Real-Time Magazine
Diepenbeemd 5 - 1650 Beersel - Belgium
Vol. 5 nr. 3 91/3, ISSN 1018-0303

ALBATROSS
An Operating-System under hard Realtime-Constraints

Ewald von Puttkamer & Uwe R. Zimmer

University of Kaiserslautern - Computer Science Department - Research Group Prof. E. v. Puttkamer
P.O. Box 3049 - W6750 Kaiserslautern - Germany
Phone: 49 631 205 2624 - Fax: 49 631 205 2803 - Telex: 04-5627 unikl d
e-mail: uzimmer@informatik.uni-kl.de

Based on the experiences from an autonomous mobile robot project called MOBOT-III, we found hard real-time-constraints for the operating-system-design. ALBATROSS is "A flexible multi-tasking and real-time network-operating-system-kernel". The focus in this article is on a communication-scheme fulfilling the previous demanded assurances. The central chapters discuss the shared buffer management and the way to design the communication architecture. Some further aspects beside the strict realtime-requirements like the possibilities to control and watch a running system, are mentioned.

ALBATROSS is actually implemented on a multi-processor VMEbus-system.

1. Motivation & Introduction

Why are realtime-aspects so important for our group? Before trying to find an answer to this question, we would like to give a really short overview of the goals in our autonomous mobile robot project called "MOBOT-III" in the form of a definition[1].

An autonomous mobile robot (AMR) is a system which perceives information about its environment in order to use this information for solving a given task. It has to be equipped with an onboard computer-system to do all computations independently and without external intervention. It must be able to explore unknown environments while building maps (of various kinds). Based on these maps the AMR navigates to specified goals while avoiding collisions with fixed or moving obstacles and performs local tasks like identifying and manipulating objects.

From the pool of realtime-problems in this area we will highlight three.

a. *Guarantee for actual data*
Most of the decisions have to be based on actual data. For example you might think of reflective navigation around a moving obstacle. If the sensor-information (i.e. the deduced map) for the pilot-component is older than about 100 ms (at a speed of 1 m/s), it is impossible to generate a smooth and really reflective track for the vehicle. And of course it is a great security-risk when the robot drives in a "world before our time".

b. *Graceful degradation*
In a lot of situations the breakdown of one component makes life too risky for the robot and the only answer will be an immediate stop of all motors. But on the other hand it is not the best idea to stop the robot because a part of, e.g. the object-recognition-component is beginning to hallucinate. In this kind of degradation, the other parts must be able to decide that the output of this component does not make sense any longer and (very important) the crashed part must be isolated, so that this component is not able to disturb the whole system.

c. *Keeping on-line in a running system*
Testing the robot in a real environment leads directly (usually at the first corner) to the question: "What are the actual internal maps, this (or the next) decision is based on?". One attempt may be to stop the machine from time to time and to look at the internal data. But this is not a realtime-test and of course not the comfortable way. So the optimal solution is to have an insight into the internal data-structures while these are build up in the running system in a way that the robot can not even detect this access.

The above points should be enough to show our motivation to build a realtime-system fulfilling some hard realtime-constraints.

This article will -hopefully- show a couple of significant differences to conventional realtime operating-systems.

In the next chapter we will discuss some main requirements as seen from the task's point of view. The main part follows in chapter 3. There the communication-scheme in ALBATROSS is being highlighted. First as the base of all information-transfers the so called realtime-ports are discussed in detail, then some higher communication-levels are mentioned. The question of the need for a communication-controller follows. Finally the communication-chapter ends with an opposition of transient and cyclic transfers.

[1] *for further information about MOBOT-III see the references at the end of the article*

Chapter 4 shows some general aspects of ALBATROSS, like the flexibility in different environments or the possibility to keep on-line in a running system. The final conclusion lists the central aspects of ALBATROSS and shows some relations to conventional realtime-systems.

2. Assurances for each task

In this article it is assumed that the underlying hardware is a distributed system with several processors, each of them is used by a single task. Seen from the perspective of a single task communicating with the whole (complex) system, what will be the necessary assurances a realtime-system has to give to this task.

a. *Full CPU-power*
All the CPU-time is exclusively reserved for the local task. This sounds easy but there are two main problems deduced by this restriction:

a-1 *No system-interrupts*
The OS has only access to the CPU if the local task is explicitly calling the OS. There can not be any background operation at all (e.g. interrupts for the system time, etc. pp.)!

a-2 *No communication-interrupts*
If any information is arriving via the communication-system, the local task can not be disturbed by interrupts. So the necessary transport-actions must be done by an other processor and the local task will be informed when the task is interested in the new data, not when the new data is arriving.

Why is this restriction so important in a realtime-system? Every task has to fulfil a couple of operations in an exactly specified maximal time. Unfortunately there is not a practical way to find out the maximal duration of a task under all conditions. But for sure this problem will not become easier when there is an other unknown variable you have to deal with: the really available CPU-time. So the simplest way to determine the real CPU-time for the task is to give all the CPU-time to that task!
But this is not the whole truth. Some tasks have to use interrupts to access the local hardware - what about them? In such a case the local task is not only responsible for the run-time of the task itself, but also for the interrupt handlers, installed by it. The OS must assure a maximal (and of course short) time-lack between the stop of the task and the beginning of the interrupt-handler. The task has to control (or to know) the interrupt frequency in order to prevent overflows.

b. *No direct connection to tasks on other processors*
All the tasks are synchronised implicitly via the flow of data, i.e. a task will start (over) when it gets some newer input data - an explicit trigger is not necessary! The decision to look for and to use new data depends on the internal state of the local task.
Assuming a task in the system is crashed - so the only effect seen by other tasks is the lack of newer data. *They* may decide what is to do in this situation (how important the missing information is) but there is no direct connection to this crashed task. In most cases such a connection would be disastrous.

c. *Non-blocking access to the communication-system*
Every call for new data or for an export of new outputs must successfully end within a fixed (and of course short) time - even if the communication partner is broken down or in any other possible constellation.

d. *Guaranteed actuality of all received information*
This implies guaranteed transfer times but because loss of data is a real thing in a moving system, this restriction may only be approximated by redundant transfers. And so the communication-system has to be able to transfer a new block of data more than once, without blocking any other packet.

Three and a half out of four assurances refer to the communication-system. So it looks like the communication-scheme is playing an important role in the design process of a realtime-system. All of the above constraints are fulfilled in ALBATROSS even if they are not mentioned again in the rest of the article. In the following chapter we will discuss some of the main communication-aspects of ALBATROSS.

3. Communication in ALBATROSS

When trying to satisfy all of the above assurances, there will be two (apparently) contradictory directions of the way to design the communication. Each of the two dispositions leads to standard solutions, when trying to fulfil them isolated. Before discussing the combined solution, we will show the two conventional paths.

a. *Couple two processes as loosely as possible*
Regarded from the viewpoint of one processor for one task, this assumption guarantees in a natural manner a kind of graceful degradation. The typical implementations of this philosophy cover the whole bandwidth of message-passing-systems.

b. *Assure restrictive transfer times*
The closer two processes are coupled (the fewer communication-layers are between them), the better are the time-assurances for the communication-accesses. So if you like to get a quick transfer, you will fit the two processes close together. Further (if this is still too slow) you will introduce restrictions in the communication-phases (e.g. no interrupts are allowed while reading in a communication-buffer).

It is obvious that solving the two requirements individually will not lead to the optimum. Up to here the article contains only problem-descriptions, so it is time to show some solutions in the following chapters.

3-1. Realtime Ports

Before talking about protocols, we have to define the hardware-environment. The hardware-structure is simple but effective. There are two independent processors connected via a dual-ported-RAM, i.e. the memory-domains of the processors overlay. What is the definition of a dual-ported-RAM in this context? Both processors may access the same memory-area *simultaneously*. A conflict at the level of a byte-access has to be solved with some special hardware, in a way that neither side is being blocked and a byte will stay an indivisible element. Access to the dual-ported-RAM-area is only allowed in the supervisor-mode, so only the operating-system may access this critical memory.

The realtime-transfer in ALBATROSS follows a realtime-philosophy which can be described by three short demands:

a. *Consistency*
Information may only be transferred in consistent units, i.e. you have to wait until a complete message is produced. If you do not consider this restriction, you have to define an extra protocol, assuring that there is no way to build a message with packets from different productions.

b. *Actuality*
Newer information has priority over the older one. This is a contradiction to the usual demand of keeping order, because you have to destroy old information at the level of the communication-system, if there is a newer one (of the same class) available.

c. *Availability*
Information must be available at any time. This does not mean that you have to fulfil restrictive time-limits for the producer. There is simply implied that an information is only destroyed when it can be replaced by a newer one.

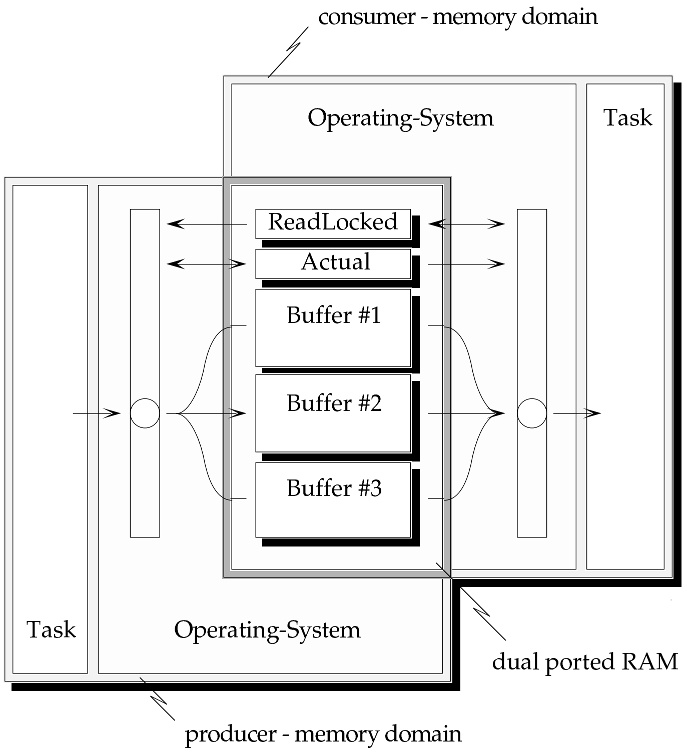

Figure 1 : memory domains

Following these demands we have designed a really simple implementation of the buffer-access as seen from the operating-system. The common base of collision-free and locking-free access with two asynchronous partners is the three-buffer structure. Figure 1 shows the connection between two processors at the hardware-level and the location of the buffers for the communication.

```
Type    BufferIndex  = (Buffer1, Buffer2, Buffer3);

Var     Actual,          {may only be written by the producer}
        ReadLocked       {may only be written by the consumer}

                    : BufferIndex;

             {    Both variables has to be one byte long and   }
             {    are located in the dual-ported-RAM area.     }
             {    The initial value of both variables          }
             {    should be "Buffer1".                         }
```

Figure 2 : common data-structures

```
Type    Actuality = (Old, New);

Function Import (Var ImportedData: Datatype): Actuality;

    Begin

        Import:= Old;

    {--- Phase 1: select a buffer for read-access}

        While ReadLocked ≠ Actual Do
            ReadLocked:= Actual;
            Import:= New
        EndWhile;

    {--- Phase 2: read-access to the selected buffer}

        ReadFrom (ReadLocked, ImportedData)

    EndFunction Import;
```

Figure 3 : buffer-access for reading

```
Procedure Export (ExportedData: Datatype);

    Var  Buffer, WriteBuffer, CopyOfReadLocked: BufferIndex;

    Begin

    {--- Phase 1: select a buffer for write-access}

        CopyOfReadLocked:= ReadLocked;

        For Buffer:= Buffer1 to Buffer3 Do
            If          (Buffer ≠ Actual) and
                        (Buffer ≠ CopyOfReadLocked) Then
                WriteBuffer:= Buffer
            EndIf
        EndFor;

    {--- Phase 2: write-access to the selected buffer}

        WriteTo (WriteBuffer, ExportedData);

    {--- Phase 3: assign completely written buffer as actual}

        Actual:= WriteBuffer

    EndProcedure Export;
```

Figure 4 : buffer-access for writing

Figure 2, 3 and 4 are the implementation fragments (in a Pascal-like syntax) for reading from and writing to the communication-area. The critical accesses to the variables "ReadLocked" and "Actual" are marked (**Bold** for a critical writing; *italic* for a critical reading).

Variables not under local control may change their values at *any* time, i.e. before you may assign the read value (i.g. of "Actual") to your local parameter, the value may have changed! This fact seems to make any formal proof of the correctness quite hard. Fortunately there is only a small number of values, the critical variables may have changed to. So it is possible to proof all the combinatorial cases step by step.

To assure the correctness of the whole realtime-transfer you have to proof the following four points in detail.

a. *Collision-free*
The producer has to select a buffer (in phase 1) which can not be used from the consumer during the whole writing-access (phase 2).

b. *Definite results*
The producer as well as the consumer selects always exactly one of the three available buffers.

c. *Termination*
The access-routines for the producer as well as the consumer have to be finished after an exact predefined time. This fact is trivial for the producer, but for the consumer a closer look is needed. The while-loop of the consumer is at most being executed two times, with the following exception: The consumer itself is much too slow, i.e. phase 1 of the consumer is executed slower than phase 1, 2 and 3 of the producer (this includes selecting a buffer and writing of the whole buffer). This limitation is tolerable, because first you are getting the actual buffer in any case, and second if the consumer is late for some reason (a long interrupt handler, or something like that) the extra loop does not mean a long time-latency in relation to the time being spent in the interrupt handler.

d. *Actuality*
The consumer gets a buffer which is, at the time of selecting the buffer, as actual as possible.

Up to here, only the access-routines at the operating-system level are mentioned. But how does this appear to the task? The syntax is really simple and the semantic is much like an electric wire. There are two special functions for each variable, so all the possibilities of range- and type-checking may be used.

For a consuming task the interface is shown in the following:

```
LookForNew<VarName> (Var <VarName>: <VarType>): Boolean;
```

as an example:

```
LookForNewRadarMap (Var RadarMap: RadarShot): Boolean;
```

The boolean result signals the actuality of the read information (Is this information ever being read before?).

For a producing task the functional interface looks like this.

```
Make<VarName>Available (<VarName>: <VarType>);
```

as an example:

```
MakeRadarMapAvailable (RadarMap: RadarShot);
```

As the final remark for this chapter once again we would like to emphasize that reading or writing in this communication-scheme is free of blocking even when the communication partner has crashed in a critical phase!

3-2. Higher communication levels

The above described realtime-transfer mechanism is well defined and easy to use. Theoretically this scheme is sufficient to construct a multiprocessor system. If you are going to design a practical system, you would not be satisfied with this kind of transfer. Well, actuality is a nice feature, but what about the "conventional" transfers with flow-control and error-reports?

So there is a need for two additional communication layers, not replacing the realtime-ports, but adding some more functionality in case you need it. In the rest of this chapter we will try to give a short overview of the higher communication levels. It is not necessary to describe the protocols in detail, most of them are well known.
The first layer is called *message-ports*, and the main functionality is flow-control, i.e. keeping order in a queue of packets, *not* destroying old information. For this purpose two realtime-ports are used (one for each direction). For the task the message-ports are as easy to use as the realtime-ports, but with a quite different semantic. You might consider the message-ports as a kind of pipelines in a UNIX-like world.

The second layer introduces some functionality for sending instructions from one task to an other and for controlling the execution. So you have flow-control *and* execution-control in this layer of the *mission-ports*. Regarding this protocol it is useful to talk about clients and servers instead of readers and writers. The server has to give a report at two points in the communication-scheme. First when it gets an new instruction, and second when the instruction is executed (successfully or with an error-message). The transfer and the execution of the instructions may overlay, so the server may accept a number of instructions, before it reports the first execution. A server can be used by several clients, but the instructions stored by the server must belong to the same client. Before a new client appears to the server, all the old instructions have to be completed.

3-3. Need for a communication-controller

Theoretically one might think of a realisation of the structure from figure 1 in form of one motherboard being equipped with both processors and the dual-ported-RAM. But it does not seem to be the practicable version, when you think of about one or two dozen processors in the whole system. In a real system one processor (or a small number of processors) will be implemented on one board.

So one of the communication-partners can only access the dual-ported-RAM via some kind of communication-system (normally a short range bus-system). This means a break in the realtime-communication scheme shown so far, because the communication is not symmetric at the physical level. One processor may access the dual-ported-RAM much like the local RAM, while the other processor has to use a communication-system to access the same dual-ported-RAM.

A new aspect appears from here on. What happens if the access to the (far) dual-ported-RAM fails, because of a disturbance on the communication-system? In the above discussion a memory-access was a local transfer and therefore without any aspects of a failed communication.

With this problem in mind we are running into a contradiction. On the one hand it is necessary to finish a transfer in a short predefined time, but on the other hand a failed access via the communication-system must be repeated. The processor for the local task is not available any longer after the first failed trial. So which processor will initiate the second trial?

The problem can only be solved by introducing a third processor as a host for the communication-controller as shown in figure 5.

From the view of the tasks any buffer-access looks like an access to the local RAM, i.e. it can be successfully done in a well defined time. The communication-controller may read from or write to the (far) dual-ported-RAMs several times, without disturbing the local tasks at all.

There is not an immediate connection between the buffer-areas of the single tasks any longer, but the communication-controller may guarantee that the transfer from one buffer-area to the other will be done with an adequate frequency. For the useful definition of this frequency see chapter 3-4 “Cyclic transfers”.

There are some additional functions a communication-controller may offer. If an information is useful for a number of consumers the communication-controller may distribute this information in the manner of a simulated broadcast. Notice that the pure realtime-transfer from chapter 3-1 is not able to distribute information - it is limited to point-to-point transfers. It should also be possible to combine the output of several producers on one “multi-producer-channel” (of course they should produce the same kind of information).

One of the most important arguments for a communication-controller is the fact that a common communication-system must be arbitrated. Thinking of a really distributed communication-scheme - i.e. there is a number of different senders - each sender has to complete an (bus-) arbitration-cycle before any communication may take place. The actual duration of one arbitration phase is principally indeterminable, there is only an upper limit (if the arbitration mechanism is “fair”). So it is quite difficult to calculate the worst case in such a system.

If there is only one participant (the communication-controller) which has to fulfil all the transfers, it may occupy the communication-media all the time. This means there is no arbitration at all and the communication-system shows a deterministic behaviour.

Another aspect is the centralized functionality. Is the centralized communication-controller a security-risk for a distributed system, i.e. what happens if the communication-controller fails, for some reason? In case that the dual-ported-memory-areas are organized in a symmetric way, each processor may play the role of the communication-controller. Further you may reserve an extra processor-system which may detect a breakdown of the bus-transfers and fulfil the communication-tasks if the main communication-controller is failed.

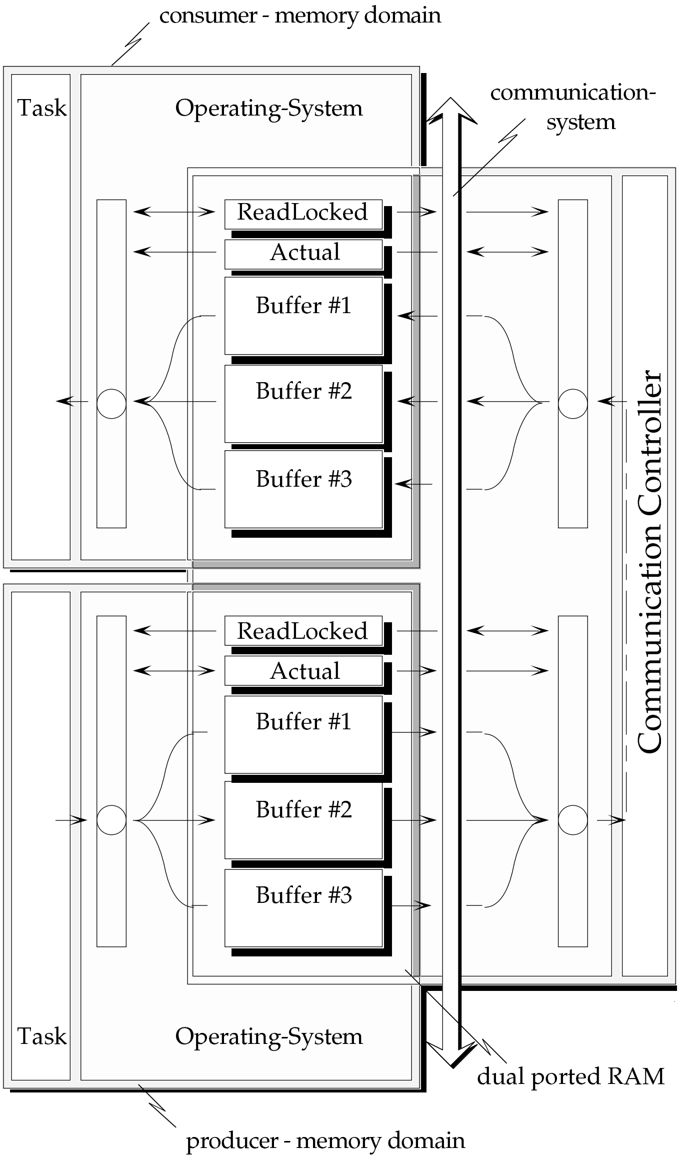

Figure 5 : the communication-controller

If you are only interested in a graceful shutdown of the system, it is not necessary to implement an extra communication-controller, but it is enough to offer a common emergency-interrupt.

3-4. *Cyclic transfers*

Cyclic transfers means that there is a pre-scheduling of the communication-slots instead of transient transfers while the system is running. The pre-calculated scheduling plan is then executed in a cyclic manner. For an efficient resource usage it is necessary to allow only powers of two from the highest frequency (i.e. the smallest time-slot) in the system.

We will highlight the aspects of this kind of transfer in the form of three questions.

What is the major problem with transient transfers?

Whenever a producer wants to distribute it's results (this happens completely asynchronously) it runs through an arbitration-phase on the communication-system. It is quite difficult to calculate the time, the process has to spend in this phase. The only way to overcome this unsure timing-knowledge is to hold out enough computational and communication resources. If you are forced to guarantee some realtime-features, this resources may become quite large.

What is the restriction to pre-scheduling?

The restriction is really simple - you have to know the maximal communication times in advance. In a realtime system the worst case conditions must be calculated. This should be possible, if you want to assure reliability of the whole system. From the worst case conditions you have to deduce the highest needed sampling frequency. This frequency is the smallest time-slot on the communication-system. All other transfers must occur as a multiple or (better) as a power of two of this smallest time-slot.

What are the advantages of a cyclic-transfer-system?

The construction is based on worst case conditions, i.e. the worst case may happen without any confusion. Additionally it is possible to do something more, a kind of over-sampling. If the maximal needed sampling frequency is known, three possibilities are implied here. First the communication-system is not able to transfer data with such a frequency. Then you have to look for some quicker hardware, or you have to relax your realtime-constraints. Second, the communication capacity just corresponds to this highest frequency, so congratulations for the configuration department. But the normal case should be, that there is some extra capacity of the communication-media, even in the worst case. In this context worst case means, the constraints of the physical system are considered, but all the computer hardware is assumed to be without any failure, especially the communication-system. So why not using this extra capacity for a number of redundant transfers? In a physical environment this may be interpreted as over-sampling. From the computer scientist's point of view this means we are using these extra resources for redundancy. The best we can do to avoid the risk of transient failures, is to use all the communication-capacity for as much redundant transfers as possible.

4. …have a closer look at some general aspects of ALBATROSS

In this chapter we will highlight two, a little bit more general aspects which are not necessarily or directly deduced from realtime-constraints. These are an easy way to keep our operating-system flexible and a construction to get an insight into the running system. There is a much longer list of features, but most of them are conventional implementations and so, outside the scope of this article.

Of course the way to handle interrupts and operating-system-traps is a critical point in any realtime-system, but there are only a small number of possibilities to get a quick response. Further these conventional implementations are extremely processor-dependant, so there can not be a general strategy, but only concrete versions for each processor.

4-1. Flexibility

We are using a quite simple way, to keep the operating-system able to adapt on a large range of several hardware-environments. The common hardware on all the allowed processor-boards must be a member of the 680x0-family and a dual-ported-RAM connected to the communication-system (for instance via VMEbus). All other components are declared in a special range of the ROM-area, the hardware-description. Each instance of the operating-system is carrying the whole range of drivers for all allowed controllers and interfaces. While starting up the system, the operating-system checks the hardware-description and installs the associated test-routines and drivers.

For the operating-system itself there are three different sources. It can be loaded from the local ROM (if it was being tested and tested and tested) or from the local serial-controller or via the communication-system from another processor-board which has already a version of the operating-system available. So the operating-system is self distributing - a nice feature in a test-environment in which the operating-system has to be updated quite often.

4-2. Keeping on-line in a running system

Assuring some realtime-features for the single task, is a necessary step in the design of a whole realtime-system, but it is not sufficient. You have to think about a possibility to monitor the tasks running under realtime-constraints.

The critical point is to have a look inside the running system without disturbing someone. It is obvious that there can not be an explicit or implicit stop-command, when observing the system.

Assuming that the needed information is being transferred sometimes via the communication-system - there is a simple, but perfect solution for this problem.

All the information on the communication-system has to be stored temporary in the local memory domain of the communication-controller. Assuming further that the local memory domain is large enough and can be shared with an other task - the supervisor. The communication-controller may store the transferred packets in the manner of a cyclic-connected list. The supervisor may also access the area of this cyclic-connected lists, so it is able to read all the communication packets, without any side-effect for the whole system.

Of course, usually it is not useful to read all the transferred information, because the user-interface is too slow to show the data in realtime, but you might think of three modes the supervisor can give useful information to the user.

First, when the system is running at full speed, the user

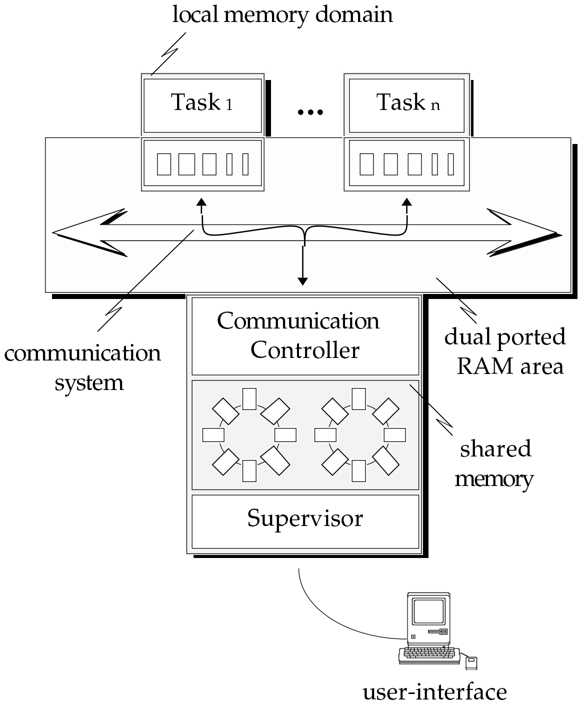

Figure 6 : the supervisor

may watch a number of samples from the real information-flow. Second, assuming that there is a fast data-logging system attached, the supervisor may record the history of the system for a specified time. And finally, if the system is crashed for some reason, the actual cyclic-connected lists in the shared memory area may be used as a post-mortem-dump.

The supervisor would not be called "supervisor", if there would be only the above described functions offered. Additionally there are some command-ports installed which are filled up by the supervisor, i.e. by the on-line-connected user.

4-3. Environment

First some words on the development environment. We are actually using Apple-Macintosh workstations for the development and testing of the operating-system itself and the distributed tasks (with all the quite comfortable features). ALBATROSS accepts Mac-object-code directly, so there is no need for a second set of development tools on the VMEbus-system. Of course there are a number of restrictions to downloaded codes. For instance you should not try to produce graphical output on the VMEbus-system, but this does not seem to be too hard.

In the actual configuration, there is a problem regarding the processor-power for the supervisor. It must be shared with the communication-controller. So if you are using all the communication-capacity, the supervisor does not get enough access to the shared processor. This is not a principle problem but it makes the user-interface quite slow. Perhaps there should be an extra processor for the supervisor to avoid this bottleneck.

5. Conclusion

Finally we will give a short collection of the main strategies used in ALBATROSS.

a. Worst case as normal case
Calculating of the worst case conditions, as given by the outer world. These conditions deduce a highest sampling frequency needed for the control of the physical system.

b. Dividing the problem
Splitting the whole problem in a number of (to a certain degree) independent tasks, synchronized via the flow of data.

c. Pre-Scheduling of the communication phases
Generating a pre-calculated (i.e. not calculated at the runtime) scheduling-plan, based on the known highest sampling frequency and the available capacity on the communication-media, using redundancy, i.e. over-sampling.

The following features must be offered by the operating-system, to assure the exact execution of the scheduling-plan.

- Locking-free shared buffer management
- An explicit communication-controller

Summarising the realtime-aspects as shows in this article, the key to realtime reliability seems to be the communication-scheme. So perhaps the often mentioned interrupts-responding times are not the whole truth in a realtime-world.

ALBATROSS

References

Brooks R.A.
A Robust Layered Control System for a Mobile Robot
IEEE Transactions on Robotics and Automation, Vol. RA-2, No. 1, 1986, pp. 14-23

Peter Hoppen, Thomas Knieriemen, Ewald von Puttkamer
Laser-Radar based Mapping and Navigation for an Autonomous Mobile Robot
IEEE International Conference on Robotics and Automation 1990, Cincinatti, Ohio 13-18 May 1990 pp. 948-953

Thomas Knieriemen, Ewald von Puttkamer
Realtime Control in an Autonomous Mobile Robot
International Workshop - Information Processing in Autonomous Mobile Robots - University of München, Germany, March 6th to 8th, 1991

Kopetz H. et. al.
Distributed Fault Tolerant Real-Time Systems: The Mars Approach
IEEE Micro Magazine, Feb. 1989, pp. 25 - 40