EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

CERN/PS/BD/Note 94-2

REAL-TIME PROGRAM FOR THE PS FFT Q-MEASUREMENT

S. Johnston

ABSTRACT

As part of the on-going controls conversion project, a real-time program has been developed for the PS FFT Q-measurement system that allows either PPM, non-PPM, or both types of operation. This real-time program, running under LynxOS, suspends PPM operation during non-PPM measurements, resuming it when non-PPM operation is completed.

A high-level description of the functionality of the software is given, which includes an explanation on the use of the Control Protocol for Instrumentation. This will be of interest to anyone who must develop software to control equipment from the DSC environment. A detailed break-down of the software is then given, which although primarily intended as a reference document for the Q-measurement, sheds more light on the operation of the real-time system.

Geneva, Switzerland
January 1994

CONTENTS

PRINCIPLE TERMS AND ABBREVIATIONS

The following is an explanation of the principle terms and abbreviations associated with writing software for the real-time control of equipment.

Application Program	A program with which a user enters control parameters and views measurements, usually running on a workstation in the accelerator control room.
Asynchronous Message Request	A message passed from an EM to a RTP which does not require an immediate response, e.g. the EM might request the RTP to perform a measurement upon a certain event, after which the RTP replies.
Device Stub Controller (DSC)	System consisting of a VME chassis, master-controller and possibly other VME modules. The master controller is a Motorola processor card (e.g. MVME147SA-1) running under the LynxOS operating system.
Equipment Module (EM)	Software which passes control messages to, and receives acquisition data from, a Real-Time Program. The EM runs on a DSC and communicates with the RTP through message queues.
Device Family	The name given to a set of devices (systems) performing a similar function, e.g. Family 72 for Q-measurement.
Message queues	A facility for queuing message structures in a FIFO basis, used as the interface between Real-Time Programs and Equipment Modules.
Non-PPM	A mode of operation whereby a system performs some function upon the arrival of a particular event and does not perform further actions until requested again.
Program Line Sequencer (PLS)	A system for timing used in the PS accelerator complex, consisting of a bit stream containing information such as user and cycle information.
Control Protocol	The fixed structure of messages passed to and from Real-Time Programs.
Pulse-to-Pulse Modulation (PPM)	A mode of operation whereby a system performs some function upon the arrival of a particular event, e.g. a user-line, and performs the same function upon every subsequent similar event without re-programming.
Real-Time Program (RTP)	Software, running in a DSC, which receives control messages from an Equipment Module. The RTP can store control messages or send immediate replies, as well as performing hardware/software controls, taking measurements and data-processing.
Synchronous Message Request	A message passed from an EM to a RTP which requires an immediate response, e.g. the EM might request some status information from the RTP.

1. INTRODUCTION

The PS FFT Q-measurement is a VME-based system which may be used to perform acquisitions, Fast Fourier Transforms (FFTs) or q-value interpolation on beam-position pick-up data. Its two main components are a Motorola MVME147 controller and a VASP-16 Digital Signal Processor (DSP) (Fig 1). The system also contains a general purpose I/O module for controls such as pick-up selection, a serial CAMAC interface to preset-counters and an FPIPLS receiver. The VASP-16 has its own interface to a NIM chassis in which is located a 10 MHz, 12-bit ADC and its interface.

The VASP-16 contains a master Texas TMS320C25 processor which controls the operation of the module, and four slave Zoran ZR34161 Vector Signal Processors capable of together performing a 1024-point magnitude squared FFT in 1.1 ms [1].

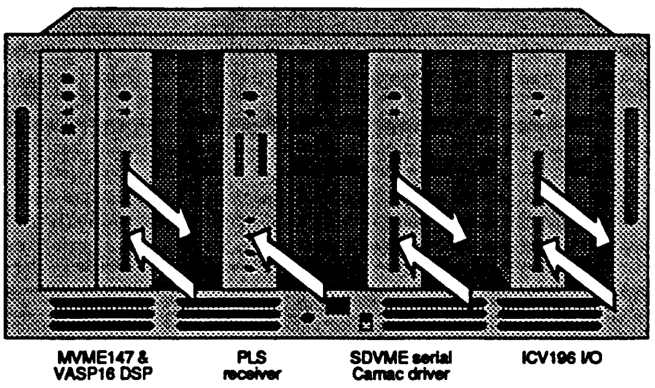


Fig 1. The VME-based PS FFT Q-measurement system. An MVME147 running under LynxOS is used to control the system, and Digital Signal Processing is performed by a VASP-16, coded in C and assembler.

The MVME147, also known as a Device Stub Controller (DSC), hosts the real-time task which manages all the system functions, such as programming and communicating with the VASP16, reception and interpretation of control messages, handling of PLS information, programming of hardware I/O, and the reception of measurement data followed by its communication to the next stage of the PS control system, the equipment module (EM).

The aim of this note is to present a detailed explanation of the real-time task. In chapter 2 the functionality of the software will be described, without details of the code itself, in order to allow the reader to familiarise their self with its operation. This includes an explanation of the use of the control protocol for instrumentation to define the format of communication messages passed between the real-time task and the equipment module. There then follows a detailed breakdown of all the components of the real-time task, with reference to the source code which is stored on a networked computer. Included within the explanation are descriptions of the use of several standard PS modules.

Specialised DSP programming of the VASP-16 nor Q-measurement theory will be discussed in this paper. For more information on these subjects the reader is kindly referred elsewhere [2], [3].

2. FUNCTIONALITY

2.1. Introduction

The majority of instrumentation systems in the accelerator environment are operated in Pulse to Pulse Modulation (PPM) whereby a system performs some operation upon the arrival of a particular event, e.g. a user line, and continues to perform the same operation upon every subsequent similar event without re-programming. For this reason a PPM model for a real-time task has evolved from the development which went into the conversion of the LPI transformers to LynxOS-based control [4]. There exists however a number of instrumentation systems which require non-PPM operation, meaning that an operation is to be performed upon the arrival of a certain event and should not be performed again until instructed to do so. Such a system is the flying wire scanner of the PS, and others systems even require both types of operation, e.g. the FFT Q-measurement systems of the PS and PSB machines. Using the original PPM model as a starting point, considerable effort has gone into the development of the real-time task to allow both PPM and non-PPM operation of the PS FFT Q-measurement. The resulting software is not however a model but is specifically designed for the Q-measurement.

The source code for the real-time task is to be found in a number of directories as illustrated in Fig 2. These files can be accessed by logging into any machine of the PS LynxOS control system. The original PPM-only code is located in the directory `/u/gelato/body` and its sub-directory `template`.

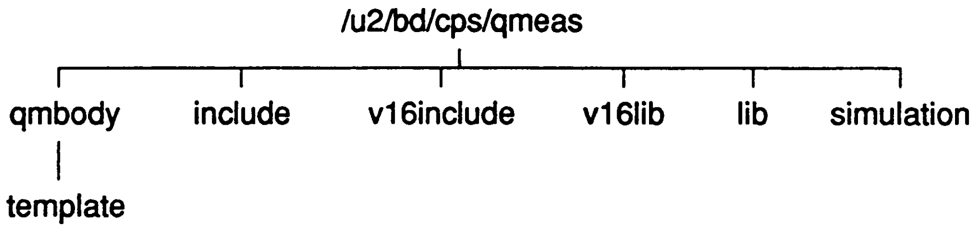


Fig 2. Directory structure containing the source code of the real-time task for the PS FFT Q-measurement system. `qmbody` and `template` contain the source for the real-time task, which uses files in `include` and `lib`. Directories `v16include` and `v16lib` are VASP-16 specific.

Someone wishing to use the operational Q-measurement system would access it by means of an application program[1] running on a workstation in the MCR. This program allows the user to enter commands and view results. The application program does not communicate with the real-time task directly however, but via an equipment module which is running on the same DSC as the real-time task. It is the duty of the equipment module to verify the validity of the request from the user before sending it to the real-time task, as well as receiving resultant messages. Communication between the two programs is via message queues which will be discussed in detail in section 2.3.

[1] Program still to be produced at time of writing.

2.2. Message protocol definition

As mentioned in section 2.1, communication between the real-time task and equipment module is facilitated using message queues (Fig 3).

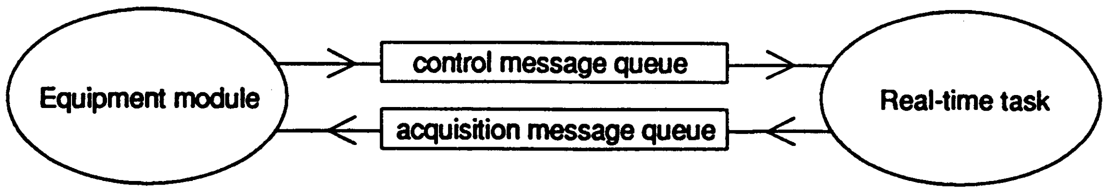


Fig. 3. Message queues are used as a communication medium between equipment modules and real-time tasks.

Normally two queues are used for communication; one for control messages coming from the equipment module, and one for acquisition messages sent by the real-time task. More than one message can exist in a queue at a time, and the reading process will receive the messages on a FIFO basis.

Such a scheme can only be used if the format of messages placed in the queues are the same for both the equipment module and the real-time task. This is achieved using the Control Protocol for Instrumentation [5].

The file `/u/psco/dsc/include/npro.h` defines the general format of control and acquisition messages as passed between equipment modules and real-time tasks. A control message consists of several units; a header, status indicator, timing values and equipment-specific control values. An acquisition message similarly consists of several units; a header, equipment status information and measurement values. Both message formats are fixed except for the number of control values allowed in a control message, and the number and type of measurement values allowed in an acquisition message. A second file `pro.h`, usually located in the working directory of the programmer, is used to define these last elements. Compilation of the two files together results in the complete definition of the message protocol (Fig 4).

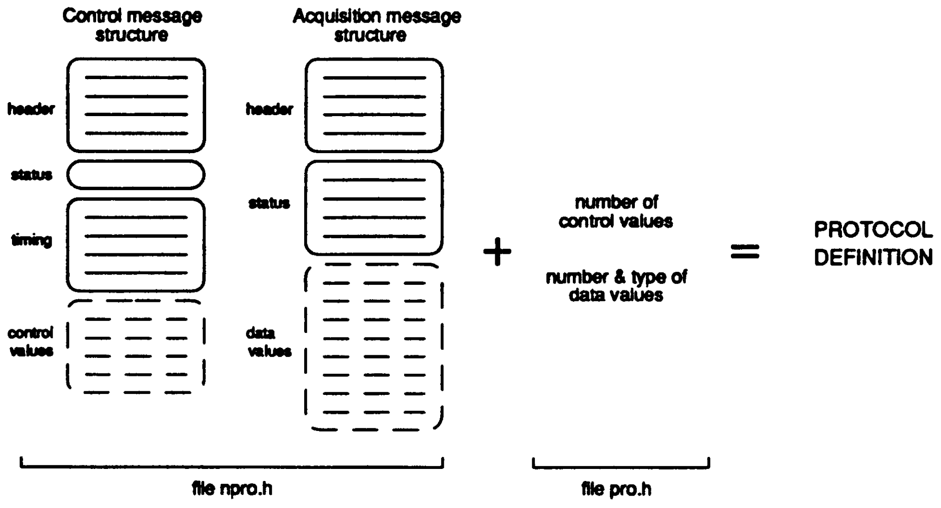


Fig. 4. The file `npro.h` defines the general format of control and acquisition messages. Compilation of `npro.h` and `pro.h`, which defines the number of control values and measurement values, results in the complete definition of the message protocol.

At this point some explanation is required as to why the PS FFT Q-measurement cannot use two message queues but instead requires three. The system must be capable of performing several hundred Q-measurements during a single PS machine cycle, and these must be passed to the control room fast enough to allow the system to operate in PPM. However, FFT analysis produces very large amounts of data (up to 10^5 words) and it is not possible to transfer such an amount of data to the control room on a PPM basis. Consequently, a single message definition for acquisition data cannot be used.

The solution has been to use a single control message for both types of measurement and two acquisition message definitions. One acquisition message queue is used for PPM messages, i.e. Q-values, and the other is for non-PPM messages, i.e. FFT data.

Every piece of equipment accessed in the control system has a family, or reference, number which uniquely identifies its function. This number is 72 for Q-measurement and 247 for FFT analysis. These numbers have been appended to the standard `npro.h` and `pro.h` files to differentiate between the two functions. The files `npro_72.h` and `pro_72.h` are used to define the control message structure for the PPM and non-PPM measurements, as well as the acquisition message structure for the Q-measurement. The files `npro_247.h` and `pro_247.h` correspondingly define the format of non-PPM acquisition messages. In this way the format of the standard `npro.h` file is retained exactly and the equipment module simply accesses the Q-measurement or FFT by correct selection of the family number in the message.

Note that reference [5] gives a very detailed description of the standard message structures and is highly recommended reading for programmers.

2.3. Makefile

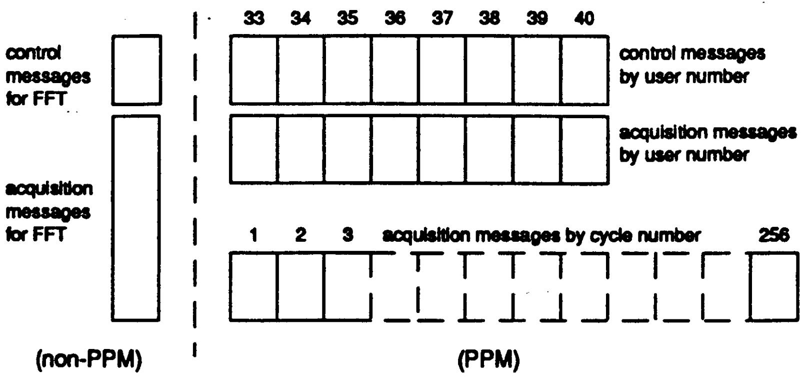
The real-time task is compiled using the `Makefile` which is located in `template`. This file is a copy of the `Makefile` used for the PPM model, minus compilation of an equipment module simulator.

To compile the real-time task the command `make` should be typed while in the `template` directory. If the source code is moved to a different directory, the root path within the `Makefile` must be corrected to allow compilation. It is possible that network libraries which are used have been moved. In this case it is best to contact the PS control group (CO) to discover the new location and modify the `Makefile` accordingly.

A conditional compilation option exists which allows screen output during real-time task execution. This output includes error and information messages, as well as indications of the times when each thread was last woken. The compiler flag for this option is `-DVISUAL`.

2.4. Initial state

The first part of the executable that runs is, as with all C programs, `main()`. `main()` commences by opening the control and acquisition message queues in a manner similar to opening files. `main()` then creates a number of buffers in memory (Fig 5). For each of the eight possible user lines of the PS accelerator a copy of the control and PPM acquisition message structures is made. A large buffer of 256 copies of the PPM acquisition message is also made, with an entry for all possible cycle numbers in the supercycle (Note that the number of cycles currently in the PS supercycle is much less than 256). Finally, one more copy of the control message structure and a copy of the non-PPM acquisition message structure is created. The purpose of these buffers will be revealed in the remaining sections of chapter 2.



control messages for FFT
acquisition messages for FFT
(non-PPM)
33 34 35 36 37 38 39 40
control messages by user number
acquisition messages by user number
1 2 3 acquisition messages by cycle number 256
(PPM)

Fig. 5. Storage buffers for real-time task copies of control & acquisition messages. For PPM measurements a copy of the control & acquisition messages exist for each PS user-line. Only one copy is made for the non-PPM measurement.

After creating the storage buffers, `main()` continues and creates the five other threads of the real-time task. As each thread is created it is given a priority higher than that of `main()` thus enabling the thread to run. The threads then re-adjust their own priority to their designed values. Fig 6 shows the program condition just after creation of the other threads.

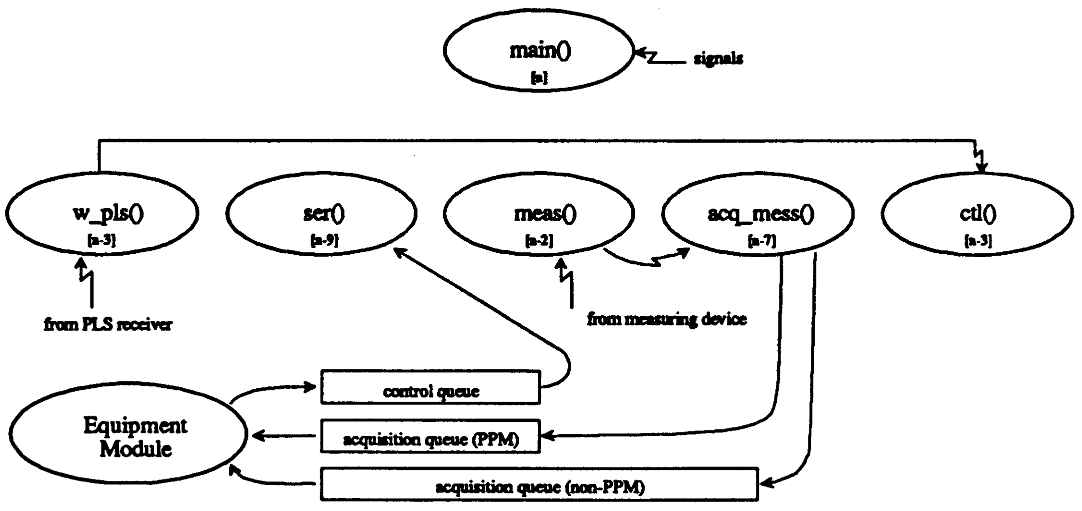


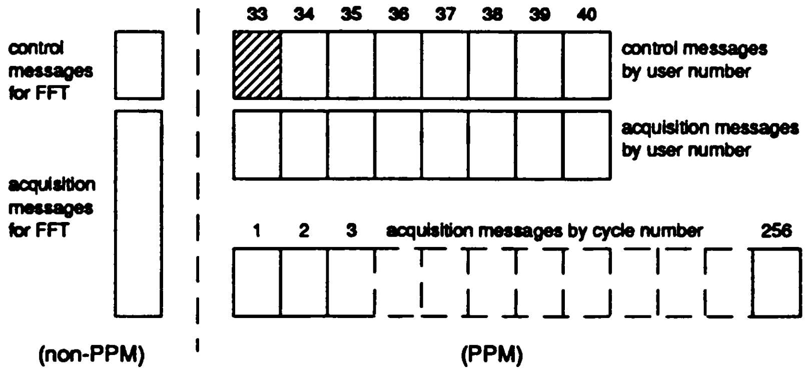
Fig. 6. `main()` creates 5 other system threads which have priorities different to that of `main()`. The message queues must already exist before the real-time task is executed.

All the threads are normally dormant until they are woken by an interrupt or signal, thus leaving the MVME147 free for other processing tasks. At this stage in the explanation of the software, then, the reader should accept that the real-time task does nothing until instructed to do so by the equipment module.

2.5. Reception of a control message

When the equipment module puts a message into the control queue, the thread `ser()` is woken by a software signal. `ser()` checks for errors in the message and then processes it if none are found. There are two types of message: synchronous and asynchronous. A synchronous message is one which requires the real-time task to give an immediate response to the equipment module, e.g. a status report. An asynchronous message on the other hand is one which does not require an immediate response. Asynchronous messages are used to instruct the real-time task to perform PPM or non-PPM measurements.

When an asynchronous message arrives, `ser()` examines the header information first. This information contains the family number, which indicates whether the request is for Q-measurement or FFT analysis, and a user-line. If the request is for Q-measurements, which is always PPM, `ser()` makes a copy of the message in the blank PPM control message structure created by `main()` (section 2.4). There is an empty control message buffer for every possible user line, so if, for example, the message is destined for user line 33, the copy would be made into the buffer for user-line 33 (Fig 7).



control messages for FFT
acquisition messages for FFT
(non-PPM)
33
34
35
36
37
38
39
40
control messages by user number
acquisition messages by user number
1
2
3
acquisition messages by cycle number
256
(PPM)

Fig. 7. A control message has arrived from the equipment module. `ser()` has discovered that it is a request for PPM Q-measurements, destined for user line 33. `ser()` has therefore made a copy of the control message in the PPM buffer corresponding to user line 33.

Alternatively the family number in the header might have indicated FFT measurement. In this case the control message would have been copied into the empty control message buffer for non-PPM measurements.

After the message has been stored, `ser()` then sleeps while waiting for a new message to arrive. To continue the explanation of the real-time task, it will be assumed that at this point in time a PPM Q-measurement is programmed to occur on machine user line 33.

2.6. Arrival of a PLS line

The thread `w_pls()` is always asleep until woken by the arrival of a PLS line. `w_pls()` reads two items of pls information, the user line and cycle number, and if there are no errors, sends a software signal to waken the thread `ctl()`.

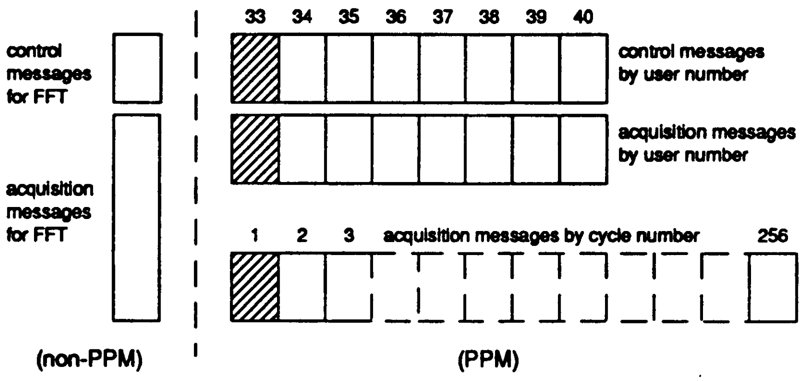
The purpose of `ctl()` is to perform all the necessary controls for the requested measurement. When awoken by the signal from `w_pls()`, `ctl()` looks into the control buffer for non-PPM measurements to check if a request has been made. `ctl()` actually looks at a line in the control message header called CCAC. This value is the actuation indicator, and its value determines whether the measurement is off or on. If `ctl()` finds that the non-PPM CCAC value is off, it then checks for a PPM measurement request. `ctl()` uses the user line number from `w_pls()` to select one of the eight PPM control buffers (Fig 7). If a control message is present for the current user line, and its CCAC is on, `ctl()` performs the requested actions.

Control actions for the Q-measurement include programming the GPPC timing modules, programming the ICV196 I/O module and the communication of measurement instructions to the VASP-16 DSP. When the last control action has been performed, `ctl()` then sleeps, again awaiting a signal from `w_pls()`.

2.7. Reception of measurement results

The VASP-16 performs a series of Q-measurements which are triggered by pulses from the GPPC modules. When the Q-measurements are complete, the VASP-16 sends an interrupt for which the thread `meas()` is waiting. `meas()` reads the results from the VASP-16 and writes them into the acquisition message, the choice of message depending of course on whether Q-measurements or FFTs were performed.

`meas()` subsequently sends a signal to the thread `acq_mess()`. `acq_mess()` continues to build the acquisition message incorporating such information as status and time. It also makes a copy of the acquisition data in the acquisition buffer for the current user and cycle numbers (Fig 8).

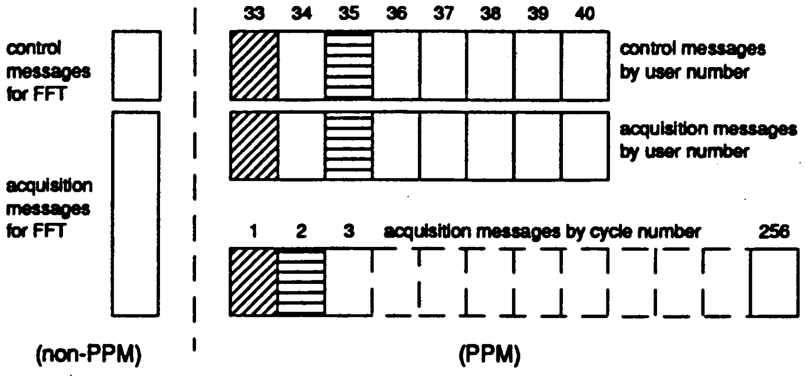


control messages for FFT
acquisition messages for FFT
(non-PPM)
33 34 35 36 37 38 39 40
control messages by user number
acquisition messages by user number
1 2 3 acquisition messages by cycle number 256
(PPM)

Fig. 8. Measurements have been performed in user-line 33. A copy of the results is stored in the buffer for user line 33, and for the cycle number in the supercycle buffer (cycle 1 is assumed).

The acquisition message is then sent to the queue, the choice of queue depending upon whether the measurement was PPM or not.

What has been described in this chapter is the complete process of the arrival of a PPM measurement request, its storage and execution at a pre-determined time, and the reception and sending of the results back to the equipment module. The real-time task will repeat the operations described in sections 2.6 and 2.7 for as long as PLS line 33 arrives. The equipment module might also request a PPM measurement on, say, user line 35. The system would thus make measurements on user lines 33 and 35 indefinitely, until the PLS line failed or the equipment module requested that either of these measurements should be given the CCAC value off (Fig 9).



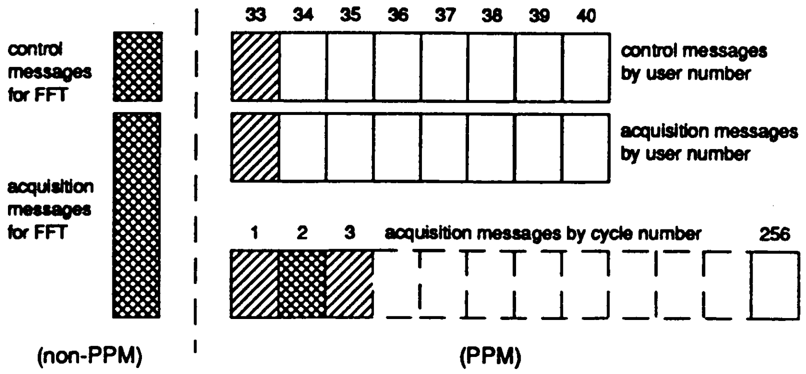
33 34 35 36 37 38 39 40
control messages for FFT
control messages by user number
acquisition messages by user number
acquisition messages for FFT
1 2 3 acquisition messages by cycle number 256
(non-PPM)
(PPM)

Fig. 9. User 33 is programmed for a PPM measurement, as is user line 35. If user lines 33 and 35 are the first two cycle in the supercycle, the storage buffers will look as shown after the two cycles have passed.

2.8. Non-PPM measurements

In the case that a non-PPM measurement is requested by the equipment module, the same process as described previously for PPM measurements will be performed. However, a non-PPM measurement has priority over PPM measurements, and therefore when the condition arrives when a non-PPM request has been made for the same user line as a pre-programmed PPM measurement, the non-PPM measurement will be performed.

Upon reception of non-PPM data, the thread `meas()` switches the actuation of the non-PPM control buffer to off. Consequently, the non-PPM request will be performed only once and the PPM measurement will re-commence upon arrival of the next similar user line (Fig 10).



33 34 35 36 37 38 39 40
control messages for FFT
control messages by user number
acquisition messages by user number
acquisition messages for FFT
1 2 3 acquisition messages by cycle number 256
(non-PPM)
(PPM)

Fig. 10. User 33 is programmed for a PPM measurement and is performed when the next user line 33 arrives, corresponding to cycle 1 of the supercycle in this example. A non-PPM request is then made for user line 33, and the arrival of the next user line 33 results in a non-PPM measurement since it has priority over PPM. Subsequent arrivals of user line 33 are treated as PPM however, because the non-PPM measurement was given an OFF actuation upon its completion.

3. DETAILED DESCRIPTION

3.1. main()

The function `main()` is one of the six threads of the real-time task, the others being `w_pls()`, `ctl()`, `meas()`, `acq_mess()` and `ser()`. `main()` is located in the file `qmbody/ini.c` and, like the other five threads, has its own include file, `ini.h`.

The first part of the executable produced by the makefile to run is `main()`. `main()` firstly performs a reset of the VASP-16 by executing the utility program `v16tool`, which must always be located in the same directory as the real-time task. The file `v16reset_ip` is re-directed to the program `v16tool` as its standard input, which results in a reset of the module. `v16tool` can be run on its own, and the commands to perform a reset of the VASP-16 are :

```
ress
init
1
q
```

`main()` next opens the VASP-16 and downloads the DSP software (`psvasp.out`) (Fig 11). Although it was intended that access to hardware should first be performed by the thread `ctl()`, it was found that in order to guarantee certain memory areas for the VASP-16 it should be attached before anything else, e.g. before the PLS software is allowed to create shared memory segments.

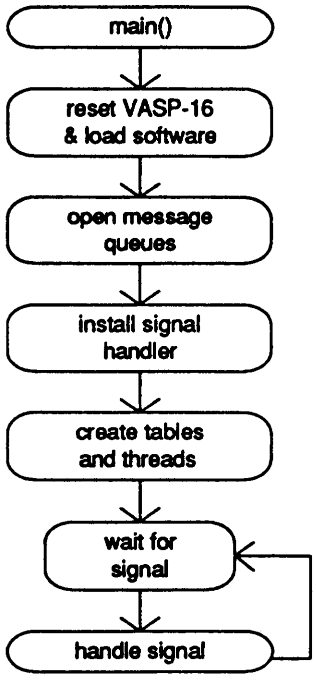


Fig 11. Functioning of main().

The function `open_qu()` is used to open the two PPM message queues, while `main()` itself contains the code to open the third, non-PPM message queue. After installing the signal handler, which is identical to the original PPM-only code, `main()` creates the data buffers and threads already mentioned in chapter 2. The only difference between the Q-measurement real-time code and the original PPM-only code in this respect is that some additional tables are created for the non-PPM measurements. These changes can be seen in the function `creer_tableaux()`.

After creating the other system threads, `main()` then acts as a simple signal handler for the remainder of program execution.

3.2. ser()

The source code for the thread `ser()` can be found in the file `ser.c`, with its respective include file `ser.h`.

The purpose of `ser()` is to service the control messages passed from the equipment module to the real-time task. After having been created by `main()`, `ser()` calls the function `q_receive2()` which is contained in the `mqlib` library. When a message arrives in the control queue, `q_receive2()` returns and `ser()` interrogates the variable `type`, which is set by `q_receive2()`, to discover whether the message is synchronous or asynchronous. Depending upon the type of message, the operation of `ser()` will then continue along one of two lines as shown in Fig 12.

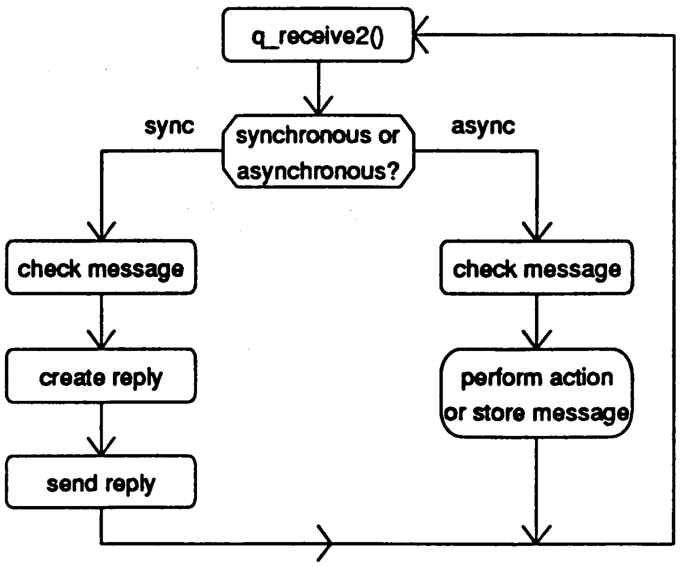


Fig. 12. Flow graph showing the functioning of `ser()`.

3.2.1. Handling of synchronous messages

Upon receipt of a synchronous message, `ser()` calls the function `syncserv()` which, as the name suggests, services the message. The function `check_sync_message()` is immediately called by `syncserv()`, which sets an error flag if any components of the message are found to be illegal, e.g. family, serial, etc. There are now five options which `syncserv()` can take. Four options deal with different types of synchronous message, and the other is taken if an error was found by `check_sync_message()`. The four types of synchronous message are:

CTRL_B_U	(Control Back User)	DATA_B_U	(Data Back User)
DATA_B_C	(Data Back Cycle)	FAUL_B_U	(Fault Back User)

CTRL_B_U means that `syncserv()` should send back a copy of the controls which are set for the user number contained in the synchronous message. Similarly, DATA_B_U is a request for the last data acquired for the user number specified in the request. DATA_B_C is the same as DATA_B_U except that the point of reference is a cycle rather than user number and hence the data is copied out of the buffer of 256 blocks. FAUL_B_U is only slightly different in that it is a request for fault information for a particular user, e.g. indicating whether the system is operational or not.

In each of the four cases, `syncserv()` copies the required information from the tables (section 2.4) into the acquisition message structure. In the event that an error was found by `check_sync_message()`, an error indicator is copied into the acquisition message meaning. The complete acquisition message is then sent to the PPM acquisition message queue using the `mqlib` library function `q_send()`.

3.2.2. Handling of asynchronous messages

Asynchronous messages are handled in a similar manner to synchronous requests as discussed in the preceding section. The function asyncserv() is the service routine which, after calling check_async_message() to verify the validty of the message, performs some sort of operation depending upon the type of asynchronous message.

asyncserv() examines the message-meaning entry to determine the type of asynchronous request that has been made. The six possibilities are:

ACT_RESET	ACT_RECOV	ACT_ON
ACT_OFF	ACT_TEST	ACT_LOCAL

The message-meaning ACT_RESET results in the user defined signal 24 being produced. This signal is picked up by the signal handler, main(), which in turn calls the function reset(). reset() releases all of the memory buffers and kills the system threads, except main(), before creating new ones again. ACT_RECOV also sends a signal to main(), although in this case only the system threads, not the memory buffers, are killed and re-created. ACT_TEST and ACT_LOCAL requests simply put the system into the special modes test and local respectively.

The final two message meanings, ACT_ON and ACT_OFF, are concerned with requests for measurements. The first, ACT_ON, is a request for a measurement to occur on a specific user line. asyncserv() makes a copy of this message in the memory buffer corresponding to the requested user line (see section 2.5) if the message is for PPM measurements, or alternatively into the single non-PPM buffer if FFT analysis has been requested.

asyncserv() then returns to ser(), which continues to listen for the arrival of control messages from the equipment module.

3.3. w_pls()

The purpose of w_pls(), contained in the files pls.c and pls.h, is to handle arriving PLS messages. w_pls() is normally asleep, waiting upon a hardware interrupt from the FPIPLS receiver (function call TgmWaitForNextTelegram()). Upon its arrival, w_pls() attempts to gain access to the mutex pls_ex and if successful, reads the user number and cycle number into the variables n_user and n_cycle respectively. The functions nu_ok() and nc_ok() are called to ensure that the two values read from the PLS are in fact legal. Finally, the function lire_other_pls_values() is called in order to read any other desired PLS information. Like the other system threads, w_pls() then goes back to sleep, waiting for a new arrival.

3.4. ctl()

The thread ctl() is to be found in the files ctl.c and ctl.h. ctl() is responsible for all necessary operations in order to perform Q-measurement or FFT analysis and is woken by a signal from w_pls().

ctl() firstly calls the function do_control(), after which ctl() goes back to sleep until a new signal is sent by w_pls(). The function do_control() performs three programming operations; on the ICV196 I/O module, the GPPC timing modules via the SDVME interface, and the VASP-16 DSP.

do_control() reads from the control message whether the requested operation is PPM or non-PPM, this information being derived from the family number (explanation in section 2.2). Based on this value, do_control() then reads the timing and equipment-specific control values, ccvs, out of the message. The ccvs have different meanings for the FFT and Q-measurements and are listed in table 1.

Table 1. Meanings of class control values for FFT & Q measurement.

ccv value	FFT (non-PPM)	Q-measurement (PPM)
0	select	size
1	size	plane
2	step_size	kicker strength
3	plane	pickup amplifier gain
4	kicker strength	pickup selection
5	pickup amplifier gain	sample frequency
6	pickup selection	- not used -
7	sample frequency	- not used -

do_control() makes a local copy of these ccvs in the array ccv_values[] to prevent an error if, e.g. ser() changed the stored ccvs while do_control was reading them. Now that the ccvs are copied locally, do_control() continues with its programming tasks. Upon completion, ctl() will go back to sleep until woken again by w_pls().

3.4.1. ICV196 I/O module

The ICV196 [6] is a general purpose 96-bit I/O module. It has three 50-pin D-type canon connectors as outputs: J1, J2 and J3. J3 has been used as the connection to the VASP-16 interface, housed in a NIM chassis, and J2 for control bits to the kicker magnet system. The bit assignment of the ICV196 outputs is shown in Table 2.

Table 2. Programming of ICV196 outputs

Bits	Purpose	Group No.
J3 18-22	Frev clock delay	10
J3 23,24	PU selection	10
J3 25	Plane preset	10
J3 26-33	Synch. to bunch	11
J3 34-43	PU amplifier gain	8 &9
J3 44	Frf/Frev selection	9
J3 45,46	Plane selection	9
J2 18-31	Kicker strength	6 &7
J2 32	Kicker on/off	7

Initial access to the ICV196 occurs in `main()` with the line

```
icv196_attach(0, 0x500000);
```

where `0x500000` is the base address to which the board has been set [7]. The library `icv196lib.o` contains the function `icv196_write()` which is used to program the output bits of the device. The syntax of the function is

```
icv196_write(0, group_number, data);
```

where `group_number` refers to one of the 12, 8-bit words into which the 96-bits have been divided (Table 2), and `data` is the value which is to be programmed.

All I/O values are programmed once at the start of the machine cycle, with the exception of the plane selection which is a little more complex. If it is desired to make measurements in more than one plane during the same machine cycle, the use of software to perform this operation increases the delay time between measurements. Consequently, some hardware was included in the interface to perform this function. Bits 45 and 46 of J1 are used to select 1 of 4 possible choices for the plane (Table 3).

Table 3. Plane selection

Bit 46	Bit 45	Plane selection
0	0	000000...
0	1	010101...
1	0	101010...
1	1	111111...

Bit 25 of J1 (plane preset) is normally kept at +5V. When the plane is to be programmed, bits 45 and 46 are set as required and a low pulse on the preset bit programs one of the output trains shown in Table 3. A "0" corresponds to the horizontal plane, and a "1" to the vertical plane.

3.4.2. GPPC timing

Two General Purpose Preset Counter (GPPC) modules [8] are used in a special manner to provide timing pulses, although these modules will in the future be replaced by a TG8 device. The control message contains three values which define the timing; start, nsamp and delay. These values define the timing of the first measurement, the number of measurements to be performed and the time space between measurements respectively. Timing is performed using the C train which gives a resolution of 1ms. A control message containing start = 400, nsamp = 20 and delay = 10, means that 20 measurements are to be performed, with the first occuring at C = 400, and a delay of 10 C pulses between each measurement.

Before the GPPC modules can be used, it may be necessary to un-bypass the Camac chassis. This can be achieved by executing the following commands on the DSC.

```
nodal
se qx=0
se scam(19,11,30,0,17,qx)=256
quit
```

Fig. 13 shows the special use of the GPPC modules to produce a burst of pulses. The GPPCs have been named START and GATE, and each contain two independent channels.

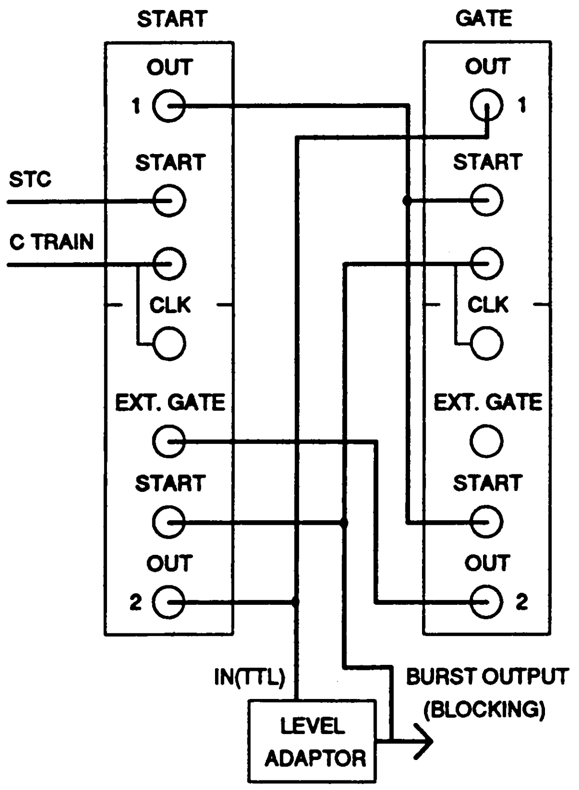


Fig. 13. GPPC connections for pulse burst.

`ctl()` performs a number of operations on the GPPC modules when it is first created by `main()`. The function `cdreg()`[1] is used to encode the GPPCs for the most commonly used sub-addresses: 0,1,2 and 3. This is then followed by a reset of both GPPCs using the `cfsa()` call with function[2] number 28. The GPPC outputs are then inhibited to prevent unwanted pulses being sent to the kicker, the C train input is selected as the clock source for both GPPCs, and the external gate for channel 2 of the 'gate' GPPC is enabled.

The preset register value for channel 1 of the 'start' GPPC is loaded with the value of `start`. Thus, an output pulse is produced by channel 1 exactly `start` C pulses after STC. This pulse is routed into the start inputs of channels 1 and 2 of the 'gate' GPPC.

The preset register value for channel 1 of the 'gate' GPPC is loaded with -1, which means that an input pulse is immediately routed to the output (with 100ns delay), which produces the first pulse of the burst at the level adapter (available from PS/CO). The output of channel 2 of the 'gate' GPPC is used to enable the output from channel 2 of the 'start' GPPC. Channel 2 of the 'start' GPPC then produces the remainder of the burst pulses, for which its preset register has been loaded with `delay`. Channel 2 of the 'gate' GPPC counts the number of pulses in the burst and when `nsamp` have been produced, it inhibits the output of channel 2 of the 'start' GPPC.

3.4.3. VASP-16 DSP

The VASP-16, if not programmed to perform measurements, is always awaiting new instructions from `ctl()`. Communication between the DSC and the VASP-16 is performed using three functions; `v16FlagT()`, `v16BoardF()` and `v16GetD()` [2]. `v16FlagT()` flags the VASP-16 to indicate that a transfer is requested, and `v16BoardF()` waits until a reply is received. An address `v16addr` is obtained using `v16GetD()` and several instruction values, e.g. measurement type, number of measurements, are then passed to the VASP-16 in relation to this address. A final `v16FlagT()` call instructs the VASP-16 that it may proceed with the measurements.

[1] For more information, use `man cdreg` while in nodal.

[2] See reference [8] for full GPPC function listing.

3.5. meas()

The thread meas() has the highest priority of all the system threads. Its function is to receive, as quickly as possible, the measurement data from the VASP-16. The source code for meas() can be found in the files meas.c and meas.h.

meas() calls the function v16myWait() to await the arrival of an interrupt from the VASP-16. When woken, meas() looks into VASP-16 global memory, which is directly mapped into VME address space, and checks which type of measurement has been performed, i.e. Q-measurement, FFT or Sliding FFT analysis.

The VASP-16 has limited on-board memory, and for this reason, after each FFT measurement, the spectrum is transferred to the DSC for storage. Q-measurements, on the other hand, produce very little data (300 words maximum), and this can easily be stored in the VASP-16 and transferred upon completion of the series of measurements, thus eliminating data transfer times.

Both Q-measurement and FFT data is written into a temporary vector, tampon[300] or tampon_1[99999] respectively. Individual FFTs, like Q values, are stored in time-order. Thus, if the FFT size is 512 bytes, then tampon_1[0-511] would contain the first FFT performed, tampon_1[512-1023] the second spectrum, and so on.

Once all the results have been obtained by meas(), the function signaler_acq_mess() is called in order to waken acq_mess(). meas() then goes back to sleep, awaiting the next interrupt from the VASP-16.

3.6. acq_mess()

acq_mess(), when woken by the signal from meas(), calls two functions only; traite_acq(), followed by stocker(). acq_mess() is contained in the files acq_mess.c and acq_mess.h.

traite_acq() sets a pointer, *data or *data_1, to the acquisition buffer tampon or tampon_1 depending upon whether the measurements were PPM or non-PPM respectively. In the case of the non-PPM FFT measurement, some special re-scaling operations are performed on the amplitudes of the spectral lines. These operations are required because of the complex FFT scaling operations performed within the Zoran ZR34161 processors of the VASP-16. A full description of this operation can be found in section 3.7 of reference [2].

stocker() first calls the function creer_acq() which, depending upon the type of measurement performed, fills in various parameters of the acquisition message such as time, status, etc. stocker() then copies the actual data values, i.e. Q-values or spectra, into the acquisition message structure. The function envoie_mesure() finally sends the message on its way, the choice of message queue being dependent upon the measurement type.

acq_mess(), like the other system threads, then goes back to sleep, awaiting a new signal in order to repeat itself again.

4. MISCELLANEOUS

4.1. Access to the PLS line

Before the real-time task is executed the PLS software must be running as a background process. This is normally performed by the start-up file `rc.local`, which also installs the device drivers. To run the PLS software manually the user should execute the program `get_tgm_vme` which is contained in the directory `/usr/local/tgm`. The program `get_tgm_simulated`, in the same directory, can be used to simulate the arrival of user lines in the absence of an FPIPLS receiver module.

4.2. Simulation of the equipment module

When developing and testing a real-time program of the type discussed in this paper, it is often of great value to be able to send messages to, and receive results from, the real-time task without the use of an equipment module. A simulator which allows this is contained in the `simulation` directory. `qmapplsim` is a menu driven program that allows the user to enter the parameters of the control message and then send it to the message queue. The user can read the acquisition message queue and view selected parts of the message. It has not been written as a general purpose program, but is specifically for the Q-measurement. Other users could use the same idea and develop the code for their protocol definitions. The source code for the simulator can be found in the file `qmapplsim.c`

4.3. Re-compilation after LynxOS version change

In the event of a LynxOS version change, which sometimes occurs as the POSIX standard develops, the source code will need to be re-compiled. To facilitate this operation, a script has been written which re-compiles the libraries, VASP-16 device driver and real-time source code automatically. The command `update_qmeasurement` should be typed while inside the directory `update`, and takes approximately 15 minutes to complete.

5. CONCLUSIONS

The use of the PPM template has greatly eased the development of the real-time task for the FFT Q-measurement. The new code needed for non-PPM measurements has been incorporated into the fabric of the original software without any alteration to its PPM functionality. This software can be used with very little modification for the PS Booster Q-measurement, and could also be used for systems such as the Closed Orbit Display (CODD) for the PS.

Given that the need for non-PPM measurement exists for a number of other systems, the control protocol definition file `npro.h` could be duplicated to provide one definition for PPM and another for non-PPM measurements, as discussed in this paper.

The control protocol provides a uniform access structure for all systems. However, because individual software developers have complete freedom when developing their real-time software, the control protocol may end up disguising a hotchpotch of varying real-time task designs. It is recommended that, like the control protocol messages, a real-time task model is properly developed. The majority of people needing to develop real-time tasks are not software experts but hardware engineers. This model would be invaluable in allowing these people to efficiently develop their systems, and would provide a stable software environment with all the associated maintenance benefits. This should be initiated well before the final slices of the LynxOS control system conversion project.

6. ACKNOWLEDGEMENTS

I would like to thank my colleagues, Elmar Schulte, Jose Gonzalez and Jeroen Belleman, for their willing assistance throughout this project; for their ideas about the software development, their help during the many meetings and their hard work during the LHC test.

Interfacing to the LynxOS control system has brought me into contact with many members of the PS controls group, all of whom have been more than willing to help.

7. REFERENCES

[1] S. Johnston, "Performance of the PS FFT Q-measurement system", CERN/PS/BD/Note 92-8, November 1992.

[2] S. Johnston, "DSP software for the PS FFT Q-measurement system", CERN/PS/BD/Note 93-5, September 1993.

[3] M. Serio, "Tune measurements", CAS, Jülich, 1990, proceedings: CERN 91-04.

[4] M. Le Gras, J. Tedesco, "Application typique du protocole pour l'instrumentation", CERN/PS/BD/Note 93-02, May 1993.

[5] USAP working group, "Final report on the uniform equipment access at CERN", CERN/PS 93-16, 18.05.93.

[6] Micronix, "ICV196", avialable from PS/CO group, August 1990.

[7] Alain Gagnaire, Wolfgang Heinze, "Selecting VME addresses and interrupt vectors", CERN/PS/CO/Note 93-054, June 1993.

[8] Part of specification PS/CCI/SPEC. 78-4, "General Purpose Counter", available from PS/CO group.

8. APPENDIX

8.1 Alphabetical function listing

The following is a comprehensive listing of the functions which constitute the real-time task. After each function name is the .c file in which it can be found.

abort (ini.c)
acq_mess (acq_mess.c)
asyncserv (ser.c)
banniere (ban.c)
box (spe.c)
check_async_message (ser.c)
check_sync_message (ser.c)
clearscreen (spe.c)
close_qu (que.c)
close_your_ctl_driver (ctl.c)
close_your_driver (meas.c)
copier_other_pls_values (spe.c)
creer_acq (ser.c)
creer_tableaux (pls.c)
creer_taches (ini.c)
creer_une_tache (ini.c)
cursor (spe.c)
do_control (ctl.c)
envoie_mesure (ser.c)
error_handler (ini.c)
fin (ini.c)
fin_meas (meas.c)
fin_pls (pls.c)
init_other_pls_values (spe.c)
lire_other_pls_values (spe.c)
lit_date (dat.c)
lit_pls (pls.c)
lu_ok (pls.c)
lu_tu (pls.c)
main (ini.c)
meas (meas.c)
menu (spe.c)
nc_ok (pls.c)
nc_tc (pls.c)
nu_ok (pls.c)
nu_tu (pls.c)
open_qu (que.c)
open_your_ctl_driver (ctl.c)
open_your_driver (meas.c)
ourtime (spe.c)
perr (spe.c)
pmsg (spe.c)
ppls (spe.c)
r_handler (ini.c)
read_def_values (set.c)
read_ligne (set.c)
recover (ini.c)
reset (ini.c)
reset_busy (ser.c)
reset_intl (ser.c)
reset_locked (ser.c)
reset_rfault (ser.c)
reset_ufault (ser.c)
reset_warning (ser.c)
reset_warning_all_user (ser.c)
send_last_mes (ser.c)
ser (ser.c)
set_busy (ser.c)
set_intl (ser.c)
set_locked (ser.c)
set_rfault (ser.c)
set_ufault (ser.c)
set_warning (ser.c)
set_warning_all_user (ser.c)
signaler_acq_mess (acq_mess.c)
skip_ligne (set.c)
spe_async (spe.c)
spe_tbit (spe.c)
special_setting (spe.c)
stocker (acq_mess.c)
syncserv (ser.c)
t_handler (meas.c)
tc_nc (pls.c)
traite_acq (spe.c)
traitement (meas.c)
tu_lu (pls.c)
tu_nu (pls.c)
tuer_tableaux (pls.c)
tuer_taches (ini.c)
tuer_une_tache (ini.c)
v16mywait (meas.c)
w_pls (pls.c)