Case study: a Real-Time Framework optimized for process control in a family of industrial equipments.

Giuliano Colla
Technical Manager - Copeca srl
Via del Fonditore 3/E - Bologna - Italy
colla@copeca.it

Abstract

The specific real-time requirements for industrial process control are illustrated, which differ from other real-time application fields, such as measurement and data collection and multimedia. Relevant factors are analysed. Namely: intrinsic performance, required human interaction, robustness and reliability, together with economic factors, such as development costs, maintenance costs and code re-usability. The YRmx framework is described, which is optimized with respect to the above factors, together with its usage in current Realtime Linux. A set of applications based on the same framework is described, used in a family of industrial equipments for the fabrication of booklets with critical high security and high productivity requirements, such as chequebooks, voucher booklets, payment booklets etc. Those equipments are currently in use by security printers all over the world.

1 Introduction

The traditional approach to software process control is well known and well established: an initialization section, followed by an endless loop which reads the inputs, applies the control algorithm, and then updates the outputs.

It is so common that the industry supplies a large number of specialized computers, providing just this processing scheme: the Programmable Logic Controllers or PLC's.[1][2][3][4]

In theory this scheme could be applied to any process, from the simplest ones, such as turning on and off a bulb following a push button, to the most complex ones, such as controlling a nuclear plant.

However, as the process complexity increases, this approach shows its limits, in terms of response time, efficiency, practical implementation, and development costs, up to the point of becoming extremely impractical or even impossible to implement.

Splitting the process in a number of sub-processes and using parallel computation techniques provides a more general solution, overcoming the flaws of the traditional approach. This requires a multi-tasking, event-driven scheduling scheme, which can be implemented on top of modern Real-Time Linux kernels.

This paper covers the case study for a family of industrial machines meant to fabricate booklets, and the chosen framework, i.e. YRmx, which is a lightweight wrapper around glibc pthread routines, to provide a set of API's optimized for industrial process control. Their simplicity and ease to use greatly help to improve software robustness, and to reduce development costs, without incurring in performance penalties.

2 Background

Industrial process requirements differ from other field of real-time applications, such as measurement and multimedia. The main peculiarity of process control is the bidirectional data flow toward external devices: data are collected from external sensors, and commands are generated toward external actuators. Another peculiarity is that individual tasks have usually very short execution times, because they must just take decisions from the input data they receive.

As the process complexity increases, the number of tasks required to implement the control functions increases accordingly, and from physical input events to physical actuation quite a number of tasks may be involved. This, together with the above peculiarities, makes task switching latency a critical factor, frequently even more important than interrupt latency.

When a number of intermediate tasks are involved between physical input and physical output, just activating a task is not sufficient: information must be passed at the same time. Therefore a mechanism which at the same time **activates a task and provides information** on “what to do” and maybe “how to do it” or “what to expect” is highly desirable. Moreover individual tasks must be as **isolated** and **independent** as possible, both for good programming practice (hidden implementation), and for ease of simultaneous developing by different programmers, and ease of debugging, testing and simulation.

In the industrial world, economic requirements must be added to the purely technical ones. Therefore **development time** and cost become important factors, together with **maintenance costs**, and **code re-usability**.

A set of API’s as simple and straightforward as possible reduces the likelihood of hard to find errors, increases the ease of verification, and encourages programmers to good programming practices. Providing programmers with the appropriate tools helps to obtain clean code, which can be written, debugged and validated in less time, and which can be easily modified and re-used, thus significantly reducing costs.

Such a set was provided by the ancient iRMX nanokernel, originally developed by Intel to promote the usage of microprocessors in the industrial automation field.[5] [6] As it fulfils quite well all of the above requirement, better than any subsequent alternative solution, we had adopted it in the past, and then we have developed our own nanokernel based on the same priciples, and evolved it for our embedded applications. At the turn of the century, when a PC solution was required, we also migrated it to a real-time Linux environment. This gave rise to YALRT, a nanokernel taking advantage of the RTAI HAL at first (RTAI 1.7) and of the RTAI Adeos patch subsequently (RTAI 3.1).[7]

With the progress of Linux kernel, and the availability of the RT_PREEMPT patch [8], which provides latencies suitable for the vast majority of industrial processes, it became possible to take advantage directly of the Linux kernel scheduling, making an extra nanokernel unnecessary, and making it much easier to adopt mainstream kernels, and current Linux distributions without being obliged to rely on third party patches. Moreover it appeared feasible to put all the real-time part in user space, as opposed to the kernel space previously used.

However, as the real-time POSIX API’s do not provide the features underlined above, we developed YRmx, a thin wrapper around pthread library[9], to expose iRMX-like API’s.

2.1 YRmx fundamentals

YRmx[13] is composed of just three software objects:

Task – a wrapper around a Linux thread, providing some extra information, such as task name and activation semaphore. Contrary to a thread, a task doesn’t terminate. The task code must be written as an endless loop. When a task is no more needed it can be deleted.

Message – a data structure constituted by a system defined header, and a user defined body. Messages aren’t physically copied or moved. Just a pointer to the message is passed. Message length is arbitrary, and user defined. The system defined header includes a length field, a type field, and a response exchange field. The latter makes it possible to avoid the dangerous and error-prone call back functions, which might otherwise be necessary.

Exchange – a data structure protected by a mutex, containing two queues (linked lists): a task queue and a message queue. At any time only one queue can be non-empty.

An exchange can be used to provide a service, i.e. to activate one or more tasks, a one-shot delay or a periodic scheduling, or memory chunks (as messages). It can be used to protect a shared resource, acting as a mutex, or to provide the functionality of a counting semaphore.

Exchanges provide a very convenient way to effectually hide tasks from one another: if good programming practice is used, only exchanges can be made publicly visible, thus hiding behind them all the implementation details.

As opposed to the over 100 primitives found in the pthread library, just two primitives are required to perform the bulk of the real-time operations, i.e. inter-task communication, task synchronization, mutual exclusion and memory management. The primitives are rqsend() and rqwait(). As they provide the basic scheduling mechanism, they're shortly described here.

rqsend()

Synopsis:

```
rqsend(*exch,*msg)
```

A task sends a message to an exchange:

1. The task queue is non-empty. The task at the head of the queue receives the message and becomes runnable.
2. The task queue is empty. The message is appended to the message queue and the running task continues execution.

rqwait()

Synopsis:

```
*msg = rqwait(*exch,timeout)
```

A task requests for a message at an exchange.

1. The message queue is non-empty. The task receives the message and continues running.
2. The message queue is empty and timeout value is zero. The task is suspended and appended to the task queue.
3. The message queue is empty and timeout value is non-zero. The task is suspended like in case 2), but if the timeout expires without a message being sent to the exchange, the task will receive a *timeout* message, and made runnable.

In addition a non-blocking function is available (***rqacpt()***) which returns a message if available and a NULL pointer if no message is available.

In summary YRmx provides:

1. 1 initialization procedure
2. 4 procedures for task synchronization, inter-task communication, mutual exclusion and memory management.
3. 3 creation procedures
4. 2 deletion procedures
5. 4 procedures for debugging purposes.

making it a very simple and easy to learn and to use tool.

A summary of YRmx basics can be found in Appendix A.

From the above description is easy to see how the YRmx framework provides all the required functionalities, with very few simple and intuitive primitives and rules, making real-time program development, debug and maintenance simpler and less costly. It can be considered a RAD tool for Linux real-time programming.

2.2 YRmx performance

The above results have not been achieved at the cost of reduced performance. Not only the overhead with respect to plain pthread library functions has shown to be negligible, but in some cases YRmx provides a significantly better performance, when timed wait are involved.

In order to choose the best solution, the ***rqwait()*** function has been implemented in YRmx both by taking advantage of the *sem_timedwait* function of the pthread library, and by exploiting the delay list technique of RMX (see appendix B for details).

Frank program test

	@1.2GHz			
		Min	Avg	Max
sem_timedwait				
	one-way	2158	16355	45616
	round trip	3458	**21184**	50379
YRmx delay-list				
	one-way	1761	6869	37505
	round trip	3234	**11287**	42658
	@2.2GHz			
sem_timedwait				
	one-way	3730	16171	55855
	round trip	5520	**20986**	57713
YRmx delay-list				
	one-way	1516	6303	33945
	round trip	2819	**15259**	35569

Here's a comparison table of the measured task switching latencies by using sem_timedwait and YRmx delay-list technique. It is the classical RTL Frank program with added profiling information and with the addition of a third task, which perturbs the two task communication each 117ms. The test has been performed on a 4 core Intel I7 processor @1.2 GHz and @2.2GHz, one isolated core for real-time tasks. 100k iterations. OS is CentOs 6.6, kernel

3.10.10-rt7.Times are in nanoseconds.

2.3 Debugging

Ease of debugging is a very important issue. As it is running in user space, gdb can be used for debugging, but a number of specific extra debugging tools are provided. Tasks status, exchanges, tasks waiting for a timeout to expire etc. can be examined. Applications specific additional debug functions can be added following the guidelines of the ones provided. A "freeze" command is also available (both from debug console and programmatically), to actually freeze real-time execution and make it possible to analyse a snapshot of the real-time situation at a given moment.

Here's a sample of the output during the Frank program execution.

Program output:

```
....
99 Frank Sinatra
100 Frank Sinatra
was The Voice
```

Debug output:

```
t  <---- user command to show tasks
-------- Task List --------
 Name  st     exch    last msg  pri
*Root* 10      (nil)      (nil)  99
*Timer 10      (nil)      (nil)  98
*DELAY 80  DELYEX_EX  0x606a08  97
 FRANK 40   FRANK_EX  0x606ec0  85
SINATR 40  SINATR_EX  0x606ec0  81
WATCHD 40  WATCHD_EX  0x606910  43
---------------------------

x  <---- user command to show exchanges
------- Exchanges --------
Exchg: DELYEX_EX Queued Tasks: *DELAY
Exchg:  FRANK_EX
Exchg: SINATR_EX Queued Tasks: SINATR
Exchg: *WAIT*_EX Queued Tasks:  FRANK
Exchg: WATCHD_EX Queued Tasks: WATCHD

d  <---- user command to show delay list
------- Delay List --------
 Name   Delay    exch    last msg   pri
 FRANK      5  *WAIT*_EX  0x606ec0  85
WATCHD     73  WATCHD_EX  0x606910  43
```

3 Project specifications

3.1 Project aim

The project aim is to implement a modular software to control a family of industrial machines to be used in conjunction with digital printers, for the fabrication of high security booklets used by security printers, banks, service centres, travel operators, government offices, etc.[12]

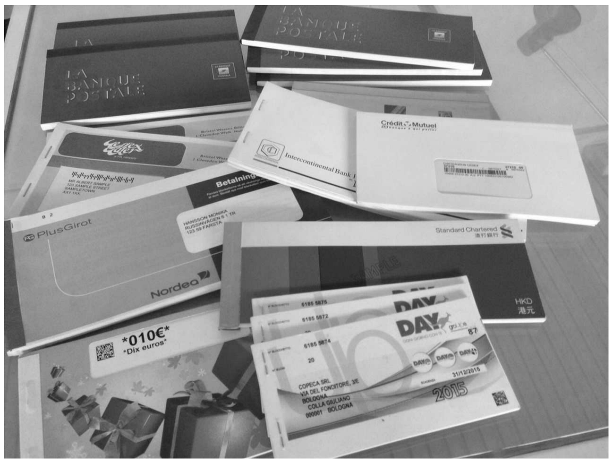

FIGURE 1: *Typical booklets*

The input stream may come in many different formats, namely cut sheet or continuous forms, either fanfold or in rolls. Cut sheet may be typically in A4 or similar format, with 3 or 4 documents per page, or in A3 or similar format with 6 or 8 documents per page, depending on the size of the final product.

Continuous form can be printed with one, or more documents per line, depending on input paper width and final booklet size. Printing can be either N-S or E-W. Up to four documents per line are supported.

One or more input sections are required, to perform the first part of booklet fabrication, i.e. cutting, slitting, merging and sequencing, in order to provide a stack of sheets which constitutes the body of the booklet.

The input section may have facilities to insert precut documents, in order to add publicity, or other required material.

Depending on the input stream format, and on the number of different streams which must be merged in the final documents, a number of different input sections must be handled, and their number may vary from one to three.

Following the input section(s) a finishing section is required, to fabricate the final product. The finishing section can add precut top and bottom covers, precut ads, rotate the booklet either upside/down or left to right, for proper orientation, add staples or apply glue to the spine, add a spine tape, divert to a separate belt or waste bin dubious booklets, insert appropriate separators, and stack the finished products in an output conveyor, or send it to a subsequent downstream equipment, for further processing (banding in groups, wrapping, etc.)

The throughput will be in the order of 2000 25 pages booklets/hour, meaning that the input section must operate above 50,000 pages/hour.

FIGURE 2: *Typical machine*

The full system must provide positive security, by keeping track of all the fabrication steps, and generate logs reporting machine activity, errors, and production information for usage by QC, for traceability and for usage optimization. It must also be able to connect to the factory LAN in order to fetch from data bases job data, and to store production logs.

3.2 Project requirements

The sequence of operations required to fabricate a booklet takes a time in the order of 20/25 seconds. Therefore, in order to achieve the desired throughput, the sequence is split into a number of steps, so that while step *n* is executed on booklet *x*, step *n+1* is executed on booklet *x-1.* Each step can be split in sub-steps, following the same logic, to achieve the required performance.

This requires a process control system which must be capable of handling simultaneously a number of sub-processes (typically 50 to over 100, depending on the machine model) with a time response in the sub-millisecond region.

The control system must have a modular structure, in order to accommodate the different machine models without rewriting common parts allowing for simultaneous development from a number of programmers.

Software modules must be as independent as possible to ease testing, simulation, validation and reuse.

A user-friendly GUI is required to provide means to operate and service the machine, program its jobs, set-up parameters, and troubleshooting.

There must be provisions for network connection, to fetch job data from company servers and to store log information on them, and for access from remote, for troubleshooting and software upgrades.

4 System design

The available techniques to achieve parallel operation are:

- intelligent peripherals

- multiple processors (distributed intelligence)

- real-time multitasking

All of the three have been used in this project, leading to a general hardware structure as shown in figure 4.

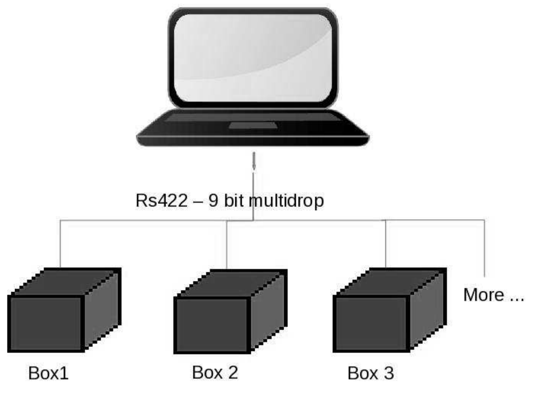

FIGURE 3: *Hardware Layout*

and to the corresponding software layout (figure 4)

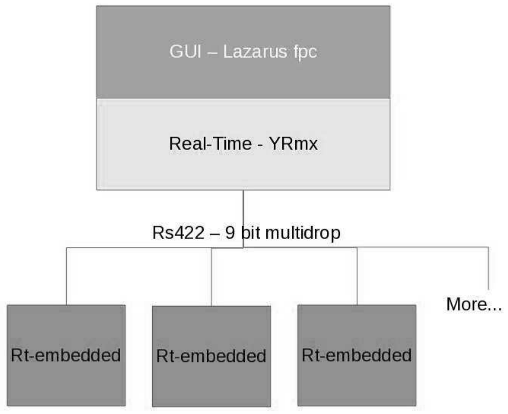

FIGURE 4: *Software Layout*

The external Boxes implementation is outside the scope of this paper, so we will concentrate to the main PC.

5 Hardware

The project specifications require all the capabilities of a desktop PC, but the harsh industrial environment must be taken into account. Moreover the hardware must be well supported by Linux.

Therefore an industrial-grade PC has been selected, from the same manufacturer we had successfully used in the past (NCS Computers[9]), which has already proved to provide the required reliability and which guarantees the long term support required by industry. A 16GB flash disk was selected to improve reliability.

Two models have been tested, one based on an Intel Atom N2600 and one based on a quad core Intel I7 processors. The Atom N2600 did show slightly better real-time performance, but the I7 was finally selected, because of the better Linux support of the graphical interface.

In order to keep CPU temperature as low as possible, for increased reliability, it is not run at full speed (2.2 GHz), but rather at reduced speed (1.2 GHz). The reduced speed doesn't show to affect significantly overall performance.

6 Software

For system OS we evaluated LFS (Linux From Scratch), but we discarded it, due to the excessive maintenance costs. Finally we resorted to CentOs 6 which provides all the required features, also in terms of configurability and long term support, and which is also compatible with RHEL 6 which we're using internally in our company.

We selected KDE as desktop environment, because of better configuration flexibility and more pleasant graphic look.

We selected kernel 3.10.10-rt7 which proved to provide the best overall performance, as a compromise between real-time and non real-time behaviour.

6.1 Human Interface

The human interface must provide a pleasant and intuitive look toward the user, and must seamlessly interface with the real-time part, both sending commands and reacting to asynchronous events. We selected fpc Lazarus [10] as a RAD tool which provides the required features.

It provides different levels of permissions, depending on the rights granted to the user logged in. As a general rule **only the commands permitted to the user, and allowed by the current context of the machine state are shown**, thus avoiding confusion and possible errors.

It is organized as a finite state machine, whose state diagram is shown in fig. 5.

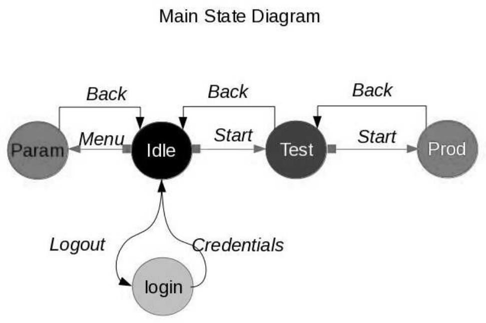

FIGURE 5: *Main State diagram*

This scheme makes it impossible to change critical parameters or to give test commands while the machine is running, while it makes possible to set up initial configuration parameters, even if the machine is non functional or not connected.

The connection with the real-time application is made using stock Linux IPC facilities, i.e. a number of fifo's, writing and reading messages already formatted as YRmx messages. The IPC scheme is shown in fig 6.

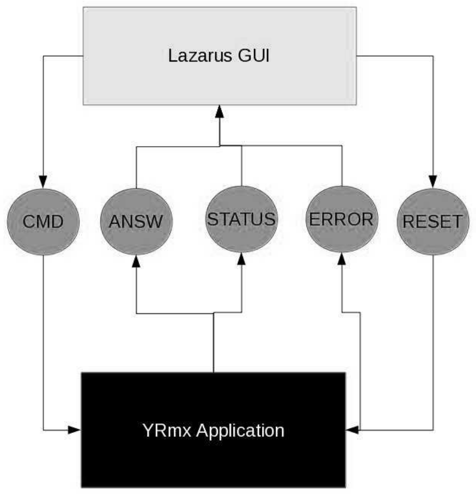

FIGURE 6: *Inter Process Communication*

The ***cmd*** fifo is used to send messages to the real-time tasks, and the ***answ*** fifo to receive associated answers.

The ***status*** and ***error*** fifo's receive asynchronous messages from the real-time tasks.

Status messages are used to update the display to provide a visual feedback of machine operation.

Error messages are used to display error information. A synoptic of the machine is shown, with the icon of the offending device highlighted.

The ***reset*** fifo is used to send a reset message to the real-time section, which clears the error condition and allows to resume the operation.

Writing to fifo's is performed in the main thread of the application, while reading is performed by an auxiliary thread, which appends received messages in appropriate queues and then wakes up the loop of the main thread by means of fpc messages, to ensure proper display of asynchronous received data.

7 Real-time design

In order to ensure consistency across the project, to take advantage of YRmx features and to provide adequate system robustness, a number of general rules have been established which every programmer is required to comply.

7.1 Public and private exchanges

In order to permit inter-task communication and synchronization, the exchanges where tasks wait for activation messages must be public and system-wide visible. They should be created at startup taking advantage of the startup procedure (***rqstart)***. On the contrary, the exchanges where it waits for responses must be private, to ensure that only valid responses are received. It is task responsibility to create them.

7.2 Message rules

All application messages must be of the same length. This makes it possible to take advantage of a common message pool. Whenever possible the Standard message structure is to be preferred.

A task receiving a message must always either forward it, or send it back to the message response_exchange, with type and content modified appropriately.

Sent Message types are in the range 0x40 – 0x7F, lower values being reserved for system messages. Response message types are in the range 0x80 – 0x9F, with even values reserved for no-error responses, and odd values for error responses. Message types for handling a process fall in the following categories:

1. Messages to **provide information**, such as configuration or job parameters – Expected response: either ***exec_cmd*** (0x80) or ***non_exec_err*** (0x81)
2. Messages to **query information** such as ***status_req*** (0x48) – Expected response: congruent with the requested information e.g. ***reply_status*** (0x84); ***non_exec_err*** (0x81) in case of error.
3. Messages **requesting an action**, such as ***start*** (0x40), ***cmd_man*** (0x42) – Expected responses: ***exec_cmd***, ***exec_err***, ***non_exec_err***.

Task action upon receiving a message is, of course task dependent.

The response to a message, and the action upon receiving a response are subject to strict rules, to ensure consistency, simplicity and robustness.

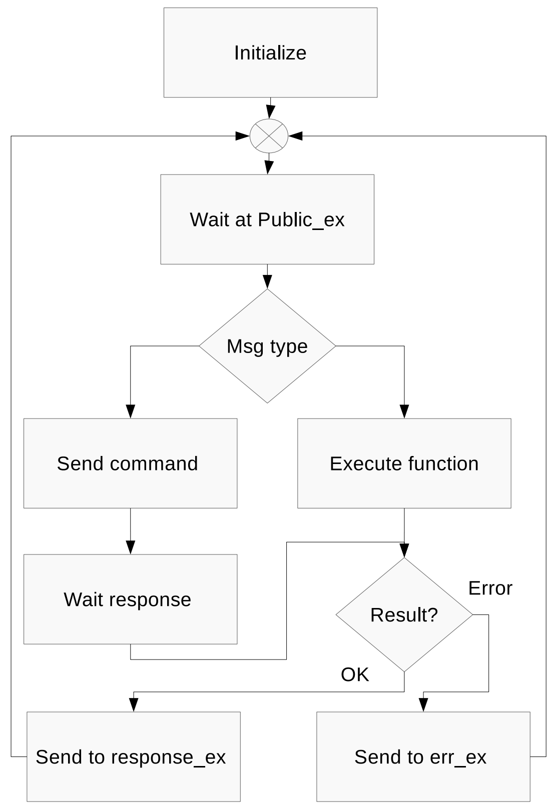

FIGURE 7: *Skeleton Application Task*

For better overall efficiency a task should **send the response as soon as there is a condition for the other tasks to continue**, and execute any required local further operation after that.

If the requested action has been successfully completed the response type will be an even type, context dependent, and it will be sent to the *response_exchange*.

In case of error, the response type will be an odd type, and the message will be sent to the *err_ex*, to be processed by the error handler task. The error handler will take care to send the message to its *response_exchange*.

An error response can be either ***non_exec_err*** or ***exec_err***. The first one means that the requested action has failed, and it should be retried. The second one means that an error occurred , but the requested action has been however completed.

The benefits of this solutions are that a tasks receives only the information relevant for its execution, i.e. continue or retry (or recover-retry depending on the specific case), and that the error is displayed as soon as it is detected, and not when the receiving task is activated.

A skeleton flow-chart of a typical application task is shown in fig. 7.

7.3 Message structure

As we have seen, the main PC is connected to a number of slave units, which take care of the lower level control functions. Each slave unit is partitioned in a number of physical or logical devices. A unique device numbering has been chosen, so that each device number maps to a single device, and to the public exchange which can be used to send messages to. Message structure must take into account this fact.

The standard message structure is described below. For some special purposes (such as sending configuration and some application dependent data), other layouts are used, but always keeping the same total length. As messages need also to be sent remotely, they're **byte packed** because the efficiency gained by better memory alignment would be offset by the overhead of longer transmission time or conversion before transmitting and after receiving.

```
#pragma pack(1)
#define MaxPosLen   192
typedef struct {
        msghdr;
        unsigned short  tag;
        unsigned char   unit;
        unsigned char   len;
        unsigned char   device;
        unsigned int    fstat;
        unsigned int    ftout;
        unsigned int    fover;
        unsigned short  cuts;
        unsigned char   pos[MaxPosLen];
        } STANDARD_MSG_DESCRIPTOR;
```

msghdr is described in Appendix A, and it includes the *type* field.

The *pos* array field is used to store task dependent information. The *len* field specifies the actual amount of the array required, when this can't be inferred by the message type.

The *device* and *unit* fields are explained above. The *unit* field is somewhat redundant because each device maps to a specific unit, but it helps to simplify the code.

The fields *fstat ftout* and *fover* are used mainly for error handling and will be seen in detail later on.

7.4 Main control logic

Also the real-time part is conceived as a finite state machine, with just two top-level states: Running and Stopped. This is achieved by means of two exchanges, named *start_ex* and *stop_ex* and a message, named *go_msg*. Whenever the *go_msg* is appended to *start_ex* the machine is in the Running state. When it's appended to *stop_ex*, the machine is in the Stopped state. The *go_msg* type provides further sub-state information. The main control logic is shown in fig.8

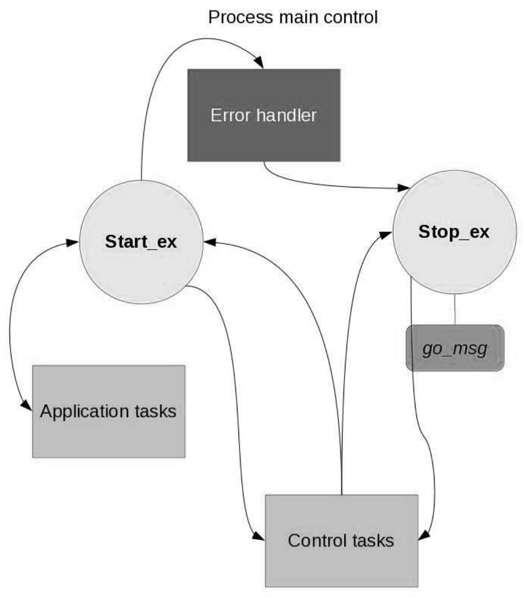

FIGURE 8: *Main control logic*

The state can be changed by

1. manual commands coming from HI,
2. a control task decision such as end of job, end of cycle in a single booklet mode, temporary hold, waiting for an upstream or downstream device
3. the Error Handler

The *go_msg* types are

1. *m_hlt*: single booklet mode
2. *m_err*: error condition, set by the Error Handler upon receiving an error, and cleared by the same upon receiving a reset
3. *m_go*: continuous operation
4. *m_hold*: waiting for an external device

At each relevant step of execution application tasks will wait for the *go_msg* at the ***start_ex*** exchange, and then put the message back.

This is a snippet of code to show how this can be used in practice:

```
static void WaitOk (void) {
MSG_DESCRIPTOR *hmsg;
    hmsg = rqwait(&start_ex,0);
    rqsend(&start_ex,hmsg);
    }
```

This is another snippet, including the support for the step-by-step operation, useful for debugging and mechanical troubleshooting:

```
static void WaitOk (void) {
MSG_DESCRIPTOR *hmsg;
    hmsg = rqwait(&start_ex,0);
    if (ByStep) hmsg->response_exchange = &stop_ex;
    rqsend(hmsg->response_exchange,hmsg);
    }
```

Multiple error may occur, therefore the Error Handler must be able to properly set the stop-in-error condition, wherever the *go_msg* is currently appended. The code used is the following:

```
static MSG_DESCRIPTOR *search_go (void) {
    MSG_DESCRIPTOR *go_msg;
    go_msg = rqacpt(&start_ex);
    if (go_msg == NULL) go_msg = rqacpt(&stop_ex);
    while (go_msg == NULL) {
        go_msg = rqwait(&start_ex,5);
        if (go_msg->type == timeout_type) {
            go_msg = rqwait(&stop_ex,5);
            if (go_msg->type == timeout_type)
              go_msg = NULL;
            }
        }
    return (go_msg);
    }

static void SetStop(void) {
MSG_DESCRIPTOR *HMsg;
    HMsg = search_go();
    HMsg->type = m_err;
    HMsg->response_exchange = &stop_ex;
    rqsend(HMsg->response_exchange,HMsg);
    if (err_msg->device == general_dev)
        MotorOff();
    }
```

```
static void ClrErr(void) {
MSG_DESCRIPTOR *HMsg;
    HMsg = search_go();
    if (HMsg->type == m_err) HMsg->type = m_hlt;
    HMsg->response_exchange = &stop_ex;
    rqsend(HMsg->response_exchange,HMsg);
    }
```

Where the procedure ***search_go*** is used to locate the *go_msg*, taking into account that other tasks may have taken it temporarily, while SetStop sets the error condition, and ClrErr clears it.

7.5 Serial Communication

To ensure the required reliability in the harsh industrial environment, we selected a four wires RS422 9 bit multidrop scheme @ 230 kBaud. We avoided to write a special driver, and we just used Linux I/O.

7.5.1 Transport layer

The transport layer protocol is a derivative of the BitBus protocol, with some additional features inspired from the ancient IBM's BiSync protocol, to improve robustness. It is a master/slave protocol allowing for packets of up to 248 bytes.

The master (PC side) performs a polling cycle every millisecond, or when it receives a message to send, whichever comes first. It keeps in a buffer a message for each slave unit, both for proper synchronization with the polling cycle, and to permit retransmission in case of error. The following snippet of code shows the core of the polling task. It demonstrates how a task can be made periodic, how a periodic task can be suspended and resumed,or started immediately.

```
if (!PollMask) tmax = 0;
else tmax = 1;
a_msg = SMD rqwait(&mast_ex,tmax);
if (a_msg->type > timeout_type) {
    if (a_msg->type == MaskDevice)
        PollMask = a_msg->pos[0];
    else {
        RX_ID = a_msg->unit & 0x7;
        xmit_buf[RX_ID].len = a_msg->pos[0];
        xmit_buf[RX_ID].retries = 10;
        }
    a_msg->type = exec_cmd;
    rqsend(a_msg->response_exchange, MD a_msg);
    }
```

The simplified logical structure is shown in fig. 9.

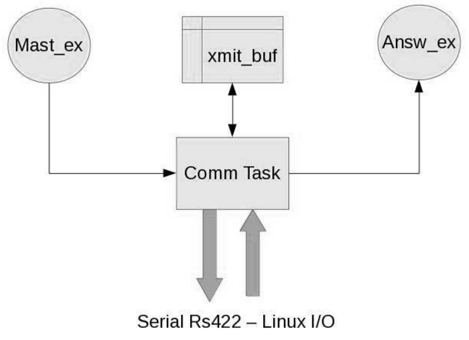

FIGURE 9: *Low level communication*

7.5.2 Data transmission

As the message structure is byte packed, data transmission may disregard the message type, and treat all messages as a sequel of bytes. This simplifies the code, and makes data transmission logic independent from the message structures actually used for different purposes.

Data transmission logic must take into account that

1. for efficiency, the packet sent should be as short as possible
2. all the messages are funnelled through a single physical serial line
3. almost simultaneous requests to send a message may come from different tasks and bound to different slaves
4. the translation of messages into serial packets involves a certain overhead, to compute the checksum, to escape the 0xFF characters, etc. This overhead should not penalize other tasks, and generate priority inversion.
5. a message can be considered successfully sent only when an ACK has been received.
6. we must be able to handle the response to a message
7. we must be able to handle gracefully the failure to send a message

In order to comply with those requirements, a number of measures have been taken, namely:

1. the full header of the message is not sent, but only the tag and type fields
2. the send logic has been split into a driver task and a send task
3. one driver task per slave has been assigned. The appropriate task is selected by a public dispatching subroutine.
4. the individual driver tasks must handle all the packet formatting functions
5. the transmission task sends the ready-to-send packets to the low level driver task, waits for the result, and, when the transmission has been successful, it stores the original message into a list to honour the subsequent response.
6. an abort command can be sent to abort the sending of a message,and avoid to wait indefinitely for the response to a message which has not been sent.

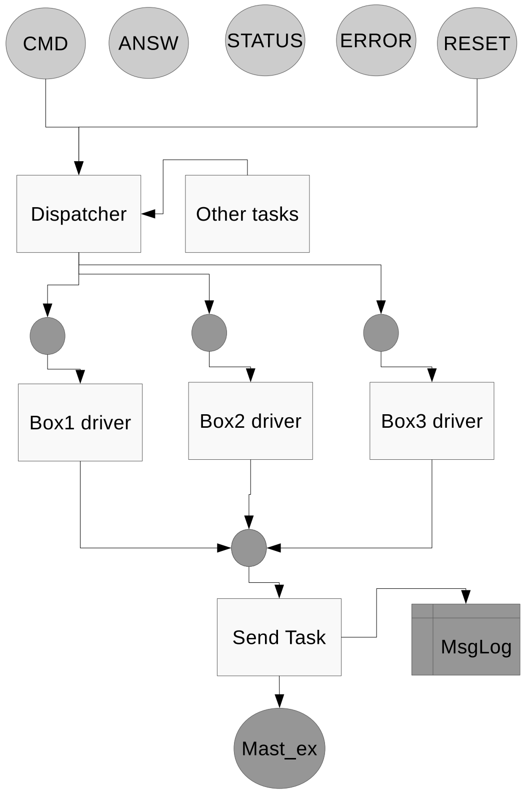

FIGURE 10: *Data transmission*

As the driver tasks share the same code, except for the exchange they wait at, and a few other minor details, multiple copies of the same task are created, with the initialization section taking care of adjusting the required details.

The simplified structure of data transmission is shown in fig. 10, with abort logic omitted for simplicity.

7.5.3 Data receive

The received message can be either the response to a message sent, or an asynchronous (unsolicited) message, to provide status or error information. An unsolicited message is recognized by a null tag field.

For a non-null tag field the receiving task takes advantage of the tag field, to recover from the MsgLog list the original message. It will copy the information received to the original message and send it to its response exchange, or to the error handler if the received type tells so.

To handle unsolicited messages, a dispatching table is supplied, which maps each sending device (from the dev field of the message) to the appropriate exchange. Therefore a new message can be obtained from the pool, the body is filled from the received message, and the response exchange is set from the dispatching table. The most usual condition is to handle just error messages. In that case the only actions required are those performed by the error handler and the response exchange will be just the message pool. The simplified structure of data receive is shown in fig. 11.

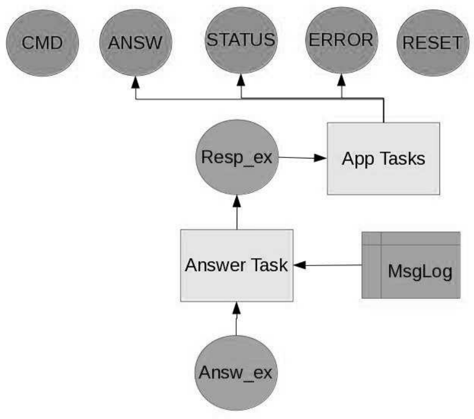

FIGURE 11: *Data receive*

7.6 Error handling

We have already seen how errors are handled in the flow of the process control execution. It remains to describe how errors are communicated to the operator, so that he/she can take corrective actions.

In order to provide appropriate user feedback, we have decided to display a synoptic of the machine, showing the state of all the sensors in the section involved, with the one(s) giving rise to the problem highlighted.

Experience has shown that the vast majority of the errors are detected by input sensors and fall into two categories:

1. an expected event failed to occur within the expected time (time-out errors)

2. an unexpected event occurred (overrun errors)

Therefore the message structure includes three 32 bits fields: one to represent the current state of up to 32 input sensors (the ***fstat*** field), one to mark the ones affected by timeout error (the ***ftout*** field), and one to mark the ones affected by overrun error (the ***fover*** field) When a slave unit has more than 32 input devices, the ***dev*** field of the message can be used to determine which part of the offending section should be displayed. Sensor state is shown with appropriate icons with transparent background, while timeout errors appear in blue background, and overrun errors in red background.

Of course error display is a function of the HI part. The real-time responsibility is to fill up the fields as required.

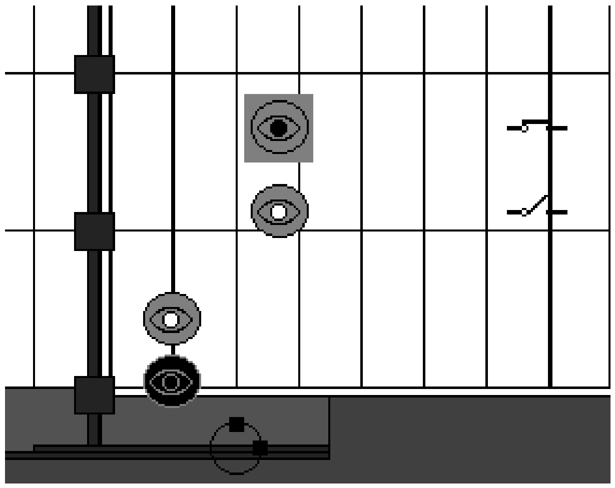

FIGURE 12: *Error display*

Errors not falling in the above categories carry a different type, and the error fields carry additional detail information. A screen snapshot of a portion of the error display is shown in fig 12.

7.7 Debug and simulation

The first machine where the YRmx framework has been used is perhaps one of the simplest of the family, however the Object Pascal code for the HI exceeds the 20000 lines of code, while the C real-time part code is little less than 10000 lines.

Simply replacing the driver tasks of the communication part with others which provided the appropriate response with an appropriate delay, and which inserted some random errors, leaving all the rest untouched, has permitted to test and debug all the software, and to evaluate the performances, without the need of the physical large machine, which was under construction.

The actual debugging on the machine has required no more than two extra days, to fix the few details which had not been detected by the simulation.

The subsequent factory tests to validate the product, which lasted several weeks, didn't show any unforeseen problem.

8 Conclusions

Our goal was to take advantage of an RT_PREEMP patched kernel and to make available a flexible, easy to use and robust set of API's for our industrial control applications.

The case selected is representative of a very wide set of applications in many industrial fields we have dealt with, such as textile, packaging, and different manufacturing equipments.

The results we have obtained speak well in favour both of the work of Linux Kernel developers, and of the YRmx framework.

YRmx source code is licensed under LGPL and can be freely downloaded from our website.[13]

A YRmx basics

A.1 Exchange structure

```
typedef struct exchange_descriptor  {
  struct msg_descriptor *message_head;
  struct msg_descriptor *message_tail;
  struct task_descriptor *task_head;
  struct task_descriptor *task_tail;
  struct exchange_descriptor *exchange_link;
  pthread_mutex_t mutex;
  unsigned char name[EXCH_NAME_LEN+1];
  } EXCHANGE_DESCRIPTOR;
```

A.2 Message structure

```
#define msghdr \
  struct msg_descriptor *link; \
  unsigned int length; \
  unsigned int type;\
  unsigned int yltime; \
  struct exchange_descriptor *home_exchange; \
  struct exchange_descriptor *response_exchange

typedef struct  msg_descriptor {
  msghdr;
  unsigned char user_defined [1];
  } MSG_DESCRIPTOR;

#define int_type  1
#define missed_int_type  2
#define timeout_type  3
```

A.3 Task structure

```
#define task_links \
  struct task_descriptor *next; \
  struct task_descriptor *prev; \
  struct task_descriptor *link; \
  unsigned short delay;\
  struct timespec ts;

#define td_middle_part\
  struct exchange_descriptor *exchange; \
  sem_t          sema; \
  pthread_t      thread; \
  void           (*thread_start)(); \
  unsigned char  priority;\
  unsigned char  status;\
  struct static_task_descriptor *nameptr;

#define td_end_part \
  struct task_descriptor *tasklink;\
  unsigned char master_mask;\
  unsigned char slave_mask;

typedef struct task_descriptor {
  task_links;
  struct msg_descriptor *message;
  td_middle_part;
  td_end_part;
  } TASK_DESCRIPTOR;

#define delayed     0x01
#define suspended   0x02
#define maskint     0x04
#define helper      0x08
#define running     0x10
#define needpost    0x20
#define deleted     0x40
#define waiting     0x80
```

A.4 API's

```
typedef EXCHANGE_LIST_PTR *IET[];
typedef STD_LIST_PTR *ITT[];

extern int reqstart (STD_LIST_PTR itt[],
  int ntask,EXCHANGE_LIST_PTR iet[],int nexch);
extern void reqctsk(STATIC_TASK_DESCRIPTOR *);
extern void reqcxch(EXCHANGE_LIST_PTR,
 const char *);
extern void reqcmsg(MSG_DESCRIPTOR * , int );

extern void reqdtsk (TASK_DESCRIPTOR *);
extern int reqdxch (EXCHANGE_DESCRIPTOR *);

extern void reqsusp (TASK_DESCRIPTOR *);
extern void reqresm (TASK_DESCRIPTOR *);

extern void reqfreeze(void);
extern void reqbake(void);

extern void reqsend (EXCHANGE_LIST_PTR ,
  MSG_DESCRIPTOR *);
extern MSG_DESCRIPTOR * reqacpt (EXCHANGE_LIST_PTR );
extern MSG_DESCRIPTOR *reqwait(EXCHANGE_LIST_PTR ,
  unsigned short);

extern unsigned int reqsystime();
```

A.5 Compatibility API's

```
#define rqstart reqstart
#define rqctsk reqctsk
#define rqcxch reqcxch
#define rqcmsg reqcmsg

#define rqdtsk reqdtsk
#define rqdxch reqdxch

#define rqsusp reqsusp
#define rqresm reqresm

#define rqfreeze reqfreeze
#define rqbake reqbake
```

```
#define rqsend reqsend
#define rqacpt reqacpt
#define rqwait reqwait

#define rqsystime reqsystime()
```

B Delay list

YRmx provides timed waits, with the resolution of the YRmx system tick whose suggested value is 1 ms. Timed waits can be used to provide a timeout when waiting for an event to occur, to insert a delay or to provide periodic task activation. The technique used provides a minimal overhead, independent of the number of timings required.

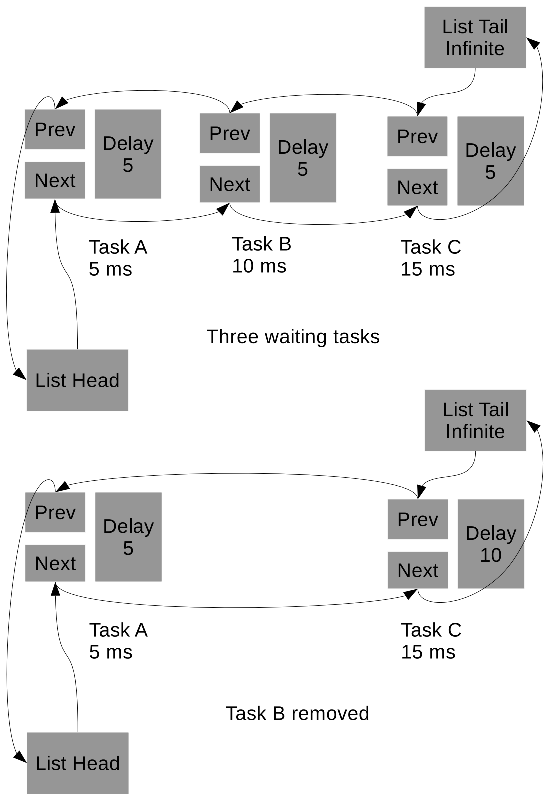

FIGURE 13: *Delay list*

It is implemented by a doubly linked list of the waiting tasks, ordered by increasing delay, and each one carrying the additional delay with respect to the previous. In this way the system tick needs only to decrement the value at the head of the list. When it goes to zero the time has expired, the task is removed from the list and made runnable, and the next one becomes the head of the list. The doubly linked list makes inserting or removing an item quite fast. This technique eliminates any overhead at each tick, at the cost of a minimal overhead to insert a new item in the appropriate place of the list. Fig. 13 shows the list status, with three tasks respectively waiting 5, 10 and 15 ticks, and the effect of removing one of them from the list.

References

[1] *https://industrial.omron.co.uk/en/home*

[2] *http://w3.siemens.com/mcms/industrial-controls/en/Pages/default.aspx*

[3] *http://www.schneider-electric.co.uk/sites/uk/en/products-services/automation-control/products-offer/automation-controllers/automation-controllers.page*

[4] *http://www.br-automation.com/en-gb/perfection-in-automation/*

[5] *Robert Kahn – A Real-Time multitasking executive* – 1972(?), IEEE Proceedings

[6] *iRMX 88 Reference Manual* – 1984, Intel Corporation

[7] *https://www.rtai.org/*

[8] *https://rt.wiki.kernel.org/index.php/*

[9] *IEEE Std 1003.1c^{TM}*-1995, IEEE Standard for Information Technology–Portable Operating System Interface (POSIX®)

[10] *http://www.ncs-computer.com/*

[11] *http://www.lazarus-ide.org/*

[12] *http://www.cemitalia.it/en/finitura_libretti.php*

[13] *http://www.copeca.it/YRmx/*