisition message now looks as below :

	STATUS MESSAGE	
	NAME	TYPE
HEADER	lin1	int16
	lin2	int16
	lin3	int16
	amm	int16
	pulsid	event
	date	time
	specialist	int16
DATA	warnings	exception
	rfaults	exception
	ufaults	exception
	interlocks	exception

Valid values of the different fields depending on equipment :

Equipment Field	P_GROUP	P_ION	P_SUBL	P_STAT	G_PIRAN	G_ION	V_VALVE
CONTROL MESSAGE :							
lin1-3	-	-	-	-	-	-	-
amm	0..5	0..5	0..5	0..5	0..5	0..5	0..5
pulsid	not used	not used	not used	not used	not used	not used	not used
date	-	-	-	-	-	-	-
specialist	-	-	-	-	-	-	-
ccsact_ chng	<0, 0, >0	<0, 0, >0	<0, 0, >0	<0, 0, >0	<0, 0, >0	<0, 0, >0	<0, 0, >0
ccsact	1, 2	1, 2	1, 2	1, 2, 3	1, 2	1, 2, 3	1, 2
ccv_ chng	not used	<0, 0, >0	<0, 0, >0	not used	not used	<0, 0, >0	not used
ccv	not used	byte : 1, 2	int32	not used	not used	int32	not used
ccv1_ chng	not used	<0, 0, >0	<0, 0, >0	not used	not used	<0, 0, >0	not used
ccv1	not used	-	-	not used	not used	-	not used
ACQUISITION MESSAGE :							
lin1-3	-	-	-	-	-	-	-
amm	0..5	0..5	0..5	0..5	0..5	0..5	0..5
pulsid	not used	not used	not used	not used	not used	not used	not used
date	-	-	-	-	-	-	-
specialist	-	-	-	-	-	-	-
phys_ status	1..4	1..4	1..4	1..4	1..4	1..4	1..4
saqn	1..5	1, 2, 3	1, 2, 3	1..6	1, 2	1, 2, 3	1..5
aspect	bit 1..3	bit 1..3	bit 1..3	bit 1..3	bit 1..3	bit 1..3	bit 1..3
qualif	bit 1..5	bit 1..5	bit 1..5	bit 1..5	bit 1..5	bit 1..5	bit 1..5
busy_ time	-	-	-	-	-	-	-
aqn	not used	byte : 1, 2	int32	real	real	int32	not used
aqn1	not used	-	-	not used	not used	-	not used
aqn2	not used	-	-	not used	not used	-	not used

Meaning of symbols in the table :

- : No restrictions on value (type permitting), for meaning see below.

not used : The field is not in use, meaning it contains an *undefined* value.

For the *ccv* and *aqn* fields, where only the type is stated, the value can be anyone the type permits (or the specific process itself must check the *ccv* value).

2.1.4 Meaning of message fields

CONTROL MESSAGE :

lin1, lin2 and lin3 :

lin1 contains the Equipment Module Number.

lin2 is composed of two numbers : The upper 8 bits are the Equipment Type, and the lower 8 bits are the Equipment Subtype, both taken directly from the datatable (TRM) and put into the Control message before it is send.

lin3 is the Serial Number (or Physical Equipment Number), read from the corresponding (SNE) datatable column.

amm :

Acquisition Message Meaning, in the control message it means the following :

0 (RET_ACQ) : Ask specific process to return a normal acquisition message for the equipment number given in *lin1* and *lin2*.

1 (RET_CTRL) : Specific process must return the most recently received control message.

2 (RET_ACQ_TEST): Return acquisition message for test.

3 (RET_CYCLE) : Return Control message (cycle no). Used for vacuum ?

4 (RET_STAT) : Return status acquisition message.

5 (NO_RET) : Do not return any message (asynchronous).

Note that the meaning of *amm* is extended rather than completely changed from previous definitions.

pulsid :

Puls identifier. Not used for vacuum because no PPM operation.

date :

The time when the message is send.

specialist :

Specialist action. The Equipment Modules are not dependent on the defined values and the corresponding actions taken in the specific process.

ccsact_chng, ccv_chng and ccv1_chng :

Flags if ccsact/ccv/ccv1 has been changed or is valid/unvalid :

< 0 (NO_CHANGE) : Valid value, no change.

> 0 (CHANGED) : Changed.

0 (NOT_VALID) : Not valid value.

The _chng fields in the DT can only be changed by the ccsact/ccv/ccv1 property to CHANGED or NO_CHANGE .

Since the only property function that (currently, see next chapter) sends a *normal* control message is the *ccsact*, the *ccsact_chng* field will always be CHANGED.

ccsact :

The Current Control Set ACTuation value, the parameter that actually controls the equipment (on/off/...).

ccv and ccv1 :

Current Control Values.

ACQUISITION MESSAGE :

lin1, lin2 and lin3 :

lin1, lin2 and *lin3* in the acquisition message are just copied from the corresponding control message.

amm :

In the acquisition message, the *amm* field is just a copy of the corresponding field in the control message received.

pulsid :

Not used for vacuum.

date :

The time when the ACQUISITION VALUES WERE MEASURED.

specialist :

See Control message.

phys_status :

The physical status of the equipment. The Equipment Modules don't depend or test the values defined below :

1 : Operational.

2 : Partly operational.

3 : Not operational.

4 : Needs commissioning.

saqn :

Read back actuation value (running/stopped/runup/...). This value is not checked for size or anything else by any of the Equipment Modules.

aspect :

The single bits have the meanings listed below. The Equipment Modules don't depend on these values.

bit 1 : Not connected.

bit 2 : Local.

bit 3 : Remote.

qualif :

Status qualifier, the single bits have the following meaning :

bit 1 : Warning.

bit 2 : Busy.

bit 3 : Resetable fault.

bit 4 : Unresetable fault.

bit 5 : Interlock.

If *qualif* contains a 0, everything is OK. As it appears, the most important condition is assigned the most significant bit.

busy_time :

The time in seconds in which the specific process will be busy

aqn, aqn1 and aqn2 :

Acquisition values.

STATUS MESSAGE :

lin1, lin2, lin3, amm, pulsid, specialist :

The same as for a normal acquisition message.

date :

The time when the message was sent.

warnings.list :

A list of warnings : Each bit corresponds to a specific warning. The specific meaning is of no concern to the Equipment Modules.

warnings.date_last :

Time of the last warning.

warnings.date_imp :

Time of the most important warning.

The above meaning of the *warnings* fields apply in the same way to *rfaults*, *ufaults* and *interlocks*.

2.1.5 Property functions

Except for single data types and a few value ranges, most of the properties in the three different Equipment Modules for vacuum are general in behavior. This general behavior is described below in a few coarse steps :

ccsact (write, 1 EqmInt) :

- Check that the equipment accessed belongs to the Equipment Module.
- Check that the parameter passed from the application program is within allowed range. This check should be based on the Equipment Type read from the DT (TRM).
- Put the parameter in the *ccsact* field in the control msg. and set *ccsact_chng*.
- Get TRM and SNE from the DT (for *lin2/3*) and compose the rest of the control message. Set *amm* = NO_RET.
- Store *all* the control values that have been changed in the DT.
- Send the *normal* control message. The property will not wait to receive an acquisition message.
- After the message is send, update the *_chng* fields in the DT.
- Return. If anything has gone wrong, a coco error code is returned.

ccv, ccv1 (write, 1 EqmFloat) :

- Check that the equipment accessed belongs to the Equipment Module.
- Check that the type of equipment accessed has a *ccv/ccv1* parameter. If not, return coco error.
- Check that the parameter passed is within allowed range (*ccv* only).
- Store the parameter (control value) at the appropriate (*ccv/ccv1* field) place in the DT.
- Set corresponding *_chng* field in the control message in the DT.
- Note that no message is send.

staq (read, 1 EqmInt), **aqn, aqn1, aqn2** (read, 1 EqmFloat) :

- Check that the equipment accessed belongs to the Equipment Module.
- Set *amm* = RET_ACQ. Compose the rest of the *request* message.
- Send the *request* control message and wait to receive a *normal* acquisition message.
- Upon receipt, *all* the fields in the acquisition message is put into the DT, and only the value asked for (*saqn, aqn,...*) is passed back to the calling program.
- If anything goes wrong with the system (most important errors), a return is made with the coco parameter set to appropriate value.
- If the system is working correctly, the *qualif* field in the received acquisition message is examined, and a corresponding coco error is returned for the most important equipment error. In this way, error conditions on the equipment will only be communicated if the control system is functioning correctly.

A group of properties gets the warning, rfault, ufault or interlock status from the specific process :

warn, rfault, ufault, intlk (read, 1 EqmInt) :

- Check that the equipment accessed belongs to the Equipment Module.
- Get TRM and SNE from DT and set *lin2/3*. Set *amm* = RET_STAT.
- Send the *request* control message and wait to receive a *status* acquisition message.
- Upon receipt, *no* new values are stored in the DT.
- The *warnings.list, rfaults.list, ufaults.list* or *interlocks.list* respectively (depending on which property) is returned to the calling progr.
- Appropriate coco is returned.

warnm, rfaulm, ufaulm, intlkm (read, EqmInt[4]) :

- Exactly the same as above, except that the *warnings.date_last* and *warnings.date_imp* is returned together in an array. The same with the dates for *rfaults, ufaults* and *interlocks*. The exact format is :
  int[1] : *xxxx.date_last* seconds
  int[2] : *xxxx.date_last* microseconds
  int[3] : *xxxx.date_imp* seconds
  int[4] : *xxxx.date_imp* microseconds

A range of rather simple properties are exactly identical in all the Equipment Modules. All they do is to return certain fields in the DT which have been received from the specific process. Note that if no acquisition message has ever been received for a given equipment number, the value returned by these properties will be the initialization value, that is, not really valid.

phstat (read, 1 EqmInt) :

- Read the *phys_status* acquisition value from the DT, and return it to the calling program.

aspect (read, 1 EqmInt) :

- Read the *aspect* acquisition value from the DT, and return it to the calling program.

date (read, EqmInt[2]) :

- Read the *date* of the last received acquisition message from the DT, and return it to the calling progr. The format is :
  int[1] contains seconds and
  int[2] microseconds.

busy (read, 1 EqmInt) :

- Return the *busy_time* acquisition value from the DT.

For test of the specific process, two special properties exists :

tbit (read/write, 1 EqmInt) :

- If the property is invoked as a *read* function, the *specialist* field from the acquisition message last received is returned.
- If the property is invoked as a *write* function, the parameter passed is written to the *specialist* (control) field in the DT. No message is send, but the *specialist* value will be copied into all subsequent control messages that are send.

test1 (read, EqmInt[40]) :

- Check that the equipment accessed belongs to the Equipment Module.
- Compose *request* message with *amm* = RET_TEST. Nothing is written to the DT.
- Send message and wait to receive acquisition message with test field instead of normal acquisition data.
- Nothing from the received acquisition message is stored in the DT.
- The test field from the received message is returned in array to the calling program.

test1 (write, EqmInt[40]) :

- Check that the equipment accessed belongs to the Equipment Module.
- Check the range of the values in the array passed from the calling program.
- Compose control message (general header), with *amm* = NO_RET (!), nothing is written to the DT.
- Store the array from the calling program in the message instead of the normal control parameters, and send the message.
- Note that it is not possible for the specific process to distinguish between a normal control message and a control message with test data, because *amm* for some reason must not be used to indicate this (a new *amm* value could be invented, as for the acquisition : RET_TEST). Now, the only way to indicate it's a test control message is to use the *specialist* property/field, but it's the specific process programmer's problem.

2.2 Implementation

2.2.1 The message system

The message system is the part of the Equipment Module that via the message queues takes care of the communication with the specific process. Similarly, there is a message system in the specific process, making it able to communicate with the Equipment Modules. The code for the message system is described below.

Controlling the message queues, and sending and receiving messages of course makes use of a range of operating system calls. To avoid having to deal with the business of the operating system calls when writing the 'real' code, i.e. the property functions and the specific process, an interface has been developed which makes the access to the message queues very straightforward. This interface to the message system is in the *mqlib.c* file, listed in appendix (chapter 3.1.1).

The developed message system interface is very general. The *mqlib.c* file can be used in both the property functions and the specific process, and indeed every other application that uses messages. It can be used in any system, with any number of queues, and thus the *mqlib.c* code contains nothing at all that is specific to vacuum or to the Equipment Module control system.

The message system interface provides the user with only four functions, which are all that is necessary to easily communicate via messages : *setup_recv*, *setup_send*, *msg_recv* and *msg_send*. Furthermore, the user is provided with two kinds of structures, one that holds all parameters necessary when receiving messages (*setting_recv*) and a similar for sending messages (*setting_send*). The meaning of the fields in the structures is commented in the code. The functions are used as follows :

setup_recv : Opens the specified queue for receiving, and initializes the specified receiving attributes (*setting_recv*) with appropriate default values. Also sets up two signal handlers, see *msg_recv* for description. This function must be called before receiving any message.

setup_send : Opens the specified queue for sending, and initializes the specified sending attributes (*setting_send*) with appropriate default values. This function must be called before sending any message.

msg_recv : Waits, according to the attributes (*setting*) to receive a message on the specified queue. When the message is received without problems, the data will be stored in memory at the place of the *data_p*.

msg_send : Sends a message on the specified queue, and according to the attributes (*setting*), containing the data pointed to by *data_p*.

Before using the send and receive functions above, some message queues must of course first be created. This is done via the *queue.c* program listed in appendix (chapter 3.1.2), which creates two queues.

The ***msg_recv* function** has shown to be the far most complicated and problematic of the four, not the least because the *mqreceive* system call can be used in a great variety of ways.

The flags (*msgcb.flags*) for the *mqreceive* system call decides some of the characteristics of the receiving procedure. In the present implementation, only the MSG_TRUNC flag is set, resulting in the following : The *mqreceive* call will wait forever for a message to be received, the only thing able to interrupt it being an ALRM signal (see later) or one of the KILL signals. It also means that, if the message received is too large for the receive buffer (specified in *setting_recv.size*), the message will be truncated.

More important however, is that the value in *msgcb.msg_data* decides how the message is actually copied from the message queue. There are two different possibilities :

Either the message received is put in a system memory buffer, and the *mqreceive* call returns a pointer to that buffer. The message can then be copied 'manually' to user data space. When finished, the buffer must be released via another system call. If the whole message is to be used, this approach will result in the message being copied twice (when receiving, but it is of course also copied when sent).

Another way is to let the message be copied into user data space automatically. This is how it works in the present implementation, thus avoiding to copy the message 'manually'. For further details one should refer to the reference manual regarding the *mqreceive* system call, which is however hardly understandable.

The operating system sends a message type along with all messages, the type being a positive number. In the message system interface (*mqlib.c*), the message type to be send along with the message is put in the *type* field of the *setting_send* structure. On the receiving side, the *type* field in the *setting_recv* structure can be used to control which types to receive :

Zero : The oldest message on the queue will be received, no matter the type.

A positive number : The first message with that number will be received.

A negative number : Indicates that the oldest message having the lowest type less than or equal to the absolute value of the negative number is to be received.

Note that, as default, the type to be send and received is set to the current process ID (PID). If this is not appropriate in the application, the fields can of course easily be given new values after the call to *setup_recv* / *setup_send*.

There has been a great many problems with the *mqreceive* system call, due to the fact that there are still some bugs in the LYNX-OS (ver. 2.0.0) operating system. Since the mqreceive system call can be used in a great many ways, and because many of the errors often first show up when a number of messages have been send, a lot of time have been used to find a way to avoid these problems and develop a functioning message system anyway. The programs (*prop.c* and *mqrec_test.c*) used for testing the *mqreceive* system call can be found in the */u/rosenst/dsc/vac* directory, and are not listed in this report.

Because of the use in the property functions (and the handler, which is not used for vacuum), it is quite important to have a time-out when waiting to receive a message. This has been tried implemented in several ways :

A previous version used the *mqreceive* call for setting up a pending asynchronous receive, and would then receive the message with an *evtpoll*, which can be used with time-out. This procedure is extremely complicated and not very elegant, using signals and signal handlers etc., and it doesn't really work. Only a few messages can be received and then the whole thing crashes, requiring a reboot.

Another way is to have a not blocking (waiting) *mqreceive* call in a loop, but this method is not that elegant because it waists system resources. This version also crashes now and then.

The new way it is implemented in the present version of the message system is very elegant and it works perfectly well. The system *alarm* timer is set before the (blocking) *mqreceive* call, and when the timer expires it will interrupt the waiting *mqreceive* call. When the *mqreceive* call returns, the *errno* is examined. If the *errno* indicates no error, a message was received normally. If the errno indicates an interrupt, it means that there has been a time-out (interrupt from timer). All the other *errno's* of course indicate normal receive faults as usual. Note that although the *mqreceive* system call is interrupted by the signal, it returns in a perfectly controlled non hazardous manner, with errno set to EINTR.

The only little problem with the present implementation is, that when a process waiting for a message is killed, the virtual terminal window must sometimes be reset. This is the reason that the *setup_recv* function sets up the SIGINT handler, catching the normal kill signals. It was thought that if the *mqreceive* call could handle a SIGALRM signal when there was a handler, it would also be able to cope with a SIGINT signal without problems, if there was just a handler for the signal. Unfortunately it is not. However, since the final application runs as a background process, it is no problem in this case..

A little thought has also been given to the fact that the signal handlers will be set up a great many times (in the *setup_recv* function) if the function is called many times. This could eventually, although not likely, result in some kind of overflow. However, during test of the message system, the function

has been called repeatedly for more than two million times, without any problems showing.

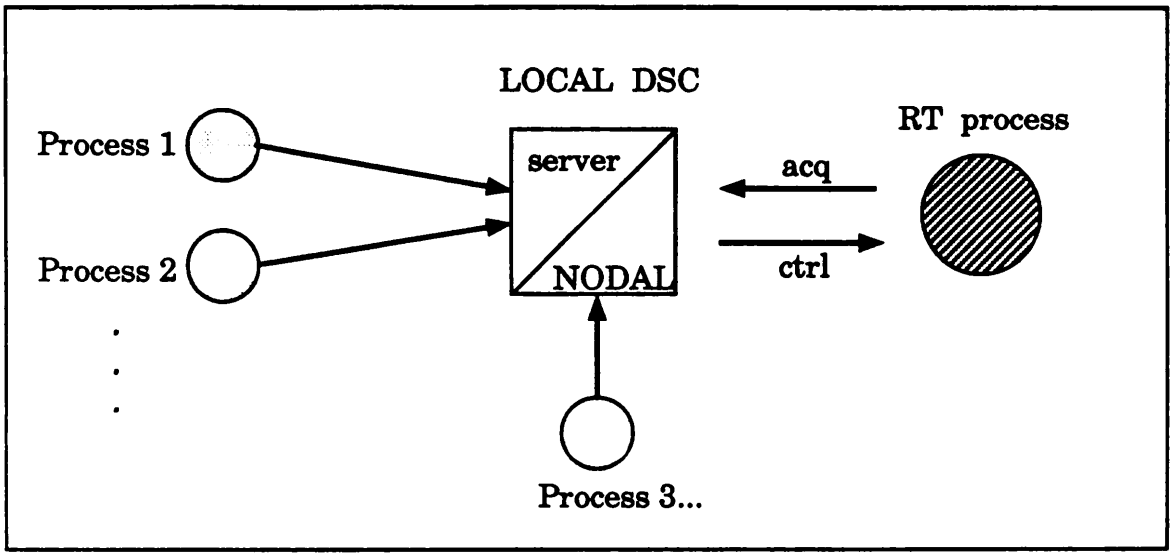
At the end a remark regarding the way the *mqlib.c* file is linked to the main programs, which is done in a quite unusual but smart way (see also chapter 2.2.5.2). The usual problem when having a set of functions that are shared by several programs, is that the corresponding data types (structures etc.), which the functions operate on, must often be known in the main programs when compiling. This is also the case for *mqlib.c*, where the two structures *setting_recv* and *setting_send* must be known in *vac_proco.c, rt.c* and *queue.c* when compiling The normal way to solve the problem is to put the structures in a header file (f. ex. *mqlib.h*), and include this in all the programs. Together the *mqlib.c* and the *mqlib.h* files would then form what could be called a module.

In the current implementation, to avoid having to deal with an extra file only containing two small structures, the *mqlib.c* file is divided in two : A header part and a code (implementation) part. The latter part is switched out via a compiler directive when the *mqlib.c* file is *included* in another program. That is, when the *mqlib.c* file is *included* in some other code, only the header part will be effective. When the *mqlib.c* file is compiled itself, the compiler must be given a directive (MQLIB_CODE) on the command line, and all the code in the file will then be effective. That is, the *mqlib.c* file is both linked to and included in the final program. This method also ensures consistency, because the *extern declarations* in the header will be compiled together with the *definitions*, and any disagreement between the two will thus lead to compilation error (In old C it is in fact not necessary with *extern* declarations of functions). In this way, a module can be contained in a single file ! In a project with a lot of different modules, this method would effectively clean up the mess of files. In this application however, it is just an idea, and the major problem will probably be that nobody is used to this practice. So, if someone don't like it, it's easy to put the header part in a separate file and do things the 1950 way.

2.2.2 Identification of messages

The control system for vacuum has a special characteristic, which is not known from other applications : Having several, and not just one Equipment Module, communicating via the same queues with a single specific process. Because of this, some complications that (might) arise regarding the identification of the messages must be investigated.

As illustrated in the figure, there is basically two ways to use an Equipment Module. Either it is called from application programs in other workstations across the network, via a Remote Procedure Call (as process 1 and 2 in the figure). This is the normal way of operation, and in this case the Equipment Module is activated through the *server*. The other way is to access the Equipment Module directly from the local DSC, via a local running version of NODAL (as process 3 in the figure).



As there are several different application programs that can activate an Equipment Module at the same time, and thus ask for an acquisition message to be returned, attention must be given to the question of how to make sure that the different Equipment Modules receive the correct acquisition messages.

In the DSC there is only one DT, shared by all the Equipment Modules installed in the DSC. The *server* is linked to the three different vacuum Equipment Modules : *vpump, vgaug* and *vvalv*. These three EM's are thus *a single one threaded process*. As program execution can only be one place at a time, it means that only one external process (application progr.) can be served at a time. This again means that in these three EM's only one acquisition message can be send and waited for at a time, and the acquisition messages will therefore always be received by the correct EM.

The problems can arise when a local running NODAL (or several) is introduced. As they are separate processes, they will indeed be able to send control messages at the same time (and at the same time as the server process). If two such separate processes ask for an acquisition message to be returned, then there will also be two processes *at the same time* waiting to receive an acquisition message. There is no guarantee for, which one will receive the first of the acquisition messages send from the RT process, and thus it might be the wrong one. A way to identify the messages is needed.

There is, however, a little point that makes the mixing up of the messages quite unlikely to happen. Because the RT process runs with a higher priority than the other processes, when it receives the first of the two (or more) control messages, it will occupy the CPU totally. The second process will thus not get time to send any control message, and will therefore not be waiting for any. This is unfortunately not any solution to the problem that can be totally relied on, since there might be some time gabs at different places etc., and maybe the RT process relinquishes the CPU while waiting for, say, an external bus.

It is of course obvious to use the (operating system) message type to make the identification, especially as the processes can be set up to receive only certain types of messages (numbers). The only question left is, which number ?

It will not work using the EM number, as different processes might access the same EM at the same time. Instead the number must distinguish not between EM's, but between processes. It is therefore likely to use the PID number (Process ID). This has some very great advantages : There is no need for some global definitions of which processes have which numbers etc., a thing that would be very tedious to manage as the number of processes are not known. Any number of processes accessing the queues can be installed. Furthermore, the allowed range of PID numbers matches the valid range of message type numbers : The message type must be between 1 (not zero) and the maximum of a long int. A PID number for a normal process can not be 0 (nor negative), and the type of the value returned from the *getpid* system call is exactly an (long) int. The PID number is assigned to both the *setting_recv.type* and *setting_send.type* fields (in the *setup_recv/setup_send* functions) in the *mqlib.c* file (appendix, chapter 3.1.1).

There is only two small disadvantages using the PID number. The first is not at all any problem in the final installation, but should be noted while testing : If a (EM) processes is killed while waiting for an acquisition message, the message in the queue will (probably) be left forever, until the queues are restarted. The reason is of course that the message has a type that doesn't match any of the process's PID. Even if the process that was killed is restarted, it might get another PID number than it had before. However, in the final installation this might in fact be an advantage, except that the message takes up some space in the queue buffer.

The other disadvantage is that it will be quite difficult to install a handler process, should it be needed in the future (or in applications for other

types of equipment). The reason is of course that all the message types are sort of occupied. The only possibility is that the message type for an asynchronous acquisition message is the PID number of the handler process, but it will be difficult for the specific process to get that number (except if the handler is a child of the specific process, although it would not be very elegant). Otherwise a (high) number can eventually be chosen, which will (probably) never be a PID number.

Note that the specific process receives *all* types of messages, but the acquisition message 'answer' to a specific control message must have the same type as the one received. This means, the only thing the specific process has to do, is simply to copy the type from the ctrl msg. to the acq. msg., and not think about which type it is.

In previous applications where a message system has been used (with only one Equipment Module), the message type was used to distinguish between synchronous and asynchronous messages. Now, this is not the case for the vacuum system, where the message type has become a sort of return/destination address. All information about what a message contains and what to do with it (and therefore also the synchronous/asynchronous information), is now put in *one* single place : the *amm* field *inside the message itself* (see definition of *amm* values in chapter 2.1.4). This is of course only possible because the different messages all have the same *header* part. The only thing the specific process has to do upon receipt of a message is to read the *amm* field, and from this decide what to do next.

This chapter has mainly concerned the problems when accessing an Equipment Module from different places (processes) at the same time. However, in normal operation, access to the specific process will only go through the *server*. There will normally not be any local NODAL installed. The problems discovered above are thus quite theoretic (see also end of chapter 2.2.3.1), and they will probably never happen in practice.

At the end of this chapter, a little remark that corresponds to the discussion in the beginning, and which has not really anything to do with the message types. : The at any time maximal number of control messages on the control queue is equal to the max. number of acquisition messages on the acquisition queue, which is the same as the number of (EM) *processes* accessing the queues. The number of such processes will probably never exceed 2, the server and a local NODAL process. This is quite important, because from this, when knowing the size of the messages, the necessary size of the message queues can be found. In the *queue.c* program (appendix, chapter 3.1.2), the size of the two message queues is set to default, which is 3968 bytes (max number of messages is 55). The size of the messages is found in chapter 2.2.5.1, and one can see that the queues are far big enough.

2.2.3 The datatable

2.2.3.1 Definition of the data columns in the DT

From the specifications in chapter 2.1.2, it appears that the DT must contain a TRM and a SNE column, plus columns for all the control and acquisition values.

The control values that *must* be stored in the DT are :

- All the fields named in the *data* part of the control message.
- A *specialist* parameter (also in the *header* of the control message).

and the acquisition values :

- All the fields named in the *status* part of the acquisition message.
- All the fields named in the *data* part of the acquisition message.
- A *specialist* parameter (also in the *header* of the acquisition message).
- A *date* parameter (also in the *header* of the acquisition message).

Having a separate column for each of the above fields would result in a total of 19 columns in the DT, which for practical reasons is very much. The list of the parameters (columns) accessed by each of the properties will be very long, and copying ten or more parameters from f. ex. an acquisition message and into separate columns in the DT will be very cumbersome, although it is of course possible. Furthermore, because the DT columns can only hold values of type EqmInt (integer) or EqmFloat (double), it is necessary to check the range and cast the values to/from the messages.

An easy solution to this, resulting in very simple code, is to store the *whole* message(s) directly as a structure in the DT. As it appears from above, nearly all the fields in the messages are to be stored in the DT anyway, so this approach only results in *lin1-3* and *amm* (plus *date* for the ctrl. msg.) being stored also. This really doesn't matter, it could be used for a property reading which equipment had been accessed last.

Because the message system is able to store/receive the message to/from anywhere in the programs memory (via a pointer), and therefore also directly in the DT, the only thing to do is to define columns in the DT which are large enough to contain a whole message. This can be done by declaring a column as an array of EqmInt's, so that the size becomes larger than the message size.

By storing the whole control message and the whole acquisition message in the DT as single items, the number of columns in the DT is greatly reduced to only four :

- Control message column (BUF1). Defined as array of EqmInt.
- Acquisition message column (BUF2). Defined as array of EqmInt.
- Treatment column (TRM).
- Serial Number (SNE).

Looking forward to chapter 2.2.5.1, we see that BUF1 and BUF2 must be able to contain at least 62 and 70 bytes respectively. With the size of BUF1 = BUF2 = 25 EqmInt's = 100 byte there are room for a little future expansion, without having to redefine the datatable !

To access the specific control or acquisition values in the DT, it is of course necessary to know the starting address of the array, and then via the message structures access the correct field. The program code will be just as clear as with the 'normal' way to use the DT. The only really disadvantage is, that it was probably not originally the intention with the DT to do it this way, but anyway it works perfect.

Below is a printout from the DT, showing the general definition of the Equipment Module *vpump*, and the definition of the four data columns. The definitions for the Equipment Modules *vgaug* and *vvalv* are similar, only the Equipment Module Number (Classno) differing (*vpump* = 230, *vgaug* = 231, *vvalv* = 232) :

```
Classname VPUMP         Classno 230     Category EM     PPM 0

Superclass EMCLASS  CMCLASS                         Alarms N

Created 03-APR-92   by  ROSENSTEDT
Updated 03-JUN-92   by  ROSENSTEDT

Short Description : Vacuum pumps

Proco Source File : vac_proco


DEFINITION OF CLASS AND INSTANCE VARIABLES

CI RW  Varname   Typ/Dim   Description

I  RO  SNE       I 1       Serial Number
I  RO  TRM       I 2       Treatment
I  RW  BUF1      I 25      Control message
I  RW  BUF2      I 25      Acquisition message
```

Although not important, attention should be given to a minor disadvantage by storing the whole control message as one piece in the DT. Problems arise because it is not possible only to access the values in the DT which are affected by a given property. To alter a few values, the whole message is copied from the DT, and when the property is finished, it is copied back (when declared as RW). Fields not affected by the property will thus remain unchanged.

From the discussion in the previous chapter, a piece of equipment can be accessed from both the control room and from the local DSC at the same time (although it must be highly confusing for the operators). This can in fact lead to a malfunction : Consider a situation where one process calls the *ccsact* property, and another process then slightly afterwards calls f. ex. the *ccv* property (before the *ccsact* property exits, and for the same equipment number). The *ccsact* takes a copy of the control message in the DT, and so does the *ccv*. It is now in fact quite likely that the *ccv* will exit as the first, because it won't have to send a message (*ccv* and *ccv1* sends no messages). Therefore *ccv* will put the control values (the whole message) back into the DT as the first, and afterwards the *ccsact* will do the same, overriding the changes made by *ccv*. That is, in this example the *ccv* call didn't achieve anything. For a property to fail, it is in fact enough that one property is activated before another is finished! It might be very unlikely to happen, but it's possible. The solution would of course be to have a lock on the access to the datatable.

Note that even if the values were put into the DT in the normal way, the same problem would of course appear if two properties had to access the same field in the DT. This is however not necessarily the case for vacuum, but it could be, if f. ex. all the *xxxx_chng* flags were concentrated in one field.

For completeness, and for the sake of the test programs in later chapters, the few equipments and the corresponding RO columns (the same for all three EM's) until now defined in the datatable is listed below. The values in the TRM[1] and SNE columns are chosen so that they are easily recognizable when testing.

Eq. no.	TRM[0]	TRM[1]	SNE
20001	1	10	11
20002	2	20	22
20003	3	30	33
20004	4	40	44
20005	5	50	55
20006	6	60	66
20007	7	70	77
20008	8	80	88

2.2.3.2 Definition of the property functions in the DT

The *property* functions are those called from the application program. When a property function call is received by the Equipment Module, program execution is transferred on to the corresponding *proco* function, which is a normal function in C. The relationship between the property and the proco function is determined in a special part of the datatable.

The definition of the property/proco functions in the DT is very straightforward. In fact, the definitions closely follows the specifications given in chapter 2.1.5. Below is a printout from the DT, showing all the newly defined property functions for the Equipment Module *vpump*. Note that the EM number by convention is included in the proconame.

```
Property Baseclas RW Proco    C/I Parameter-list

AQN      VPUMP     R R230AQN  I   BUF1, BUF2, TRM, SNE
AQN1     VPUMP     R R230AQN1 I   BUF1, BUF2, TRM, SNE
AQN2     VPUMP     R R230AQN2 I   BUF1, BUF2, TRM, SNE
ASPECT   VPUMP     R R230ASPE I   BUF2, TRM
BUSY     VPUMP     R R230BUSY I   BUF2, TRM
CCSACT   VPUMP     W W230CCSA I   BUF1, TRM, SNE
CCV      VPUMP     W W230CCV  I   BUF1, TRM
CCV1     VPUMP     W W230CCV1 I   BUF1, TRM
DATE     VPUMP     R R230DATE I   BUF2, TRM
INTLK    VPUMP     R R230INTL I   BUF1, TRM, SNE
INTLKM   VPUMP     R R230INTM I   BUF1, TRM, SNE
PHSTAT   VPUMP     R R230PHST I   BUF2, TRM
RFAULM   VPUMP     R R230RFAM I   BUF1, TRM, SNE
RFAULT   VPUMP     R R230RFAU I   BUF1, TRM, SNE
STAQ     VPUMP     R R230STAQ I   BUF1, BUF2, TRM, SNE
TBIT     VPUMP     R R230TBIT I   BUF1, BUF2, TRM, SNE
TBIT     VPUMP     W W230TBIT I   BUF1, BUF2, TRM, SNE
TEST1    VPUMP     R R230TST1 I   BUF1, BUF2, TRM, SNE
TEST1    VPUMP     W W230TST1 I   BUF1, BUF2, TRM, SNE
UFAULM   VPUMP     R R230UFAM I   BUF1, TRM, SNE
UFAULT   VPUMP     R R230UFAU I   BUF1, TRM, SNE
WARN     VPUMP     R R230WARN I   BUF1, TRM, SNE
WARNM    VPUMP     R R230WARM I   BUF1, TRM, SNE
```

The parameter-lists expresses which parameters are used by the proco function (see chapter 2.2.4). Unfortunately the list generated from the DT above does not indicate the RO/RW/WO status of the parameters in the parameter-list. This information is therefore written as comments in the proco source code (*vac_proco.c*).

The property functions defined in the other Equipment Modules *vgaug* and *vvalv* are exactly the same as for *vpump*, except that the number in the names are 231 for *vgaug* and 232 for *vvalv*.

2.2.4 *Principles in writing property functions*

In order to develop a full functioning Equipment Module, a huge amount of existing code is loaded upon. Because of the complication of this general software, it is impossible to give a full description here, which is indeed not even needed. However, in order to make it possible for the reader to understand the property code presented in the following chapters, a brief description of the 'external' code referred to is given below.

The property functions are actually in connection with three distinct 'worlds'. One is in the function call itself, in the (standard) way the property functions are invoked from the application program via the *frame*. Then there is the access to the *datatable*, and the interface to the equipment, in this application via a *message system*. The latter is part of the present development, and is described in the previous chapter.

First of all, the property code must be linked to the *frame*, that is, the general software that receives the standard property function call from the application program, checks various parameters, and after a look in the datatable passes the program execution on to the correct piece of proco code. The compilation and linking is accomplished via a makefile (*makenodal*), listed in appendix, chapter 3.1.10.

There is no need to understand the internal workings of the frame in order to understand the property code, only the standard calling sequence of the property functions (which at the same time is the interface to the variables in the DT) must be known. This interface is accomplished via a macro included from the file *proco_header.h* :

```
#define sproco(name,record,value_type)    \
name(dtr,value,size,membno,plsline,coco) \
record *dtr;                              \
value_type *value;                        \
int  size;                                \
int  membno;                              \
int  plsline;                             \
int *coco;
```

name : The name of the proco (property function), as defined in the DT. The name is not the same as the property function name seen from the application program.

record : A structure containing the fields in the DT accessed by the proco. *Each* property has a structure associated to it describing the fields of the tables being accessed. Before accessing the proco by the frame, this structure is filled with the values of the fields marked RW or RO. No initialization is done for WO parameters. When the property returns, those fields marked RW or WO are written into the DT, i.e. the DT is updated. Note that the proco function works on a copy of the datatable.

value_type : The type of the values contained in the array, pointed to by *value*, that is passed to or from the application program.

dtr : Pointer to the structure containing the fields accessed by the proco. The pointer is of course already initialized by the frame when the proco is invoked.

value : A pointer to the array passed to or from the application program. This pointer is also already initialized by the frame when the proco is invoked.

size : Number of elements in the **value* array.

membno : The equipment number being accessed.

plsline : The pls being accessed, not in use for vacuum.

**coco* : The completion code returned by the proco. Several standard coco errors are defined in the file *gm_constants.h*.

The *gm_constants.h* file also contains the Equipment Module numbers, defined as constants (*vpump* = 230, *vgaug* = 231 and *vvalv* = 232). Another file which is included from the property code is the *gm_types.h*. This file contains among other things the definition of an EqmInt (long int) and an EqmFloat (double).

To compile and install three full functioning Equipment Modules for local access (via NODAL), the following procedure must be followed :

- Create the files *gm_pbt.c* and *gm_dt.c* from ORACLE (DSC : *dtest2*).
- Copy the files created by the ORACLE database (dir. : */tmp*) to the working directory, which in the development phase is */u/rosenst/dsc/vac*.
- Use the *makenodal* and *makecreat* makefiles to compile and create the *nodal** and *creadt** programs.
- Use *makequeue* and *makert* makefiles to create the *queue** and *rt** programs.
- Logged in on the local DSC, start the programs *creadt** and *queue**, which creates the datatable and the two queues. If a datatable is already installed it can be removed with *ipcrm -M [key]*, where *key* is found with *ipcs*.
- Run the *nodal** and *rt** (test) programs in separate windows on the local DSC, or run the *rt** as a background process.

2.2.5 Property functions

2.2.5.1 The message structures

The message structures, which are common to all the Equipment Modules, are defined in the *message.h* file, listed in appendix, chapter 3.1.3. The file contains the four message structures : *req_msg*, *ctrl_msg*, *acq_msg* and *status_msg*. To make changes easier, all types in the messages are defined via the intermediate types *vac_xxxx*, which are derived from basic C types. Note the use of a *union* to switch between the data or test field.

It might have been a good idea to define separate types for every single field in the messages, i.e. a *lin1_type*, *amm_type* etc. It would add a little to the complexity of the message structure definitions, but in case of changes it would only be necessary to change the type definition at one place, and not as it is now, at many places throughout the proco code.

The *message.h* file is of course to be included by both the property code and the specific process, since the message structures are to be known in both the sending and the receiving part. Hence, this file is a good place to define other constants which are common to the two processes, rather than defining them twice, separately in each process. Therefore the last part of the file contains definitions regarding the types used, and especially the meaning of some of the fields in the message structures. Values for fields, which are defined in the specifications, but which the properties do not depend on, are not defined in this file, as they are only of interest to the specific process.

The *message.h* file mostly contain type definitions and preprocessor directives (true header file), and therefore there is no reason to apply the method used for *mqlib.c*, as most of *message.h* can be included in *vac_proco.c* without problems (see chapter 2.2.5.2). However, two variables are defined : *ctrlq_name* and *acqq_name*. To overcome the problem of including defined variables, they are defined *static*, meaning there will be a copy for each EM in the final *nodal** code. This is better than using a *#define*, which would result in the string being placed many times in the code.

To make the BUF1 and BUF2 columns in the datatable large enough, it is necessary to know the size of the *ctrl_msg* and the *acq_msg*. Finding the size manually from the structures in the *message.h* file gives the following results :

req_msg : 22 bytes

ctrl_msg : 62 bytes

acq_msg : 70 bytes

status_msg : 102 bytes

Exactly the same results are obtained from the *sizeof* operator, which indicates that the compiler doesn't add any alignment bytes. However, if the test field in the *ctrl_msg* does not contain an even number of bytes, the compiler adds one alignment byte. This doesn't matter in the present application, because both the sender and the receiver of the messages are programmed in C code and compiled with the same compiler. But in other applications it could cause some problems if the sender and receiver process are programmed in different languages, and thus perhaps implements alignment in another way or not at all.

2.2.5.2 The property code

As stated in the specifications, all the equipment is divided in three groups, and there is therefore three different Equipment Modules. Having three different Equipment Modules is the same as three different sets of property functions. However, since the division of the equipment in Equipment Modules is only made for convenience reasons, the property functions in the different Equipment Modules for vacuum are in fact almost identical. This is why the property specifications in chapter 2.1.5 do not distinguish between different Equipment Modules.

This leads to the discussion of, if and how the property functions can be implemented in the same sourcecode file. This would be very advantageous when correcting and testing the code. There are several ways to do it :

One is to write all the property functions so that they are exactly identical in all Equipment Modules. A disadvantage is that there will be some code in all of the EM's which will never be used (f. ex. code for valves in the *vpump* EM etc., how little it may be), and that it will be possible to access f. ex. a pump from the *vgaug* EM, which can certainly not be allowed. The latter could be avoided by checking that the accessed Equipment Type belongs to the correct EM number, but this would make a not desired connection in the code between these two numbers, in case the numbers should change (although, why should they change ? The EM Number is already found in the name of all the proco functions !).

Another way, and the one which has been chosen, is to use #IFDEF and #ENDIF compiler directives to switch in/out the (small) parts of the code specific to certain Equipment Modules, and leave the general parts. This method has the extremely great advantage, that changes which are made immediately applies to all of the Equipment Modules. There are not the problems and dangers in having three nearly identical versions of code, that must all be updated simultaneously. The disadvantages are that the code might get a little confused, and that there are some problems by having some (not proco) functions and variables in the same file. This is discussed in greater detail in the following.

Thus, the complete source code for all the properties in all the Equipment Modules are located in the *vac_proco.c* file, listed in appendix, chapter 3.1.4. To switch between which Equipment Module the code is compiled for, one of the following preprocessor variables must be set (-Dxxxx compiler option) when compiling : *vpump_proco*, *vgaug_proco* or *vvalv_proco*.

The property code is not very complicated, and as many of the properties are more or less identical, only the general outline of a few of the property functions will be described below.

First a few comments to the **global definitions** in the beginning of the file :

Via conditional compilation the EQ_MOD_NO is assigned the correct EM number. The EM numbers are written directly in the code, as it is more likely that the EM name changes than that the numbers do. However, this could also have been done by reading a (new) MO_NO column in the DT. At the same time a macro is defined, which includes the EM number in the proco function names.

Some other macro definitions follow : *check_size*, *check_int* and *set_lin*. The reason they are defined as macros rather than functions is that some of the types they operate on are not the same in the different proco functions. F. ex. in the *set_lin* macro, the *dtr* pointer has different types in the different proco functions.

Next follows the definitions that have a connection with the definitions in the DT, and which must therefore always correspond exactly to the DT. The *xxxx_dtr* structures contain the list of parameters in the DT (order is significant) which are accessed by the procos. These structures must correspond to the DT definitions (parameter list) listed in chapter 2.2.3.2.

Some *coco* error constants are defined, which links the *coco* names used in the proco code with some of the *coco* error numbers defined in the *gm_constants.h* file. This makes it easy to change the codes, which is important because some of the *coco* numbers are not finally defined yet.

The seven **normal functions** are used by most of the proco functions. Note that there are a total of four global variables : *ctrlq_setting, acqq_setting, ctrlq_name* and *acqq_name*, the last two coming from the *message.h* file. The reason for using global variables is that it would simply be too tedious to pass them the normal way as parameters.

check_eq_type : Checks if the Equipment Type passed to the function belongs to the Equipment Module.

set_time : Sets the *vac_time* structure pointed to by *time_p* to the actual time.

check_qualif : The parameter passed to the function (the *qualif* field) is examined and, according to the bit definitions, a *coco* error code is returned for the most significant bit that is set, i.e. the most significant condition.

open_queues : Simply opens the control queue and the acquisition queue for writing and reading respectively, and sets up the *setting* structures. Since there is no general initialization routine available when writing proco functions, the queues must be opened and closed for every access. This does not cause any problems, see chapter 2.2.1.

close_queues : Closes the control and the acquisition queues.

send_async : Opens the queues, sends a message on the control queue and closes the queues again.

send_sync : Opens the queues, sends a message on the control queue, waits to receive a message on the acquisition queue and then closes the queues again.

If a function is present in the normal way in the *vac_proco.c* code, it will be defined in all of the object files created from the single source code file. It is of course not possible to link such object files containing duplicate function definitions. Therefore a method must be found to ensure that normal non proco functions are only present in the created object files once.

The function *check_eq_type* has the same name in all EM's, but as the content is different, it is necessary to make sure that one version of the function can not be seen from the other EM's when they have been linked. This is of course easily done by defining it *static*.

The other functions (and the two global variables) however, have the same contents in all EM's, and although they could also be defined *static* (would solve the problem with duplicate names), it is more elegant to have only one copy in the final code. This can be achieved by having an extra file with the functions, plus a header file to be included, or use the same method as for *mqlib.c*. However, a more elegant method is found, using the conditional com-

piler directive already used in the code, so that the functions are only effective with the *vpump* EM. The *extern* declaration of the functions ensure a compile error if anything is typed wrong. The *extern* declaration of the two global variables are necessary to be able to compile.

The same problem as above appears when *including* files with defined functions or variables. This is the reason that the full *mqlib.c* and *message.h* can not be included directly.

At last to the **proco functions** themselves :

In the ***w230ccsa* proco** (*ccsact* property) the type of the equipment accessed is first checked, and at the same time the size of the parameter passed via the property function is checked (see table with allowed values in chapter 2.1.3). The size limits could have been defined in a separate column in the DT, and then received from there, which would have been a more ideal solution. However, since it is said that the limits will never be changed, they have been put directly into the code. Next, all the fields in the general header of the control message are given new values. Again, because no initialization routine exists, the *lin* values f. ex. must be computed and stored in the message each time the proco is called. It is easier to compute them each time than to have a kind of flag indicating if the values are already OK.

Note that all the new values assigned are in fact written directly into the control message in the DT !, i.e. the DT is updated. When the message is send, the message data from the DT is read via the pointer. That is, no copying of the DT into an external buffer is necessary.

Before the message is send, the *ccsact_chng* field is set to CHANGED. After the message have been sent, the *ccsact_chng* field is set back to

NO_CHANGE, and the other *_chng* fields is set to NO_CHANGE (if valid). In this way, even if the sending of the message somehow fails, the *xxxx_chng* fields will contain correct values.

The ***w230ccv* and *w230ccv1* procos** (*ccv* and *ccv1* properties) are a little simpler than the *w230ccsa* proco, because they do not send any messages. The parameter passed via the property is just copied into the DT and the *_chng* field is set to CHANGED. For the *w230ccv*, the storing of the *ccv* value in the message is dependent on the Equipment Type, because the *ccv* field has different types (union).

The procos which receives acquisition values : ***r230staq*, *r230aqn*, *r230aqn1* and *r230aqn2*** (*staq, aqn, aqn1* and *aqn2* properties), are in fact very similar. As always, it is first checked that the equipment accessed belongs to the correct Equipment Module, and that the parameter (*aqn, aqn1* and *aqn2*) really exists for that equipment. Then new values are put into the header of the control message in the DT, and a request message is send. Note that the size of the acquisition message that is expected to be received must be set first.

When an acquisition message is received, it is put directly into the datatable, thus avoiding any copying. The value asked for is taken from the DT, and returned to the calling program. This is a bit complicated for *aqn*, because this field has different types (union) depending on the Equipment Type.

If there was no problems in the sending and receiving of the messages, the *qualif* field in the received acquisition message is examined. In this way, important errors from the message system itself is not overwritten by the less important error conditions from the message system.

The ***r230phst, r230aspe, r230date*** **and** ***r230busy*** **procos** (*phstat, aspect, date* and *busy* properties) are all very simple. First the usual check is made, that the equipment accessed belongs to the Equipment Module, and then the field in the acquisition message in the DT is returned. Note in the *r230busy* proco, if the busy time is too large to be returned to the calling program (because of the type of the parameter), a coco error will be returned. A bit strange, but necessary.

The ***r230warn, r230rfau, r230ufau*** **and** ***r230intl procos*** (*warn, rfault, ufault* and *intlk* properties) are in fact identical. There are only two small differences from the previously described procos : The acquisition message to be received is now a status message. Upon receipt the message is not stored directly in the datatable, but instead the message is put into the *statusmsg* variable, so that the acquisition values in the datatable are not overwritten. The appropriate field from the message is returned.

Very similar to the four procos above are the ***r230warm, r230rfam, r230ufam*** **and** ***r230intm*** **procos** (*warnm, rfaulm, ufaulm* and *intlkm* properties), which receives the corresponding dates. It can be seen from the code how an array of values are returned to the calling program.

The ***tbit*** **test property** is very simple, but unlike the properties above, there are both a read and a write version of the proco. The *r230tbit* proco simply returns the specialist field from the acquisition message in the datatable. The *w230tbit* proco writes the parameter into the *specialist* field in the control message in the datatable.

The other test property, ***test1***, has also a read and a write version of the proco : ***r230tst1*** **and** ***w230tst1***, which are a bit more complicated. Nothing

is changed in the datatable by these two procos, so the header of the message is computed and put in the *ctrlmsg* variable. As the *specialist* field to be send is in the datatable, it is copied into the *ctrlmsg* variable from the control message in the DT.

In the ***w230tst1*** **proco**, the array of test values is copied into the *test* field of the *ctrlmsg* variable and at the same time each value is checked for size. Finally the message is send, but as there is no *amm* value defined as 'this is a *test* acquisition message' (why can't there be such an *amm* value ?), the *amm* is set to NO_RET. This means that the specific process can not distinguish between a *test* message and a *normal* control message. Therefore, before calling the *test1* (write) property, the *tbit* (write) property must be called by the user, and the *specialist* field set to something the specific process can recognize, thus indicating that it is a *test* message. As the property functions are not dependent on this, the specification and implementation have only to do with the specific process.

There are no such problems with the ***r230tst1*** **proco**, because an *amm* value (RET_TEST) has been defined, indicating that a *test* message is to be returned. After the proco has send the *request* control message, it waits to receive a *test* acquisition message. Because the DT must not be affected by this proco, the received message is put in the variable *acqmsg*, and the array of test values are returned to the calling program. It might have been logical at least to update the DT with the *specialist* field and the *status* part of the acquisition message, but this is not so in the present specification.

2.3 Testing

2.3.1 The real time process for testing

To be able to test the property functions, a test real time program (*rt.c*) has been developed, which is listed in appendix, chapter 3.1.5. The test RT program receives and sends messages exactly as the final specific process will do, but it is of course not connected to any equipment, and is therefore neither assigned a higher priority.

The following **functions** are present in the *rt.c* file :

set_time : Stores the current time in the *vac_time* structure pointed to by *time_p*.

set_acq_vals : This function stores some test values in the acquisition message. To make testing easier, most of the values are taken from the control message :

All the fields in the *header* is copied from the control message, except the *date* field, which is assigned the actual time. The *aqn* fields are assigned some of the values from the corresponding *ccv* fields in the control message, and the fields in the status part of the message is assigned the *ccsact* value plus some constants, for checking that the values are really transferred. See the code for further details.

set_stat_vals : Some test values are stored in the *status* message. See the code for further details.

send_norm_acq, *send_stat_acq* and *send_test_acq* : First the new values are stored in the messages, and then the contents of the message is printed. The printing is accomplished via the functions included from the *msg_print.h* file (appendix, chapter 3.1.6). Afterwards the message is send. Note that the message type is set explicitly (*acqq_setting.type* = *ctrlq_setting.type_recv*), so that the message type send is the same as the one received.

In the ***main*** **part of the program**, the two queues are first opened and the *setting* structures initialized. Note that the buffer for receiving the con-

trol message is set to the size of the largest message that can be received, here of course the *normal* control message. Time-out is set to 0 sec., meaning there will never be a time-out, and the type to be received is set to 0, resulting in all messages being received no matter the type.

Upon receipt of a control message, the contents are first printed and the control fields in the message are stored. If the *specialist* field is zero, the message is treated as a *normal* control message and the control values are stored. If the *specialist* field is non zero, the message is treated as a *test* message, and the *test* field is stored. Depending on the *amm* field in the message received, either a *normal* acquisition, *test* acquisition or *status* message is send on the acquisition queue.

2.3.2 Testing with NODAL

Three very simple NODAL test programs have been written, which tests all of the property functions :

test1.nod : Used for testing the *ccsact, ccv, ccv1, staq, aqn, aqn1, aqn2, phstat, aspect, date* and *busy* properties, quite a number. However, since the procedure of testing is merely a matter of calling a property and examining the resulting printout of the messages and the *coco* error, it is quite simple. A listing of the *test1.nod* file, the printout when the program is run, together with a printout of the contents of the messages, are to be found in appendix, chapter 3.1.7.

test2.nod : Testing of the *warn, rfault, ufault, intlk, warnm, ufaulm, rfaulm* and *intlkm* properties. A listing of the *test2.nod* file and the printout from the program are shown in appendix, chapter 3.1.8. The quite comprehensive message contents are not shown.

test3.nod : Testing of the *tbit(r/w)* and *test1(r/w)* properties. A listing of the *test3.nod* file plus the printout from the program, is in appendix, chapter 3.1.9. The message contents are not shown.

The property functions have of course also been tested with other values and EM's than those shown in the test programs. According to the printout from the test programs and the printout of the message contents, everything seems to work as expected.

3 Appendix

3.1 Program listings

3.1.1 mqlib.c

```
/*************************************************************************/
/* CERN/PS/CO                                                            */
/* Date : 8/7 1992                                                       */
/* By   : Anker Rosenstedt                                               */
/* File : mqlib.c                                                        */
/* Uses :                                                                */
/* Status : Final                                                        */
/*          Timeout functions perfect.                                   */
/*          Program sometimes crash when killed in waiting mqreceive     */
/*          system call, but reboot NOT nescesarry.                      */
/* Descr. : Routines for message queue communication                     */
/*                                                                       */
/*                                                                       */
/*************************************************************************/



/* HEADER */

typedef struct {
    int         type_recv;/* Type of message that WAS received         */
    int         size;     /* Max size of message to be received        */
    int         fd;       /* Queue file descriptor                     */
    int         type;     /* Type msg TO BE recv., see man mqreceive   */
    unsigned    timeout;  /* Timeout in seconds. 0 -> never timeout    */
} setting_recv;

typedef struct {
    int         type;     /* Type of message to be send, see manual    */
                          /* for mqsend                                */
    int         size;     /* Size of message to be send                */
    int         fd;       /* Queue file descriptor                     */
} setting_send;

extern int setup_recv(setting_recv*, char*);
extern int setup_send(setting_send*, char*);
extern int msg_recv(setting_recv*, void*);
extern int msg_send(setting_send*, void*);



/* IMPLEMENTATION */

#ifdef MQLIB_CODE

#include <mqueue.h>
#include <errno.h>
#include <file.h>
#include <signal.h>
#include <stdio.h>


/* Function : quit ******************************************************/
void quit()
{         /* Catch kill signals, to make controlled exit               */
          /* Not used for anything. When process is killed mqreceive   */
          /* still crashes now and then, though reboot NOT nescessary  */
```

```
    printf("\nProcess killed\n");
    exit(0);
}



/* Function : catch ******************************************************/
void catch()
{                       /* Catch timeout (alarm) signal, do nothing    */
    return;             /* (interrupts mqreceive system call)          */
}



/* Function : setup_recv *************************************************/
int setup_recv(setting, queue_name)
setting_recv *setting;
char *queue_name;
{
 int qfd;

    setting->timeout = 10;      /* 10 seconds timeout (default)         */
    setting->type = getpid();   /* Only msg's of this type are received */
                                /* by default. Should be set to 0 to    */
                                /* receive oldest message on queue      */
    if ((qfd = open(queue_name, O_RDONLY, 0)) < 0)
            return(errno);
    setting->fd = qfd;

    signal(SIGALRM, catch);  /* Avoid process termination when timeout */
    signal(SIGINT, quit);    /* Catch kill signals and exit            */

    return(0);
}



/* Function : setup_send *************************************************/
int setup_send(setting, queue_name)
setting_send *setting;
char *queue_name;
{
 int qfd;

    setting->type = getpid();   /* Type message that are send (deflt.) */
    if ((qfd = open(queue_name, O_WRONLY, 0)) < 0)
            return(errno);
    setting->fd = qfd;
    return(0);
}



/* Function : msg_recv ***************************************************/
int msg_recv(setting, data_p)
setting_recv *setting;
void *data_p;
{
 static struct msgcb msg;

    msg.msg_type    = (long) setting->type;
    msg.msg_flags   = MSG_TRUNC;        /* Wait in system call         */
    msg.msg_bufsize = setting->size;    /* Max length of recv msg's    */
    msg.msg_data    = data_p;           /* Msg is copied to point loc. */

    alarm(setting->timeout);            /* Set timeout clock           */
    if (mqreceive(setting->fd, &msg) < 0) {
            alarm(0);                   /* Timeout clock off,          */
```

```
            return(errno);              /* in case other receive fault */
    }
    alarm(0);                           /* Msg recv, timeout clock off */
    setting->type_recv = msg.msg_type;  /* Msg_type received           */
    return(0);
}



/* Function : msg_send ************************************************/
int msg_send(setting, data_p)
setting_send *setting;
void *data_p;
{
 static struct msgcb msg;

    msg.msg_flags   = MSG_NOWAIT | MSG_COPY;
    msg.msg_type    = setting->type;
    msg.msg_length  = setting->size;
    msg.msg_bufsize = setting->size;
    msg.msg_data    = data_p;

    if (mqsend(setting->fd, &msg) < 0)
            return(errno);
    return(0);
}

#endif /* MQLIB_CODE */
```

3.1.2 queue.c

```
/*****************************************************************************/
/* CERN/PS/CO                                                                */
/* Date : 8/7 1992                                                           */
/* By   : Anker Rosenstedt                                                   */
/* File : queue.c                                                            */
/* Uses : To be linked with mqlib.c                                          */
/* Status : Final                                                            */
/* Descr. : Installs two message queue files                                 */
/*          If started with a command line parameter 'a', no queues          */
/*          are installed, only the attributes are printed                   */
/*          If started with a command line parameter 'c', no queue           */
/*          are installed, but the attributes are printed continuously       */
/*****************************************************************************/
#include <mqueue.h>
#include "message.h"
#include "mqlib.c"
#include <errno.h>
#include <stdio.h>



/* Function : print_q_attr ***************************************************/
int print_q_attr(q_name)
char *q_name;
{
 setting_recv setting;
 struct mqstatus mqstat;
 int err;

    if ((err = setup_recv(&setting, q_name)) != 0) {
            printf("Error opening '%s' queue : %d\n", q_name, err);
            return(1);
    }

    if (mqgetattr(setting.fd, &mqstat) < 0) {
            printf("Error getting '%s' queue attributes : %d\n"
              , q_name, errno);
            return(1);
    }
    printf("\nCurrent attributes for '%s' queue : \n", q_name);
                /* Note that in the manual (mqgetattr), the text for   */
                /* the two first and the two next fields in the        */
                /* mqstatus structure have been reversed (ver. 2.0.0)  */
    printf("mqmaxmsg (number of messages currently free) : %d\n"
      , mqstat.mqmaxmsg);
    printf("mqrsvmsg (max number of messages) : %d\n"
      , mqstat.mqrsvmsg);
    printf("mqmaxbytes (bytes currently free) : %d\n"
      , mqstat.mqmaxbytes);
    printf("mqrsvbytes (max number of bytes) : %d\n"
      , mqstat.mqrsvbytes);
    printf("mqwrap     : %d\n", mqstat.mqwrap);
    printf("mqmaxarcv  : %d\n", mqstat.mqmaxarcv);
    printf("mqcurmsgs  : %d (number of messages currently on queue)\n"
      , mqstat.mqcurmsgs);
    printf("mqsendwait : %d\n", mqstat.mqsendwait);
    printf("mqrcvwait  : %d\n", mqstat.mqrcvwait);

    close(setting.fd);
    return(0);
}
```

```
/* Main ***************************************************************/
main(argc, argv)
int argc;
char *argv[];
{
 int err;

    if (argc == 1) {                    /* Create queues               */

      /* Create control queue */
      unlink(ctrlq_name);               /* If file exists, delete it   */
      if (mkmq(ctrlq_name, MQ_PERSIST) < 0) {
              printf("\nError creating control queue : %d\n", errno);
              exit(1);
      }
      chmod(ctrlq_name, 0777);  /* Set permission to rwx for all users */
      printf("\nControl queue created\n\n");

      /* Create acquisition queue */
      unlink(acqq_name);
      if (mkmq(acqq_name, MQ_PERSIST) < 0) {
              printf("Error creating acquisition queue : %d\n", errno);
              exit(1);
       }
      chmod(acqq_name, 0777);
      printf("Acquisition queue created\n\n");
    }

    if ((argc == 1) || (*argv[1] == 'a')) {     /* Print attributes    */

      if (print_q_attr(ctrlq_name) != 0) {
              print_q_attr(acqq_name);
              exit(1);                  /* Error in reading ctrlq attr */
      }
      if (print_q_attr(acqq_name) != 0)
              exit(1);                  /* Error in reading acqq attr  */
      printf("\n");
      exit(0);
    }

    if ((argc == 1) || (*argv[1] == 'c')) {     /* Print attributes    */
                                                /* continuously        */
      for(;;) {
            if (print_q_attr(ctrlq_name) != 0) {
                print_q_attr(acqq_name);
                exit(1);
            }
            if (print_q_attr(acqq_name) != 0)
                exit(1);
            sleep(3);
      }
    }
}
```

3.1.3 message.h

```
/*************************************************************************/
/* CERN/PS/CO                                                            */
/* Date : 8/7 1992                                                       */
/* By   : Anker Rosenstedt                                               */
/* File : message.h                                                      */
/* Uses :                                                                */
/* Status : Final                                                        */
/* Descr. : Message structures for VACUUM (used for all EM's)            */
/*                                                                       */
/*                                                                       */
/*************************************************************************/



/*************************************************************************/
/* Common definitions                                                    */
/*************************************************************************/
typedef unsigned char      vac_byte;    /*  8 bit unsigned integer     */
typedef signed char        vac_byte_s;  /*  8 bit signed integer       */
typedef unsigned short int vac_int16;   /* 16 bit unsigned integer     */
typedef unsigned long int  vac_int32;   /* 32 bit unsigned integer     */
typedef float              vac_real;    /* 32 bit real                 */

typedef struct {                        /* Format of dates             */
    long int    sec;
    long int    usec;
} vac_time;

typedef union {                         /* Union used for ccv and aqn  */
    vac_byte    byte;
    vac_int16   int16;
    vac_int32   int32;
    vac_real    real;
} mul_type;

typedef struct {
    vac_int16   class_ev;               /* Class of events             */
    vac_int16   spec_ev;                /* Specific event              */
} event;



/*************************************************************************/
/* Define REQUEST (CONTROL) message structure                            */
/*************************************************************************/
typedef struct {
    vac_int16   lin1;           /* Logical Identification Numbers      */
    vac_int16   lin2;
    vac_int16   lin3;
    vac_int16   amm;            /* Acquisition Message Meaning         */
    event       pulsid;         /* Puls identifier                     */
    vac_time    date;           /* Time when this message was send     */
    vac_int16   specialist;     /* Specialist action                   */
} req_msg;



/*************************************************************************/
/* Define CONTROL message structure                                      */
/*************************************************************************/
typedef struct {
    vac_byte_s  ccsact_chng;    /* Flags ccsact value is changed/valid */
    vac_byte    ccsact;         /* Current Control Set ACTuation       */
    vac_byte_s  ccv_chng;       /* Flags ccv value is changed/valid    */
    mul_type    ccv;            /* Current Control Value               */
```

```
    vac_byte_s  ccv1_chng;      /* Flags ccv1 value changed/valid      */
    vac_real    ccv1;           /* Current Control Value 1             */
} ctrl_data;

#define TEST_LENGTH_CTRL 40     /* Length of test field                */
                                /* MUST agree with def. in DT          */
typedef union {
    ctrl_data   data;                           /* Data field          */
    vac_byte    test[TEST_LENGTH_CTRL];         /* Test field          */
} ctrl_u;

typedef struct {
    vac_int16   lin1;
    vac_int16   lin2;
    vac_int16   lin3;
    vac_int16   amm;
    event       pulsid;
    vac_time    date;           /* Time when this message was send     */
    vac_int16   specialist;

    ctrl_u      u;              /* Data or test field                  */
} ctrl_msg;



/*********************************************************************/
/* Define ACQUISITION message structure                              */
/*********************************************************************/
typedef struct {
    mul_type    aqn;            /* Acquisition value                   */
    vac_real    aqn1;           /* Acquisition value 1                 */
    vac_real    aqn2;           /* Acquisition value 2                 */
} acq_data;

#define TEST_LENGTH_ACQ 40      /* Length of test field                */
                                /* MUST agree with def. in DT          */
typedef union {
    acq_data    data;                           /* Data field          */
    vac_byte    test[TEST_LENGTH_ACQ];          /* Test field          */
} acq_u;

typedef struct {
    vac_int16   lin1;
    vac_int16   lin2;
    vac_int16   lin3;
    vac_int16   amm;
    event       pulsid;
    vac_time    date;           /* Time of ACQUISITIONS                */
    vac_int16   specialist;

    vac_byte    phys_status;    /* Physical status                     */
    vac_byte    saqn;           /* Read back actuation value           */
    vac_byte    aspect;         /* Not connected/local/remote          */
    vac_byte    qualif;         /* Warn/busy/rfault/ufault/interlock   */
    vac_int32   busy_time;      /* Busy time in seconds                */

    acq_u       u;              /* Data or test field                  */
} acq_msg;



/*********************************************************************/
/* Define STATUS (ACQUISITION) message structure                     */
/*********************************************************************/
typedef struct {
    vac_int32   list;           /* A bit for each warn/fault/interl    */
    vac_time    date_last;      /* Time of last warn/rfault/ufaul/intl */
    vac_time    date_imp;       /* Time of most important - - -        */
} exception;
```

```
typedef struct {
    vac_int16   lin1;
    vac_int16   lin2;
    vac_int16   lin3;
    vac_int16   amm;
    event       pulsid;
    vac_time    date;           /* Time when this message was send     */
    vac_int16   specialist;

    exception   warnings;       /* Warnings                            */
    exception   rfaults;        /* Resetable faults                    */
    exception   ufaults;        /* Unresetable faults                  */
    exception   interlocks;     /* Interlocks                          */
} status_msg;



/*************************************************************************/
/* Other definitions (common to properties and specific process)         */
/*************************************************************************/
/* Queue names */
static char ctrlq_name[] = "/sem/ctlmsgqueue";/* Ctrl queue path + name*/
static char acqq_name[] = "/sem/acqmsgqueue"; /* Acq queue path + name */

/* Range of vacuum types */
#define BYTE_MIN    0                   /* From file limits.h :        */
#define BYTE_MAX    0xff
#define BYTE_S_MIN -128
#define BYTE_S_MAX  127
#define INT16_MIN   0
#define INT16_MAX   0xffff
#define INT32_MIN   0
#define INT32_MAX   0xffffffff
#define REAL_MIN    1.17549436e-38      /* From file float.h :         */
#define REAL_MAX    3.40282346e38
#define EqmInt_MIN -2147483648          /* From file limits.h :        */
#define EqmInt_MAX  2147483647

/* Meaning af amm */
#define RET_ACQ      0          /* Return normal acquisition (request) */
#define RET_CTRL     1          /* Return control message (read back)  */
#define RET_TEST     2          /* Return acq. msg. for test           */
#define RET_CYCLE    3          /* Return ctrl. msg. cycle no.         */
#define RET_STAT     4          /* Return status (acq) msg.            */
#define NO_RET       5          /* NO ret. message (asynchronous)      */

/* Meaning of ccsact_chng, ccv_chng and ccv1_chng */
#define NO_CHANGE -1            /* Must be negative                    */
#define CHANGED    1            /* Must be positive                    */
#define NOT_VALID  0

/* Meaning of qualif (bits) */
#define QUALIF_ILOCK  0x10
#define QUALIF_UFAULT 0x08
#define QUALIF_RFAULT 0x04
#define QUALIF_BUSY   0x02
#define QUALIF_WARN   0x01

/* Definition of equipment type numbers */
#define P_GROUP 1
#define P_ION   2
#define P_SUBL  3
#define P_STAT  4
#define G_PIRAN 5
#define G_ION   6
#define V_VALVE 7
```

3.1.4 vac_proco.c

```
/************************************************************************/
/* CERN/PS/CO                                                           */
/* Date : 8/7 1992                                                      */
/* By   : Anker Rosenstedt                                              */
/* File : vac_proco.c                                                   */
/* Uses : To be linked with mqlib.c and frame etc.                      */
/* Status : Final                                                       */
/* Descr. : Property functions for ALL VACUUM EM : VPUMP, VGAUG, VVALV */
/*                                                                      */
/*          Compiler option must include flags -DVPUMP_PROCO,           */
/*          -DVGAUG_PROCO or -DVVALV_PROCO                              */
/*                                                                      */
/************************************************************************/
#include <gm_constants.h>
#include <gm_types.h>
#include "message.h"
#include "mqlib.c"
#include <proco_header.h>
#include <errno.h>
#include <time.h>



/* CONDITIONAL INCLUSION OF EQUIPMENT MODULE NO */

#ifdef VPUMP_PROCO

#define EQ_MOD_NO 230  /* Eq. Mod. No. should fit with gm_constants.h  */
/* Macro for proco headers */
#define proco_head(p_rw, p_name, p_record, p_value_type) \
            sproco(p_rw##230##p_name, p_record, p_value_type)

#endif
#ifdef VGAUG_PROCO

#define EQ_MOD_NO 231
#define proco_head(p_rw, p_name, p_record, p_value_type) \
            sproco(p_rw##231##p_name, p_record, p_value_type)

#endif
#ifdef VVALV_PROCO

#define EQ_MOD_NO 232
#define proco_head(p_rw, p_name, p_record, p_value_type) \
            sproco(p_rw##232##p_name, p_record, p_value_type)

#endif



/* MACRO DEFINITIONS */

#define check_size(v,l,h) if( (v<l)||(v>h) ) {*coco = LIMERR; return; }

/* Check if integral number, return usual coco error if not */
/* (value must be within INT32_MIN...INT32_MAX) */
#define check_int(v) if (v != (vac_int32) v) { check_size(1, 0, 0); }

/* Set lin1, lin2 and lin3 in ctrlmsg */
#define set_lin ctrl_p->lin1 = (vac_int16) EQ_MOD_NO;          \
        ctrl_p->lin2 = (vac_int16) ((dtr->trm[EQ_TYPE] << 8)   \
          | (dtr->trm[EQ_STYPE] & 255) );                      \
        ctrl_p->lin3 = (vac_int16) dtr->sno;
```

```
/* DATATABLE FORMATS, must fit with datatable definitions */

/* Meaning of TRM array */
#define EQ_TYPE 0
#define EQ_STYPE 1

/* Length of acq and ctrl message fields defined in DT */
/* MUST be >= the size of the corresponding struct., MUST fit with DT  */

#define CTRL_DT_LENGTH 25 /* 100 byte */
#define ACQ_DT_LENGTH 25  /* 100 byte */
typedef EqmInt ctrl_dt_type[CTRL_DT_LENGTH];
typedef EqmInt acq_dt_type[ACQ_DT_LENGTH];

typedef struct {
    ctrl_dt_type ctrlmsg; /*RW*/
    EqmInt       trm[2];  /*RO*/
    EqmInt       sne;     /*RO*/
} ccsact_dtr;                    /* Used for ccsact                      */

typedef struct {
    ctrl_dt_type ctrlmsg; /*RW*/
    EqmInt       trm[2];  /*RO*/
} ccv_dtr;                       /* Used for both ccv and ccv1           */

typedef struct {
    ctrl_dt_type ctrlmsg; /*RW*/
    acq_dt_type  acqmsg;  /*WO*/
    EqmInt       trm[2];  /*RO*/
    EqmInt       sne;     /*RO*/
} aqn_dtr;                       /* Used for both staq, aqn, aqn1-2      */

typedef struct {
    acq_dt_type  acqmsg;  /*RO*/
    EqmInt       trm[2];  /*RO*/
} read_acq_dtr;                  /* Used for phstat, aspect, date & busy*/

typedef struct {
    ctrl_dt_type ctrlmsg; /*RW*/
    EqmInt       trm[2];  /*RO*/
    EqmInt       sne;     /*RO*/
} except_dtr;                   /* Used for warn, rfault, ufault, intlk, */
                                /* warnm, rfaulm, ufaulm and intlkm      */

typedef struct {
    ctrl_dt_type ctrlmsg; /*RW*/
    acq_dt_type  acqmsg;  /*RO*/
    EqmInt       trm[2];  /*RO*/
    EqmInt       sne;     /*RO*/
} test_dtr;                      /* Used for tbit(r/w) and test1(r/w)    */


/* COCO ERROR CODES, (re)defined here because nobody knows the values  */

#define ENILL_COCO  ENILL       /* Equipment Number ILL                 */
#define TIMOUT_COCO TIMOUT      /* Timeout                              */
#define NOQ_COCO    NOQ         /* Queue error                          */
#define ILCK_COCO   1016        /* Interlock          (from qualif)     */
#define UFAUL_COCO  1008        /* Unresetable fault  (from qualif)     */
#define RFAUL_COCO  1004        /* Resetable fault    (from qualif)     */
#define BUSY_COCO   1002        /* Busy               (from qualif)     */
#define WARN_COCO   1001        /* Warning            (from qualif)     */
```

```
/* FUNCTIONS */

/* Function : check_eq_type ***********************************************/
static int check_eq_type(eq_type)    /* MUST be static because function */
int eq_type;                         /* has the same name in all 3 EM's,*/
{                                    /* but the content is different    */
#ifdef VPUMP_PROCO

    switch(eq_type) {
      case P_GROUP :
      case P_ION :
      case P_SUBL :
      case P_STAT :
              break;
      default :
              return(ENILL_COCO);        /* Not pump equipment          */
              break;
    }

#endif
#ifdef VGAUG_PROCO

    switch(eq_type) {
      case G_PIRAN :
      case G_ION :
              break;
      default :
              return(ENILL_COCO);        /* Not gauge equipm.           */
              break;
    }

#endif
#ifdef VVALV_PROCO

    switch(eq_type) {
      case V_VALVE :
              break;
      default :
              return(ENILL_COCO);        /* Not valve equipm.           */
              break;
    }

#endif

    return(NOERR);
}



/* FUNCTIONS and GLOBAL VARIABLES which should only be present in the  */
/* final code once (here with VPUMP). It thus makes the final code     */
/* more compact. It would be possible to declare all the functions     */
/* static as the one above, but the functions would then be included   */
/* several times, one time for each EM (= 3 times in present version). */
/* If someone do not like it this way, they can of course put the      */
/* functions in a separate file, and link it to the rest.              */

extern setting_send ctrlq_setting;
extern setting_recv acqq_setting;

extern void set_time(vac_time*);
extern int check_qualif(int);
extern int open_queues();
extern void close_queues();
extern int send_async(void*);
extern int send_sync(void*, void*);

#ifdef VPUMP_PROCO
```

```
setting_send ctrlq_setting;
setting_recv acqq_setting;


/* Function : set_time ************************************************/
void set_time(time_p)
vac_time *time_p;
{
 struct timeval timev, *time = &timev;
 struct timezone timez, *zone = &timez;          /* Not used            */

    gettimeofday(time, zone);
    time_p->sec = timev.tv_sec;
    time_p->usec = timev.tv_usec;
    return;
}


/* Function : check_qualif ********************************************/
int check_qualif(qualif)          /* Returns appropriate coco error for  */
int qualif;                       /* most important qualif condition     */
{
    if (qualif & QUALIF_ILOCK)  return(ILCK_COCO);
    if (qualif & QUALIF_UFAULT) return(UFAUL_COCO);
    if (qualif & QUALIF_RFAULT) return(RFAUL_COCO);
    if (qualif & QUALIF_BUSY)   return(BUSY_COCO);
    if (qualif & QUALIF_WARN)   return(WARN_COCO);
    return(NOERR);
}


/* Function : open_queues *********************************************/
int open_queues()
{
 int err;

    if ((err = setup_send(&ctrlq_setting, ctrlq_name)) != 0) {
            return(err);
    }
    if ((err = setup_recv(&acqq_setting, acqq_name)) != 0) {
            return(err);
    }

    return(0);
}


/* Function : close_queues ********************************************/
void close_queues()
{
    close(ctrlq_setting.fd);
    close(acqq_setting.fd);
    return;
}


/* Function : send_async **********************************************/
int send_async(send_p)
void *send_p;
{
    if (open_queues() != 0) {
            close_queues();
            return(NOQ_COCO);           /* Queue error                 */
    }
```

```
    if (msg_send(&ctrlq_setting, send_p) != 0) {
            close_queues();
            return(NOQ_COCO);
    }
    close_queues();
    return(NOERR);
}


/* Function : send_sync ************************************************/
int send_sync(send_p, recv_p)
void *send_p;
void *recv_p;
{
 int err;

    if (open_queues() != 0) {
            close_queues();
            return(NOQ_COCO);
    }

    if (msg_send(&ctrlq_setting, send_p) != 0) {
            close_queues();
            return(NOQ_COCO);
    }

    err = msg_recv(&acqq_setting, recv_p);
    switch(err) {
      case 0 :                          /* Reception OK              */
              close_queues();
              return(NOERR);
              break;
      case EINTR :                      /* Timeout                   */
              close_queues();
              return(TIMOUT_COCO);
              break;
      default :                         /* Receive fault             */
              break;
    }
    close_queues();
    return(NOQ_COCO);
}

#endif /* VPUMP_PROCO */


/* PROPERTY FUNCTIONS **************************************************/

/* Function : w___ccsa *************************************************/
proco_head(w, ccsa, ccsact_dtr, EqmInt)
{
 ctrl_msg *ctrl_p;
 ctrl_data *data_p;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    data_p = (ctrl_data *) &(ctrl_p->u.data);


#ifdef VPUMP_PROCO

    switch(dtr->trm[EQ_TYPE]) {
      case P_GROUP :
      case P_ION :
      case P_SUBL :
              check_size(*value, 1, 2);
              break;
```

```
      case P_STAT :
              check_size(*value, 1, 3);
              break;
      default :
              *coco = ENILL_COCO;/* Bad equipment no. Not pump equipm. */
              return;
              break;
    }

#endif
#ifdef VGAUG_PROCO

    switch(dtr->trm[EQ_TYPE]) {
      case G_PIRAN :
              check_size(*value, 1, 2);
              break;
      case G_ION :
              check_size(*value, 1, 3);
              break;
      default :
              *coco = ENILL_COCO;/* Bad equipment no. Not gauge equipm.*/
              return;
              break;
    }

#endif
#ifdef VVALV_PROCO

    switch(dtr->trm[EQ_TYPE]) {
      case V_VALVE :
              check_size(*value, 1, 2);
              break;
      default :
              *coco = ENILL_COCO;/* Bad eq. no. Not valve equipm.      */
              return;
              break;
    }

#endif


    set_lin;
    ctrl_p->amm = NO_RET;       /* Don't send acq. msg. back          */
    data_p->ccsact_chng = CHANGED;
    data_p->ccsact = (vac_byte) *value;
    ctrlq_setting.size = sizeof(ctrl_msg);      /* Normal control msg. */
    set_time(&(ctrl_p->date));  /* Time right before message is send   */

    if ((*coco = send_async(ctrl_p)) != NOERR) return;  /* Send msg.   */

    /* Update _chng fields in DT */
    data_p->ccsact_chng = NO_CHANGE;
    if (data_p->ccv_chng != NOT_VALID) data_p->ccv_chng = NO_CHANGE;
    if (data_p->ccv1_chng != NOT_VALID) data_p->ccv1_chng = NO_CHANGE;

    return;
}



/* Function : w___ccv ************************************************/
proco_head(w, ccv, ccv_dtr, EqmFloat)
{
 ctrl_msg *ctrl_p;
 ctrl_data *data_p;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    data_p = (ctrl_data *) &(ctrl_p->u.data);
```

```
#ifdef VPUMP_PROCO

    switch(dtr->trm[EQ_TYPE]) {
      case P_ION :
              check_size(*value, 1, 2);
              check_int(*value);
              data_p->ccv.byte = (vac_byte) *value;
              break;
      case P_SUBL :
              check_size(*value, INT32_MIN, INT32_MAX);
              check_int(*value);
              data_p->ccv.int32 = (vac_int32) *value;
              break;
      default :                 /* No ccv parameter for that equipment */
              *coco = ENILL_COCO;   /* or not pump equipment           */
              return;
              break;
    }

#endif
#ifdef VGAUG_PROCO

    switch(dtr->trm[EQ_TYPE]) {
       case G_ION :
               check_size(*value, INT32_MIN, INT32_MAX);
               check_int(*value);
               data_p->ccv.int32 = (vac_int32) *value;
               break;
       default :                /* No ccv parameter for that equipment */
               *coco = ENILL_COCO;   /* or not gauge equipment         */
               return;
               break;
     }

#endif
#ifdef VVALV_PROCO

    *coco = ENILL_COCO;         /* No ccv parameter for valves         */
    return;

#endif


    data_p->ccv_chng = CHANGED;
    return;
}



/* Function : w___ccv1 ************************************************/
proco_head(w, ccv1, ccv_dtr, EqmFloat)
{
 ctrl_msg *ctrl_p;
 ctrl_data *data_p;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    data_p = (ctrl_data *) &(ctrl_p->u.data);


#ifdef VPUMP_PROCO

    switch(dtr->trm[EQ_TYPE]) {
      case P_ION :
      case P_SUBL :
              break;
      default :
              *coco = ENILL_COCO; /* No ccv1 parameter for that equipm.*/
              return;             /* or not pump equipment             */
```

```
            break;
    }

#endif
#ifdef VGAUG_PROCO

    switch(dtr->trm[EQ_TYPE]) {
      case G_ION :
            break;
      default :
            *coco = ENILL_COCO;    /* No ccv1 param. for that equipm. */
            return;                /* or not gauge equipm.            */
            break;
    }

#endif
#ifdef VVALV_PROCO

    *coco = ENILL_COCO;          /* No ccv1 parameter for valves        */
    return;

#endif

    /* Check size of parameter */
    if (*value > 0) { check_size(*value, 0, REAL_MAX) }
    else { check_size( -(*value), 0, REAL_MAX); }

    data_p->ccv1 = (vac_real) *value;
    data_p->ccv1_chng = CHANGED;

    return;
}



/* Function : r___staq ************************************************/
proco_head(r, staq, aqn_dtr, EqmInt)
{
 ctrl_msg *ctrl_p;
 acq_msg *acq_p;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    acq_p = (acq_msg *) &(dtr->acqmsg);

    set_lin;
    ctrl_p->amm = RET_ACQ;              /* Get acq msg. back from RT   */
    ctrlq_setting.size = sizeof(req_msg);/* Send only request msg.,    */
    acqq_setting.size = sizeof(acq_msg);/* receive normal acq. msg.    */
    set_time(&(ctrl_p->date));

    /* Send message and put recv. msg. direct in DT */
    if ((*coco = send_sync(ctrl_p, acq_p)) != NOERR) return;
    *value = (EqmInt) acq_p->saqn;

    *coco = check_qualif(acq_p->qualif);/* coco from send function is  */
                                        /* the most important, will    */
    return;                             /* not be overwritten          */
}



/* Function : r___aqn *************************************************/
proco_head(r, aqn, aqn_dtr, EqmFloat)
{
 ctrl_msg *ctrl_p;
 acq_msg *acq_p;
```

```
    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    acq_p = (acq_msg *) &(dtr->acqmsg);


#ifdef VPUMP_PROCO

    switch(dtr->trm[EQ_TYPE]) {
      case P_ION :
      case P_SUBL :
      case P_STAT :
              break;
      default :                 /* No aqn parameter for that equipm.   */
              *coco = ENILL_COCO;   /* or not pump equipment           */
              return;
              break;
    }

#endif
#ifdef VGAUG_PROCO

    switch(dtr->trm[EQ_TYPE]) {
      case G_PIRAN :
      case G_ION :
              break;
      default :
              *coco = ENILL_COCO;   /* Not gauge equipment             */
              return;
              break;
    }

#endif
#ifdef VVALV_PROCO

    *coco = ENILL_COCO;         /* No aqn parameter for valves         */
    return;

#endif


    set_lin;
    ctrl_p->amm = RET_ACQ;
    ctrlq_setting.size = sizeof(req_msg);
    acqq_setting.size = sizeof(acq_msg);
    set_time(&(ctrl_p->date));

    if ((*coco = send_sync(ctrl_p, acq_p)) != NOERR) return;


#ifdef VPUMP_PROCO

    switch(dtr->trm[EQ_TYPE]) {    /* EQ_TYPE could also be taken from */
      case P_ION :                 /* the received acq. message        */
              *value = (EqmFloat) acq_p->u.data.aqn.byte;
              break;
      case P_SUBL :
              *value = (EqmFloat) acq_p->u.data.aqn.int32;
              break;
      case P_STAT :
              *value = (EqmFloat) acq_p->u.data.aqn.real;
              break;
      default :                         /* This will never happen      */
              *coco = ENILL_COCO;
              return;
              break;
    }

#endif
#ifdef VGAUG_PROCO
```

```
    switch(dtr->trm[EQ_TYPE]) {     /* EQ_TYPE could also be taken from */
      case G_PIRAN :                /* the received acq. message        */
              *value = (EqmFloat) acq_p->u.data.aqn.real;
              break;
      case G_ION :
              *value = (EqmFloat) acq_p->u.data.aqn.int32;
              break;
      default :                           /* This will never happen      */
              *coco = ENILL_COCO;
              return;
              break;
    }

#endif
#ifdef VVALV_PROCO

    *coco = ENILL_COCO;                   /* This will never happen !    */
    return;

#endif


    *coco = check_qualif(acq_p->qualif);
    return;
}



/* Function : r___aqn1 ************************************************/
proco_head(r, aqn1, aqn_dtr, EqmFloat)
{
 ctrl_msg *ctrl_p;
 acq_msg *acq_p;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    acq_p = (acq_msg *) &(dtr->acqmsg);


#ifdef VPUMP_PROCO

    switch(dtr->trm[EQ_TYPE]) {
      case P_ION :
      case P_SUBL :
              break;
      default :
              *coco = ENILL_COCO;
              return;
              break;
    }

#endif
#ifdef VGAUG_PROCO

    switch(dtr->trm[EQ_TYPE]) {
      case G_ION :
              break;
      default :
              *coco = ENILL_COCO;
              return;
              break;
    }

#endif
#ifdef VVALV_PROCO

    *coco = ENILL_COCO;
    return;

#endif
```

```
    set_lin;
    ctrl_p->amm = RET_ACQ;
    ctrlq_setting.size = sizeof(req_msg);
    acqq_setting.size = sizeof(acq_msg);
    set_time(&(ctrl_p->date));

    if ((*coco = send_sync(ctrl_p, acq_p)) != NOERR) return;

    *value = (EqmFloat) acq_p->u.data.aqn1;
    *coco = check_qualif(acq_p->qualif);

    return;
}



/* Function : r___aqn2 ************************************************/
proco_head(r, aqn2, aqn_dtr, EqmFloat)
{
 ctrl_msg *ctrl_p;
 acq_msg *acq_p;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    acq_p = (acq_msg *) &(dtr->acqmsg);


#ifdef VPUMP_PROCO

    switch(dtr->trm[EQ_TYPE]) {
      case P_ION :
      case P_SUBL :
              break;
      default :
              *coco = ENILL_COCO;
              return;
              break;
    }

#endif
#ifdef VGAUG_PROCO

    switch(dtr->trm[EQ_TYPE]) {
      case G_ION :
              break;
      default :
              *coco = ENILL_COCO;
              return;
              break;
    }

#endif
#ifdef VVALV_PROCO

    *coco = ENILL_COCO;
    return;

#endif


    set_lin;
    ctrl_p->amm = RET_ACQ;
    ctrlq_setting.size = sizeof(req_msg);
    acqq_setting.size = sizeof(acq_msg);
    set_time(&(ctrl_p->date));

    if ((*coco = send_sync(ctrl_p, acq_p)) != NOERR) return;
```

```
    *value = (EqmFloat) acq_p->u.data.aqn2;
    *coco = check_qualif(acq_p->qualif);

    return;
}


/* Function : r___phst ************************************************/
proco_head(r, phst, read_acq_dtr, EqmInt)
{
 acq_msg *acq_p;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    acq_p = (acq_msg *) &(dtr->acqmsg);
    *value = (EqmInt) acq_p->phys_status;

    return;
}


/* Function : r___aspe ************************************************/
proco_head(r, aspe, read_acq_dtr, EqmInt)
{
 acq_msg *acq_p;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    acq_p = (acq_msg *) &(dtr->acqmsg);
    *value = (EqmInt) acq_p->aspect;

    return;
}


/* Function : r___date ************************************************/
proco_head(r, date, read_acq_dtr, EqmInt)
{
 acq_msg *acq_p;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    acq_p = (acq_msg *) &(dtr->acqmsg);
    *value = (EqmInt) acq_p->date.sec;
    *(value+1) = (EqmInt) acq_p->date.usec;

    return;
}


/* Function : r___busy ************************************************/
proco_head(r, busy, read_acq_dtr, EqmInt)
{
 acq_msg *acq_p;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    acq_p = (acq_msg *) &(dtr->acqmsg);
    check_size(acq_p->busy_time, 0, EqmInt_MAX);
    *value = (EqmInt) acq_p->busy_time;

    return;
}
```

```
/* Function : r___warn ************************************************/
proco_head(r, warn, except_dtr, EqmInt)
{
 status_msg statusmsg;
 ctrl_msg *ctrl_p;
 status_msg *stat_p = &statusmsg;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    set_lin;
    ctrl_p->amm = RET_STAT;
    ctrlq_setting.size = sizeof(req_msg);
    acqq_setting.size = sizeof(status_msg);
    set_time(&(ctrl_p->date));

    if ((*coco = send_sync(ctrl_p, stat_p)) != NOERR) return;

    check_size(stat_p->warnings.list, 0, EqmInt_MAX);
    *value = (EqmInt) stat_p->warnings.list;

    return;
}



/* Function : r___rfau ************************************************/
proco_head(r, rfau, except_dtr, EqmInt)
{
 status_msg statusmsg;
 ctrl_msg *ctrl_p;
 status_msg *stat_p = &statusmsg;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    set_lin;
    ctrl_p->amm = RET_STAT;
    ctrlq_setting.size = sizeof(req_msg);
    acqq_setting.size = sizeof(status_msg);
    set_time(&(ctrl_p->date));

    if ((*coco = send_sync(ctrl_p, stat_p)) != NOERR) return;

    check_size(stat_p->rfaults.list, 0, EqmInt_MAX);
    *value = (EqmInt) stat_p->rfaults.list;

    return;
}



/* Function : r___ufau ************************************************/
proco_head(r, ufau, except_dtr, EqmInt)
{
 status_msg statusmsg;
 ctrl_msg *ctrl_p;
 status_msg *stat_p = &statusmsg;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    set_lin;
    ctrl_p->amm = RET_STAT;
    ctrlq_setting.size = sizeof(req_msg);
    acqq_setting.size = sizeof(status_msg);
    set_time(&(ctrl_p->date));
```

```
    if ((*coco = send_sync(ctrl_p, stat_p)) != NOERR) return;

    check_size(stat_p->ufaults.list, 0, EqmInt_MAX);
    *value = (EqmInt) stat_p->ufaults.list;

    return;
}


/* Function : r___intl ************************************************/
proco_head(r, intl, except_dtr, EqmInt)
{
 status_msg statusmsg;
 ctrl_msg *ctrl_p;
 status_msg *stat_p = &statusmsg;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    set_lin;
    ctrl_p->amm = RET_STAT;
    ctrlq_setting.size = sizeof(req_msg);
    acqq_setting.size = sizeof(status_msg);
    set_time(&(ctrl_p->date));

    if ((*coco = send_sync(ctrl_p, stat_p)) != NOERR) return;

    check_size(stat_p->interlocks.list, 0, EqmInt_MAX);
    *value = (EqmInt) stat_p->interlocks.list;

    return;
}


/* Function : r___warm ************************************************/
proco_head(r, warm, except_dtr, EqmInt)
{
 status_msg statusmsg;
 ctrl_msg *ctrl_p;
 status_msg *stat_p = &statusmsg;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    set_lin;
    ctrl_p->amm = RET_STAT;
    ctrlq_setting.size = sizeof(req_msg);
    acqq_setting.size = sizeof(status_msg);
    set_time(&(ctrl_p->date));

    if ((*coco = send_sync(ctrl_p, stat_p)) != NOERR) return;

    *value = stat_p->warnings.date_last.sec;
    *(value+1) = stat_p->warnings.date_last.usec;
    *(value+2) = stat_p->warnings.date_imp.sec;
    *(value+3) = stat_p->warnings.date_imp.usec;

    return;
}


/* Function : r___rfam ************************************************/
proco_head(r, rfam, except_dtr, EqmInt) /* Who decided there can be    */
{                                       /* only 4 describing letters ? */
 status_msg statusmsg;
 ctrl_msg *ctrl_p;
```

```
 status_msg *stat_p = &statusmsg;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    set_lin;
    ctrl_p->amm = RET_STAT;
    ctrlq_setting.size = sizeof(req_msg);
    acqq_setting.size = sizeof(status_msg);
    set_time(&(ctrl_p->date));

    if ((*coco = send_sync(ctrl_p, stat_p)) != NOERR) return;

    *value = stat_p->rfaults.date_last.sec;
    *(value+1) = stat_p->rfaults.date_last.usec;
    *(value+2) = stat_p->rfaults.date_imp.sec;
    *(value+3) = stat_p->rfaults.date_imp.usec;

    return;
}


/* Function : r___ufam ************************************************/
proco_head(r, ufam, except_dtr, EqmInt)
{
 status_msg statusmsg;
 ctrl_msg *ctrl_p;
 status_msg *stat_p = &statusmsg;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    set_lin;
    ctrl_p->amm = RET_STAT;
    ctrlq_setting.size = sizeof(req_msg);
    acqq_setting.size = sizeof(status_msg);
    set_time(&(ctrl_p->date));

    if ((*coco = send_sync(ctrl_p, stat_p)) != NOERR) return;

    *value = stat_p->ufaults.date_last.sec;
    *(value+1) = stat_p->ufaults.date_last.usec;
    *(value+2) = stat_p->ufaults.date_imp.sec;
    *(value+3) = stat_p->ufaults.date_imp.usec;

    return;
}


/* Function : r___intm ************************************************/
proco_head(r, intm, except_dtr, EqmInt)
{
 status_msg statusmsg;
 ctrl_msg *ctrl_p;
 status_msg *stat_p = &statusmsg;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    set_lin;
    ctrl_p->amm = RET_STAT;
    ctrlq_setting.size = sizeof(req_msg);
    acqq_setting.size = sizeof(status_msg);
    set_time(&(ctrl_p->date));

    if ((*coco = send_sync(ctrl_p, stat_p)) != NOERR) return;
```

```
    *value = stat_p->interlocks.date_last.sec;
    *(value+1) = stat_p->interlocks.date_last.usec;
    *(value+2) = stat_p->interlocks.date_imp.sec;
    *(value+3) = stat_p->interlocks.date_imp.usec;

    return;
}


/* Function : r___tbit ************************************************/
proco_head(r, tbit, test_dtr, EqmInt)
{
 acq_msg *acq_p;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    acq_p = (acq_msg *) &(dtr->acqmsg);
    *value = (EqmInt) acq_p->specialist;
    return;
}


/* Function : w___tbit ************************************************/
proco_head(w, tbit, test_dtr, EqmInt)
{
 ctrl_msg *ctrl_p;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    check_size(*value, INT16_MIN, INT16_MAX);
    ctrl_p->specialist = (vac_int16) *value;
    return;
}


/* Function : r___tst1 ************************************************/
proco_head(r, tst1, test_dtr, EqmInt)
{
 ctrl_msg ctrlmsg;
 acq_msg acqmsg;
 int n;
 ctrl_msg *ctrl_p;
 ctrl_msg *dt_ctrl_p;
 acq_msg *acq_p;
 vac_byte *test_p;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    /* NO new values are written into the DT                          */
    ctrl_p = &ctrlmsg;
    acq_p  = &acqmsg;
    test_p = (vac_byte *) &(acq_p->u.test);

    set_lin;
    ctrl_p->amm = RET_TEST;

    /* The specialist field must be copied from the DT */
    dt_ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    ctrl_p->specialist = dt_ctrl_p->specialist;

    ctrlq_setting.size = sizeof(req_msg); /* Request control msg and   */
    acqq_setting.size = sizeof(acq_msg);  /* normal acq msg. length    */

    set_time(&(ctrl_p->date));
    if ((*coco = send_sync(ctrl_p, acq_p)) != NOERR) return;
```

```
    /* Nothing from the message received is put into the DT ?          */
    /* If lin, amm, specialist and date should be stored in DT,        */
    /* it must be done here                                            */

    /* Take test array from received msg and return it to call. progr. */
    for (n = 0; n < TEST_LENGTH_ACQ; n++) {
            *(value+n) = (EqmInt) *(test_p+n);
    }

    return;
}



/* Function : w___tst1 ************************************************/
proco_head(w, tst1, test_dtr, EqmInt)
{
 ctrl_msg ctrlmsg;
 int n;
 ctrl_msg *ctrl_p;
 ctrl_msg *dt_ctrl_p;
 vac_byte *test_p;

    if ((*coco = check_eq_type(dtr->trm[EQ_TYPE])) != NOERR) return;

    ctrl_p = &ctrlmsg;  /* NO new values are written into the DT !     */
    test_p = (vac_byte *) &(ctrl_p->u.test);

    /* Put test array in message */
    for(n = 0; n < TEST_LENGTH_CTRL; n++) {
            check_size(*(value+n), BYTE_MIN, BYTE_MAX);
            *(test_p+n) = (vac_byte) *(value+n);
    }

    set_lin;
    ctrl_p->amm = NO_RET;       /* Don't send acq. msg. back           */
            /* Note : Specialist field must be set before this         */
            /* property is called, to indicate test msg.               */
            /* Why can't the amm field be used for that ?              */

    /* The specialist field from DT must be send with message          */
    dt_ctrl_p = (ctrl_msg *) &(dtr->ctrlmsg);
    ctrl_p->specialist = dt_ctrl_p->specialist;

    ctrlq_setting.size = sizeof(ctrl_msg);/* Normal ctrl msg. length   */

    set_time(&(ctrl_p->date));
    if ((*coco = send_async(ctrl_p)) != NOERR) return;  /* Send msg.   */

    return;
}
```

3.1.5 rt.c

```
/*****************************************************************************/
/* CERN/PS/CO                                                                */
/* Date : 8/7 1992                                                           */
/* By   : Anker Rosenstedt                                                   */
/* File : rt.c                                                               */
/* Uses : To be linked with mqlib.c                                          */
/* Status : Final                                                            */
/* Descr. : The real time (specific) process for testing                     */
/*          Receives msg's from ctrlq and passing msg's on acqq              */
/*          Only used for testing queues and property functions              */
/*****************************************************************************/
#include <gm_constants.h>
#include <gm_types.h>
#include "message.h"
#include "mqlib.c"
#include <time.h>
#include <errno.h>
#include <stdio.h>



setting_send acqq_setting;
setting_recv ctrlq_setting;

ctrl_msg     ctrl_msg_buf, *ctrl_msg_p = &ctrl_msg_buf;
acq_msg      acq_msg_buf,  *acq_msg_p =  &acq_msg_buf;
status_msg   stat_msg_buf, *stat_msg_p = &stat_msg_buf;
vac_byte     test[TEST_LENGTH_CTRL];    /* TEST_LENGTH_CTRL/ACQ, what- */
                                        /* ever is the longest         */

/* Variables for passing the last received control values */
/* into aqn values in acq msg. (for test) */
vac_byte ccsact = 0;
mul_type ccv;
vac_real ccv1 = 0;



#define NO_SPEC 0       /* No special action (specialist field)        */
#include "msg_print.h"



/* Function : set_time ***************************************************/
void set_time(time_p)
vac_time *time_p;
{
 struct timeval timev, *time = &timev;
 struct timezone timez, *zone = &timez;         /* Not used            */

    gettimeofday(time, zone);
    time_p->sec = timev.tv_sec;
    time_p->usec = timev.tv_usec;
    return;
}



/* Function : set_acq_vals ***********************************************/
void set_acq_vals(ctrl_p, acq_p)        /* Read status from hardw. and */
ctrl_msg  *ctrl_p;                      /* put new data in acq msg.    */
acq_msg *acq_p;
{
    /* The header of the ctrl msg. is copied to the acq msg. (not date)*/
    acq_p->lin1 = ctrl_p->lin1;
```

```
    acq_p->lin2 = ctrl_p->lin2;
    acq_p->lin3 = ctrl_p->lin3;
    acq_p->amm = ctrl_p->amm;
    acq_p->specialist = ctrl_p->specialist;

    /* The acq values are also just copies of the ctrl values          */
    /* received with the last control message (test)                   */
    acq_p->saqn = ccsact;
    acq_p->u.data.aqn1 = ccv1;
    acq_p->u.data.aqn2 = ccv1;  /* Also set to ccv1 because no ccv2    */

    /* Special case for ccv to be copied into aqn */
    switch (ctrl_p->lin2 >> 8) {       /* Switch on EQ_TYPE            */
      case P_ION :
              acq_p->u.data.aqn.byte = ccv.byte;
              break;
      case P_SUBL :
      case G_ION :
              acq_p->u.data.aqn.int32 = ccv.int32;
              break;
      case G_PIRAN :
      case P_STAT :
              /* No valid ccv value -> a random (real) number is used  */
              acq_p->u.data.aqn.real = 99.99;
              break;
      default :          /* ccv and aqn fields not used                */
              break;

    }

    acq_p->phys_status = ccsact + 10; /* In lack of better test values */
    acq_p->aspect = ccsact + 100;
    acq_p->qualif = (vac_byte) ccv1;
    acq_p->busy_time = (vac_int32) ccsact + 200;

    /* Set date NOW, right before message is send (should be meas time)*/
    set_time(&(acq_p->date));

    return;
}



/* Function : set_stat_vals ***********************************************/
void set_stat_vals(ctrl_p, stat_p)      /* Read status from hardware   */
ctrl_msg *ctrl_p;                       /* and put data in status msg. */
status_msg *stat_p;
{
    /* The header of the ctrl msg. is copied to the acq msg. (not date)*/
    stat_p->lin1 = ctrl_p->lin1;
    stat_p->lin2 = ctrl_p->lin2;
    stat_p->lin3 = ctrl_p->lin3;
    stat_p->amm = ctrl_p->amm;
    stat_p->specialist = ctrl_p->specialist;

    /* The lists is assigned some accidental test data */
    stat_p->warnings.list = 10;
    stat_p->rfaults.list = 20;
    stat_p->ufaults.list = 30;
    stat_p->interlocks.list = 40;

    /* The dates are all assigned the time now (or now) */
    set_time(&(stat_p->warnings.date_last));
    set_time(&(stat_p->warnings.date_imp));

    set_time(&(stat_p->rfaults.date_last));
    set_time(&(stat_p->rfaults.date_imp));

    set_time(&(stat_p->ufaults.date_last));
```

```
    set_time(&(stat_p->ufaults.date_imp));

    set_time(&(stat_p->interlocks.date_last));
    set_time(&(stat_p->interlocks.date_imp));

    set_time(&(stat_p->date));

    return;
}


/* Function : send_norm_acq ***********************************************/
void send_norm_acq()
{
 int err;

    set_acq_vals(ctrl_msg_p, acq_msg_p);
    printf("\nSending message\n");
    /* Send same type as just received */
    acqq_setting.type = ctrlq_setting.type_recv;
    print_acq_msg(acqq_setting.type, acq_msg_p);

    /* Normal acq msg. to be send */
    acqq_setting.size = sizeof(acq_msg);
    if ((err = msg_send(&acqq_setting, acq_msg_p)) != 0) {
            printf("Error in sending message : %d\n", err);
            exit(1);
    }

    return;
}


/* Function : send_stat_acq ***********************************************/
void send_stat_acq()
{
 int err;

    set_stat_vals(ctrl_msg_p, stat_msg_p);
    printf("\nSending message\n");
    /* Send same type as just received */
    acqq_setting.type = ctrlq_setting.type_recv;
    print_stat_msg(acqq_setting.type, stat_msg_p);

    /* Status msg. to be send */
    acqq_setting.size = sizeof(status_msg);
    if ((err = msg_send(&acqq_setting, stat_msg_p)) != 0) {
            printf("Error in sending message : %d\n", err);
            exit(1);
    }

    return;
}


/* Function : send_test_acq ***********************************************/
void send_test_acq()
{
 int err;
 vac_byte *test_p;
 int n;

    /* The header of the ctrl msg. is copied to the acq msg. (not date)*/
    acq_msg_p->lin1 = ctrl_msg_p->lin1;
    acq_msg_p->lin2 = ctrl_msg_p->lin2;
    acq_msg_p->lin3 = ctrl_msg_p->lin3;
```

```
    acq_msg_p->amm = ctrl_msg_p->amm;
    acq_msg_p->specialist = ctrl_msg_p->specialist;

    /* Put last received test data in message to be send */
    test_p = (vac_byte *) &(acq_msg_p->u.test);
    for(n = 0; n < TEST_LENGTH_ACQ; n++) {
            *(test_p+n)  = test[n];
    }

    set_time(&(acq_msg_p->date));

    printf("\nSending message\n");
    /* Send same type as just received */
    acqq_setting.type = ctrlq_setting.type_recv;
    print_acq_msg(acqq_setting.type, acq_msg_p);

    /* Normal (test) acq msg. to be send */
    acqq_setting.size = sizeof(acq_msg);
    if ((err = msg_send(&acqq_setting, acq_msg_p)) != 0) {
            printf("Error in sending message : %d\n", err);
            exit(1);
    }

    return;
}


/* Main ***************************************************************/
main()
{
 unsigned long int msg_no = 1;
 vac_byte *test_p;
 int err;
 int n;

    printf("Setting up\n");
    if ((err = setup_send(&acqq_setting, acqq_name)) != 0) {
            printf("Error in opening '%s' queue : %d\n", acqq_name, err);
            exit(1);
    }

    if ((err = setup_recv(&ctrlq_setting, ctrlq_name)) != 0) {
            printf("Error in opening '%s' queue : %d\n", ctrlq_name, err);
            exit(1);
    }
    ctrlq_setting.size = sizeof(ctrl_msg);/* Largest msg to be received*/
                                          /* (request msg. is smaller) */
    ctrlq_setting.timeout = 0;  /* No timeout in RT process           */
    ctrlq_setting.type = 0;     /* Receive all types of msg's,        */
                                /* oldest first                       */


    for(;; msg_no++) {
            printf("\nWaiting to receive a message\n");
            err = msg_recv(&ctrlq_setting, ctrl_msg_p);

            switch (err) {

              case 0 :              /* Reception OK                   */
                  printf("(%d) Message received\n", msg_no);
                  print_ctrl_msg(ctrlq_setting.type_recv, ctrl_msg_p);

                  /* STORE VALUES FROM CONTROL MESSAGE */

                  if (ctrl_msg_p->amm == NO_RET) {  /* Async ctrl msg. */

                          if (ctrl_msg_p->specialist == NO_SPEC) {
                                                    /* NOT TEST */
```

```
                        /* Normal ctrl msg. : Store control values */
                        ccsact = ctrl_msg_p->u.data.ccsact;
                        ccv1 = ctrl_msg_p->u.data.ccv1;

                        switch (ctrl_msg_p->lin2 >> 8) {
                              case P_ION :
                                  ccv.byte = ctrl_msg_p->u.data.ccv.byte;
                                  break;
                              case P_SUBL :
                              case G_ION :
                                  ccv.int32 = ctrl_msg_p->u.data.ccv.int32;
                                  break;
                              default :
                                  break;
                        }
                    }
                    else {                                    /* TEST MSG */

                        /* Test msg. : Store test array */
                        test_p = (vac_byte *) &(ctrl_msg_p->u.test);
                        for (n = 0; n < TEST_LENGTH_CTRL; n++) {
                                test[n] = *(test_p+n);
                        }
                    }

                }

                /* SEND MESSAGE (normal, status, test or nothing) */

                switch(ctrl_msg_p->amm) {
                      case RET_ACQ :
                          send_norm_acq();
                          break;
                      case RET_STAT :
                          send_stat_acq();
                          break;
                      case RET_TEST :
                          send_test_acq();
                          break;
                      default :
                          break;
                }
                break;

            case EINTR :          /* Timeout                              */
                printf("Timeout\n");
                break;

            default :
                printf("Error in receiving message : %d\n", err);
                exit(1);
                break;
        }
    }
}
```

3.1.6 msg_print.h

```
/*****************************************************************************/
/* CERN/PS/CO                                                                */
/* Date : 8/7 1992                                                           */
/* By   : Anker Rosenstedt                                                   */
/* File : msg_print.h                                                        */
/* Uses :                                                                    */
/* Status : Updated along with message.h (and rt.c)                          */
/* Descr. : 3 functions that print the contents of a control message,        */
/*          acquisition message or status message.                           */
/*                                                                           */
/*****************************************************************************/



/* Function : print_ctrl_msg ************************************************/
void print_ctrl_msg(msg_type, ctrl_p)
int msg_type;
ctrl_msg *ctrl_p;
{
 char *time_p;
 vac_byte *test_p;
 int n;

    if (ctrl_p->amm == NO_RET) printf("CONTROL message :\n");
    else printf("REQUEST message : \n");

    printf("Type of message : %d\n", msg_type);
    printf("(Size of a normal ctrl message : %d)\n", sizeof(ctrl_msg));
    printf("(Size of a request ctrl msg.   : %d)\n", sizeof(req_msg));

    printf("lin1 (Eq mod no) : %d\n", ctrl_p->lin1);
    printf("lin2 : %d\n", ctrl_p->lin2);
    printf("Extracted from lin2 :\n");
    printf("  EQ_TYPE    : %d\n", ctrl_p->lin2 >> 8);
    printf("  EQ_STYPE   : %d\n", ctrl_p->lin2 & 255);
    printf("lin3 (Phys Eq No) : %d\n", ctrl_p->lin3);
    printf("amm : %d\n", ctrl_p->amm);

    /* Convert date to printable format */
    time_p = ctime(&(ctrl_p->date.sec));
    printf("date : (%d) %s", ctrl_p->date.sec, time_p);
    printf("  usec : %d\n", ctrl_p->date.usec);

    printf("specialist : %d\n", ctrl_p->specialist);

    if (ctrl_p->amm != NO_RET) return;  /* Req msg, do not print rest  */
    if (ctrl_p->specialist != NO_SPEC) {        /* Test msg            */
            printf("Test data : \n");
            test_p = (vac_byte *) &(ctrl_p->u.test);
            for(n = 0; n < TEST_LENGTH_CTRL; n++) printf("%d ", *(test_p+n));
            printf("\n");
            return;
    }

    printf("ccsact_chng : %d\n", ctrl_p->u.data.ccsact_chng);
    printf("ccsact : %d\n", ctrl_p->u.data.ccsact);

    /* Print ccv ... ccv1 depending on EQ_TYPE */
    switch (ctrl_p->lin2 >> 8) {       /* Switch on EQ_TYPE            */
      case P_ION :
              printf("ccv_chng : %d\n", ctrl_p->u.data.ccv_chng);
              printf("ccv (byte) : %d\n", ctrl_p->u.data.ccv.byte);
              printf("ccv1_chng : %d\n", ctrl_p->u.data.ccv1_chng);
              printf("ccv1 : %f\n", ctrl_p->u.data.ccv1);
              break;
```

```
      case P_SUBL :
      case G_ION :
              printf("ccv_chng : %d\n", ctrl_p->u.data.ccv_chng);
              printf("ccv (int32) : %d\n", ctrl_p->u.data.ccv.int32);
              printf("ccv1_chng : %d\n", ctrl_p->u.data.ccv1_chng);
              printf("ccv1 : %f\n", ctrl_p->u.data.ccv1);
              break;
      default :
              printf("no ccv/ccv1\n");
              break;
    }

    return;
}



/* Function : print_acq_msg ***********************************************/
void print_acq_msg(msg_type, acq_p)
int msg_type;
acq_msg *acq_p;
{
 char *time_p;
 vac_byte *test_p;
 int n;

    printf("ACQUISITION message :\n");
    printf("Type of message : %d\n", msg_type);
    printf("(Size of normal acq message : %d)\n", sizeof(acq_msg));

    printf("lin1 (Eq mod no) : %d\n", acq_p->lin1);
    printf("lin2 : %d\n", acq_p->lin2);
    printf("Extracted from lin2 :\n");
    printf("  EQ_TYPE    : %d\n", acq_p->lin2 >> 8);
    printf("  EQ_STYPE   : %d\n", acq_p->lin2 & 255);
    printf("lin3 (Phys Eq No) : %d\n", acq_p->lin3);
    printf("amm : %d\n", acq_p->amm);

    /* Convert date to printable format */
    time_p = ctime(&(acq_p->date.sec));
    printf("date : (%d) %s", acq_p->date.sec, time_p);
    printf("  usec : %d\n", acq_p->date.usec);

    printf("specialist : %d\n", acq_p->specialist);

    printf("phys status : %d\n", acq_p->phys_status);
    printf("saqn : %d\n", acq_p->saqn);
    printf("aspect : %d\n", acq_p->aspect);
    printf("qualif : %d\n", acq_p->qualif);
    printf("busy_time : %d\n", acq_p->busy_time);

    if (acq_p->specialist != NO_SPEC) {        /* Test msg.            */
            printf("Test data : \n");
            test_p = (vac_byte *) &(acq_p->u.test);
            for(n = 0; n < TEST_LENGTH_ACQ; n++) printf("%d ", *(test_p+n));
            printf("\n");
            return;
    }

    /* Print aqn, aqn1 & aqn2 depending on EQ_TYPE */
    switch (acq_p->lin2 >> 8) {
      case P_ION :
              printf("aqn (byte) : %d\n", acq_p->u.data.aqn.byte);
              printf("aqn1 : %f\n", acq_p->u.data.aqn1);
              printf("aqn2 : %f\n", acq_p->u.data.aqn2);
              break;
      case P_SUBL :
      case G_ION :
              printf("aqn (int32) : %d\n", acq_p->u.data.aqn.int32);
```

```
            printf("aqn1 : %f\n", acq_p->u.data.aqn1);
            printf("aqn2 : %f\n", acq_p->u.data.aqn2);
            break;
      case G_PIRAN :
      case P_STAT :
            printf("aqn (real) : %f\n", acq_p->u.data.aqn.real);
            printf("no aqn1/aqn2\n");
            break;
      default :
            printf("no aqn/aqn1/aqn2\n");
            break;
   }

   return;
}



/* Function : print_stat_msg ***********************************************/
void print_stat_msg(msg_type, stat_p)
int msg_type;
status_msg *stat_p;
{
 char *time_p;

   printf("STATUS message :\n");
   printf("Type of message : %d\n", msg_type);
   printf("(Size of status message : %d)\n", sizeof(status_msg));

   printf("lin1 (Eq mod no) : %d\n", stat_p->lin1);
   printf("lin2 : %d\n", stat_p->lin2);
   printf("Extracted from lin2 :\n");
   printf("  EQ_TYPE   : %d\n", stat_p->lin2 >> 8);
   printf("  EQ_STYPE   : %d\n", stat_p->lin2 & 255);
   printf("lin3 (Phys Eq No) : %d\n", stat_p->lin3);
   printf("amm : %d\n", stat_p->amm);

   /* Convert date to printable format */
   time_p = ctime(&(stat_p->date.sec));
   printf("date : (%d) %s", stat_p->date.sec, time_p);
   printf("  usec : %d\n", stat_p->date.usec);

   printf("specialist : %d\n", stat_p->specialist);

   printf("Warnings :\n");
   printf("  list      : %d\n", stat_p->warnings.list);
   printf("  date_last : %d\n", stat_p->warnings.date_last.sec);
   printf("    usec    : %d\n", stat_p->warnings.date_last.usec);
   printf("  date_imp  : %d\n", stat_p->warnings.date_imp.sec);
   printf("    usec    : %d\n", stat_p->warnings.date_imp.usec);

   printf("Rfaults :\n");
   printf("  list      : %d\n", stat_p->rfaults.list);
   printf("  date_last : %d\n", stat_p->rfaults.date_last.sec);
   printf("    usec    : %d\n", stat_p->rfaults.date_last.usec);
   printf("  date_imp  : %d\n", stat_p->rfaults.date_imp.sec);
   printf("    usec    : %d\n", stat_p->rfaults.date_imp.usec);

   printf("Ufaults :\n");
   printf("  list      : %d\n", stat_p->ufaults.list);
   printf("  date_last : %d\n", stat_p->ufaults.date_last.sec);
   printf("    usec    : %d\n", stat_p->ufaults.date_last.usec);
   printf("  date_imp  : %d\n", stat_p->ufaults.date_imp.sec);
   printf("    usec    : %d\n", stat_p->ufaults.date_imp.usec);

   printf("Interlocks :\n");
   printf("  list      : %d\n", stat_p->interlocks.list);
   printf("  date_last : %d\n", stat_p->interlocks.date_last.sec);
   printf("    usec    : %d\n", stat_p->interlocks.date_last.usec);
```

```
    printf("  date_imp   : %d\n", stat_p->interlocks.date_imp.sec);
    printf("    usec     : %d\n", stat_p->interlocks.date_imp.usec);

    return;
}
```

3.1.7 Test1 program with printout & messages

3.1.7.1 Test1.nod program

```
1.10 % File : TEST1.NOD
1.20 % Test of CCSACT, CCV, CCV1, SAQN, AQN, AQN1, AQN2,
1.30 % PHSTAT, ASPECT, DATE & BUSY property functions
1.40 SE EQ = 20003
1.50 TY "EQUIPMENT NO : " EQ !!

2.10 SE C = 0
2.20 TY "SET CCV1 = 32" !
2.30 SE VPUMP(EQ,CCV1,0,C) = 32
2.40 TY "COCO = " C !!

3.10 SE C = 0
3.20 TY "SET CCSACT = 1" !
3.30 SE VPUMP(EQ,CCSACT,0,C) = 1
3.40 TY "COCO = " C !!

4.10 SE C = 0
4.20 TY "SET CCV = 2" !
4.30 SE VPUMP(EQ,CCV,0,C) = 2
4.40 TY "COCO = " C !!

5.10 SE C = 0
5.20 TY "READ STAQ"
5.30 TY VPUMP(EQ,STAQ,0,C) !
5.40 TY "COCO = " C !!

6.10 SE C = 0
6.20 TY "SET CCSACT = 2" !
6.30 SE VPUMP(EQ,CCSACT,0,C) = 2
6.40 TY "COCO = " C !!

7.10 SE C = 0
7.20 TY "READ STAQ"
7.30 TY VPUMP(EQ,STAQ,0,C) !
7.40 TY "COCO = " C !!

8.10 SE C = 0
8.20 TY "SET CCSACT = 3" !
8.30 SE VPUMP(EQ,CCSACT,0,C) = 3
8.40 TY "COCO = " C !!

9.10 SE C = 0
9.20 TY "SET CCSACT = 4" !
9.30 SE VPUMP(EQ,CCSACT,0,C) = 4
9.40 TY "COCO = " C !!

10.10 SE C = 0
10.20 TY "SET CCSACT = 0" !
10.30 SE VPUMP(EQ,CCSACT,0,C) = 0
10.40 TY "COCO = " C !!

11.10 SE C = 0
11.20 TY "READ AQN"
11.30 TY VPUMP(EQ,AQN,0,C) !
11.40 TY "COCO = " C !!

12.10 SE C = 0
12.20 TY "READ AQN1"
12.30 TY VPUMP(EQ,AQN1,0,C) !
12.40 TY "COCO = " C !!
```

```
 13.10 SE C = 0
 13.20 TY "READ AQN2"
 13.30 TY VPUMP(EQ,AQN2,0,C) !
 13.40 TY "COCO = " C !!

 14.10 SE C = 0
 14.20 TY "SET CCV = 1.5" !
 14.30 SE VPUMP(EQ,CCV,0,C) = 1.5
 14.40 TY "COCO = " C !!

 15.10 SE C = 0
 15.20 TY "READ PHSTAT"
 15.30 TY VPUMP(EQ,PHSTAT,0,C) !
 15.40 TY "COCO = " C !!

 16.10 SE C = 0
 16.20 TY "READ ASPECT"
 16.30 TY VPUMP(EQ,ASPECT,0,C) !
 16.40 TY "COCO = " C !!

 17.10 SE C = 0
 17.20 DIM-L D(2)
 17.30 TY "READ DATE" !
 17.40 VPUMP(EQ,DATE,0,C,D)
 17.50 TY "sec.  : " D(1) !
 17.60 TY "usec. : " D(2) !
 17.70 TY "COCO = " C !!

 18.10 SE C = 0
 18.20 TY "READ BUSY"
 18.30 TY VPUMP(EQ,BUSY,0,C) !
 18.40 TY "COCO = " C !!

% program file name : test1.nod
```

3.1.7.2 Printout from test1.nod

```
EQUIPMENT NO :  20003

SET CCV1 = 32
COCO =      0

SET CCSACT = 1
COCO =      0

SET CCV = 2
COCO =      0

READ STAQ     1
COCO =      0

SET CCSACT = 2
COCO =      0

READ STAQ     2
COCO =      0

SET CCSACT = 3
COCO =    180

SET CCSACT = 4
COCO =    180
```

```
SET CCSACT = 0
COCO =    180

READ AQN     2
COCO =       0

READ AQN1    32
COCO =       0

READ AQN2    32
COCO =       0

SET CCV = 1.5
COCO =    180

READ PHSTAT    12
COCO =       0

READ ASPECT   102
COCO =       0

READ DATE
sec.  : 708874974
usec. : 960000
COCO =       0

READ BUSY   202
COCO =       0
```

3.1.7.3 Message contents from test1.nod

```
Setting up

Waiting to receive a message
(1) Message received
CONTROL message :
Type of message : 17
(Size of a normal ctrl message : 62)
(Size of a request ctrl msg.   : 22)
lin1 (Eq mod no) : 230
lin2 : 798
Extracted from lin2 :
  EQ_TYPE   : 3
  EQ_STYPE   : 30
lin3 (Phys Eq No) : 33
amm : 5
date : (708874972) Thu Jun 18 14:42:52 1992
  usec : 460000
specialist : 0
ccsact_chng : 1
ccsact : 1
ccv_chng : -1
ccv (int32) : 2
ccv1_chng : 1
ccv1 : 32.000000

Waiting to receive a message
(2) Message received
REQUEST message :
Type of message : 17
(Size of a normal ctrl message : 62)
(Size of a request ctrl msg.   : 22)
lin1 (Eq mod no) : 230
lin2 : 798
```

```
Extracted from lin2 :
  EQ_TYPE    : 3
  EQ_STYPE   : 30
lin3 (Phys Eq No) : 33
amm : 0
date : (708874972) Thu Jun 18 14:42:52 1992
  usec : 480000
specialist : 0

Sending message
ACQUISITION message :
Type of message : 17
(Size of normal acq message : 70)
lin1 (Eq mod no) : 230
lin2 : 798
Extracted from lin2 :
  EQ_TYPE    : 3
  EQ_STYPE   : 30
lin3 (Phys Eq No) : 33
amm : 0
date : (708874974) Thu Jun 18 14:42:54 1992
  usec : 690000
specialist : 0
phys status : 11
saqn : 1
aspect : 101
qualif : 32
busy_time : 201
aqn (int32) : 2
aqn1 : 32.000000
aqn2 : 32.000000

Waiting to receive a message
(3) Message received
CONTROL message :
Type of message : 17
(Size of a normal ctrl message : 62)
(Size of a request ctrl msg.   : 22)
lin1 (Eq mod no) : 230
lin2 : 798
Extracted from lin2 :
  EQ_TYPE    : 3
  EQ_STYPE   : 30
lin3 (Phys Eq No) : 33
amm : 5
date : (708874974) Thu Jun 18 14:42:54 1992
  usec : 710000
specialist : 0
ccsact_chng : 1
ccsact : 2
ccv_chng : 1
ccv (int32) : 2
ccv1_chng : -1
ccv1 : 32.000000

Waiting to receive a message
(4) Message received
REQUEST message :
Type of message : 17
(Size of a normal ctrl message : 62)
(Size of a request ctrl msg.   : 22)
lin1 (Eq mod no) : 230
lin2 : 798
Extracted from lin2 :
  EQ_TYPE    : 3
  EQ_STYPE   : 30
lin3 (Phys Eq No) : 33
amm : 0
date : (708874974) Thu Jun 18 14:42:54 1992
```

```
  usec : 720000
specialist : 0

Sending message
ACQUISITION message :
Type of message : 17
(Size of normal acq message : 70)
lin1 (Eq mod no) : 230
lin2 : 798
Extracted from lin2 :
  EQ_TYPE   : 3
  EQ_STYPE   : 30
lin3 (Phys Eq No) : 33
amm : 0
date : (708874974) Thu Jun 18 14:42:54 1992
  usec : 750000
specialist : 0
phys status : 12
saqn : 2
aspect : 102
qualif : 32
busy_time : 202
aqn (int32) : 2
aqn1 : 32.000000
aqn2 : 32.000000

Waiting to receive a message
(5) Message received
REQUEST message :
Type of message : 17
(Size of a normal ctrl message : 62)
(Size of a request ctrl msg.   : 22)
lin1 (Eq mod no) : 230
lin2 : 798
Extracted from lin2 :
  EQ_TYPE   : 3
  EQ_STYPE   : 30
lin3 (Phys Eq No) : 33
amm : 0
date : (708874974) Thu Jun 18 14:42:54 1992
  usec : 800000
specialist : 0

Sending message
ACQUISITION message :
Type of message : 17
(Size of normal acq message : 70)
lin1 (Eq mod no) : 230
lin2 : 798
Extracted from lin2 :
  EQ_TYPE   : 3
  EQ_STYPE   : 30
lin3 (Phys Eq No) : 33
amm : 0
date : (708874974) Thu Jun 18 14:42:54 1992
  usec : 860000
specialist : 0
phys status : 12
saqn : 2
aspect : 102
qualif : 32
busy_time : 202
aqn (int32) : 2
aqn1 : 32.000000
aqn2 : 32.000000

Waiting to receive a message
(6) Message received
REQUEST message :
```

```
Type of message : 17
(Size of a normal ctrl message : 62)
(Size of a request ctrl msg.   : 22)
lin1 (Eq mod no) : 230
lin2 : 798
Extracted from lin2 :
  EQ_TYPE   : 3
  EQ_STYPE   : 30
lin3 (Phys Eq No) : 33
amm : 0
date : (708874974) Thu Jun 18 14:42:54 1992
  usec : 890000
specialist : 0

Sending message
ACQUISITION message :
Type of message : 17
(Size of normal acq message : 70)
lin1 (Eq mod no) : 230
lin2 : 798
Extracted from lin2 :
  EQ_TYPE   : 3
  EQ_STYPE   : 30
lin3 (Phys Eq No) : 33
amm : 0
date : (708874974) Thu Jun 18 14:42:54 1992
  usec : 900000
specialist : 0
phys status : 12
saqn : 2
aspect : 102
qualif : 32
busy_time : 202
aqn (int32) : 2
aqn1 : 32.000000
aqn2 : 32.000000

Waiting to receive a message
(7) Message received
REQUEST message :
Type of message : 17
(Size of a normal ctrl message : 62)
(Size of a request ctrl msg.   : 22)
lin1 (Eq mod no) : 230
lin2 : 798
Extracted from lin2 :
  EQ_TYPE   : 3
  EQ_STYPE   : 30
lin3 (Phys Eq No) : 33
amm : 0
date : (708874974) Thu Jun 18 14:42:54 1992
  usec : 940000
specialist : 0

Sending message
ACQUISITION message :
Type of message : 17
(Size of normal acq message : 70)
lin1 (Eq mod no) : 230
lin2 : 798
Extracted from lin2 :
  EQ_TYPE   : 3
  EQ_STYPE   : 30
lin3 (Phys Eq No) : 33
amm : 0
date : (708874974) Thu Jun 18 14:42:54 1992
  usec : 960000
specialist : 0
phys status : 12
```

```
saqn : 2
aspect : 102
qualif : 32
busy_time : 202
aqn (int32) : 2
aqn1 : 32.000000
aqn2 : 32.000000

Waiting to receive a message

Process killed
```

3.1.8 Test2 program with printout

3.1.8.1 Test2.nod program

```
1.10 % File : TEST2.NOD
1.20 % Test of WARN, RFAULT, UFAULT, INTLK,
1.30 % WARNM, UFAULM, RFAULM & INTLKM property functions
1.40 SE EQ = 20003
1.50 TY "EQUIPMENT NO : " EQ !!
1.60 DIM-L D(4)

2.10 SE C = 0
2.20 TY "READ WARN list" !
2.30 TY VPUMP(EQ,WARN,0,C) !
2.40 TY "COCO = " C !!

3.10 SE C = 0
3.20 TY "READ WARN dates" !
3.30 VPUMP(EQ,WARNM,0,C,D)
3.40 TY "last :   sec  : " D(1) !
3.50 TY "         usec : " D(2) !
3.60 TY "import : sec  : " D(3) !
3.70 TY "         usec : " D(4) !
3.80 TY "COCO = " C !!

4.10 SE C = 0
4.20 TY "READ RFAULT list" !
4.30 TY VPUMP(EQ,RFAULT,0,C) !
4.40 TY "COCO = " C !!

5.10 SE C = 0
5.20 TY "READ RFAULT dates" !
5.30 VPUMP(EQ,RFAULM,0,C,D)
5.40 TY "last :   sec  : " D(1) !
5.50 TY "         usec : " D(2) !
5.60 TY "import : sec  : " D(3) !
5.70 TY "         usec : " D(4) !
5.80 TY "COCO = " C !!

6.10 SE C = 0
6.20 TY "READ UFAULT list" !
6.30 TY VPUMP(EQ,UFAULT,0,C) !
6.40 TY "COCO = " C !!

7.10 SE C = 0
7.20 TY "READ UFAULT dates" !
7.30 VPUMP(EQ,UFAULM,0,C,D)
7.40 TY "last :   sec  : " D(1) !
7.50 TY "         usec : " D(2) !
7.60 TY "import : sec  : " D(3) !
7.70 TY "         usec : " D(4) !
7.80 TY "COCO = " C !!

8.10 SE C = 0
8.20 TY "READ INTLK list" !
8.30 TY VPUMP(EQ,INTLK,0,C) !
8.40 TY "COCO = " C !!

9.10 SE C = 0
9.20 TY "READ INTLK dates" !
9.30 VPUMP(EQ,INTLKM,0,C,D)
9.40 TY "last :   sec  : " D(1) !
9.50 TY "         usec : " D(2) !
9.60 TY "import : sec  : " D(3) !
9.70 TY "         usec : " D(4) !
9.80 TY "COCO = " C !!

% program file name : test2.nod
```

3.1.8.2 Printout from test2.nod

```
EQUIPMENT NO :  20003

READ WARN list
    10
COCO =      0

READ WARN dates
last :   sec  : 708875406
             usec : 720000
import : sec  : 708875406
             usec : 720000
COCO =      0

READ RFAULT list
    20
COCO =      0

READ RFAULT dates
last :   sec  : 708875406
             usec : 960000
import : sec  : 708875406
             usec : 960000
COCO =      0

READ UFAULT list
    30
COCO =      0

READ UFAULT dates
last :   sec  : 708875407
             usec : 230000
import : sec  : 708875407
             usec : 230000
COCO =      0

READ INTLK list
    40
COCO =      0

READ INTLK dates
last :   sec  : 708875407
             usec : 480000
import : sec  : 708875407
             usec : 480000
COCO =      0
```

3.1.9 Test3 program with printout

3.1.9.1 Test3.nod program

```
  1.10 % File : TEST3.NOD
  1.20 % Test of TBIT(r/w) and TEST1(r/w) property functions
  1.30 SE EQ = 20003
  1.40 TY "EQUIPMENT NO : " EQ !!
  1.50 DIM-L T(40)

  2.10 SE C = 0
  2.20 TY "SET TBIT = 111" !
  2.30 SE VPUMP(EQ,TBIT,0,C) = 111
  2.40 TY "COCO = " C !!

  3.10 SE C = 0
  3.20 TY "READ STAQ"
  3.30 TY VPUMP(EQ,STAQ,0,C) !
  3.40 TY "COCO = " C !!

  4.10 SE C = 0
  4.20 TY "READ TBIT" !
  4.30 TY VPUMP(EQ,TBIT,0,C) !
  4.40 TY "COCO = " C !!

  5.10 SE C = 0
  5.20 TY "SEND TEST1 DATA" !
  5.30 FOR N=1, 40; SE T(N) = N*6
  5.40 VPUMP(EQ,TEST1,0,C,T,-1)
  5.50 TY "COCO = " C !!

  6.10 SE C = 0
  6.20 TY "RECEIVE TEST1 DATA" !
  6.30 VPUMP(EQ,TEST1,0,C,T)
  6.40 FOR N=1, 40; TY T(N)
  6.50 TY ! "COCO = " C !!

% program file name : test3.nod
```

3.1.9.2 Printout from test3.nod

```
EQUIPMENT NO :  20003

SET TBIT = 111
COCO =      0

READ STAQ     0
COCO =      0

READ TBIT
   111
COCO =      0

SEND TEST1 DATA
COCO =      0

RECEIVE TEST1 DATA
     6         12         18         24         30         36         42
    48         54         60         66         72         78         84
    90         96        102        108        114        120        126
   132        138        144        150        156        162        168
   174        180        186        192        198        204        210
   216        222        228        234        240
COCO =      0
```

3.1.10 Makenodal

```
# 8/7 1992
# Make file to build Nodal with local eqm on MVME147 LynxOS
 SUFFIXES : .o .c

CC=     gcc
PROJECT= nodal
SHELL=  /bin/sh
E2=/u/dscps/icv196fclty
E3=/u/dscps/fpiplsfclty
VAR = /u/rosenst/dsc/vac

CINCLUDE = -I/usr/local/include/tgm -I/usr/local/include/err \
               -I/u/app/dsc/include -I$(VAR)

# C compiler flags:
CCFLAGS = -o $*.o -c -g -DLYNX $(CINCLUDE)

# How to compile a .c program to produce a .o:
.c.o:
            @echo --\> Compile c program $*.c
            @$(CC) $(CCFLAGS) $*.c

LKFLAGS =  -g

# GM host library directory
GMLIB  = /u/app/dsc

# Property-code library directory
PROCOLIB  = /u/psco/dsc

PSLIB = /usr/local/lib

ODIR = /u/perrioll/dsc/nodal

LIBS =    $(PSLIB)/libfltcvt.a -lnetinet -lm \
              -lbsd /usr/local/lib/liberr_dummy.a

TLGFIL =  $(PSLIB)/libtgm.a \
              $(PSLIB)/liberr.a

CONNCT = $(PSLIB)/gpsynchrolib.o

OBJECTS= $(ODIR)/nodal.o \
             $(PROCOLIB)/emmess.o \
             $(PSLIB)/vmebuslib.o \
             $(E2)/icv196lib.o \
             $(E3)/fpiplslib.o \
             $(PSLIB)/camaclib.o \
             $(PSLIB)/ioconfig.o \
              $(CONNCT)  \
              $(PSLIB)/noduser_vmebase.o \
              $(PSLIB)/nodlist_vmebase.o


$(PROJECT):  $(VAR)/gm_pbt.o $(PROCOLIB)/procolib.a \
                   vac_vpump vac_vgaug vac_vvalv mqlib
             $(CC) $(LKFLAGS) $(OBJECTS) \
             $(VAR)/gm_pbt.o $(GMLIB)/gmlib.a $(PROCOLIB)/procolib.a \
             $(VAR)/vac_vpump.o $(VAR)/vac_vgaug.o $(VAR)/vac_vvalv.o \
             $(VAR)/mqlib.o \
             $(TLGFIL) $(LIBS) -o $(PROJECT)
```

```
vac_vpump: $(VAR)/vac_proco.c
	$(CC) -c -g -DLYNX -DVPUMP_PROCO \
	$(CINCLUDE) vac_proco.c -o vac_vpump.o

vac_vgaug: $(VAR)/vac_proco.c
	$(CC) -c -g -DLYNX -DVGAUG_PROCO \
	$(CINCLUDE) vac_proco.c -o vac_vgaug.o

vac_vvalv: $(VAR)/vac_proco.c
	$(CC) -c -g -DLYNX -DVVALV_PROCO \
	$(CINCLUDE) vac_proco.c -o vac_vvalv.o

mqlib:     $(VAR)/mqlib.c
	$(CC) -c -g -DLYNX -DMQLIB_CODE \
	$(CINCLUDE) mqlib.c -o mqlib.o
```

Distribution (of abstract)

G.P. Benincasa
F. Perriollat
C. Serre
SLM

AT/Vacuum Section