NIKHEF
nationaal instituut voor kernfysica en hoge-energiefysica

July, 1984

FACULTAIRE VAKGROEP INFORMATICA FWN

NIKHEF-H/84-8
FVI-A01/84-

FVI

A Distributed Real-Time Operating System

F. Tuynman

L.O. Hertzberger

Computer Science Department University of Amsterdam

Dutch National Institute for Nuclear and High Energy Physics

Submitted to Software, Practice and Experience

A Distributed Real-Time Operating System

F. Tuyrman

L.O. Hertzberger

Computer Science Departement University of Amsterdam

Dutch National Institute for Nuclear and High Energy Physics

Summary

A distributed real-time operating system, Fados, has been developed for an embedded multi-processor system. The operating system is based on a host target approach and provides for communication between arbitrary processes on host and target machine. The facilities offered are, apart from process communication, access to the file system on the host by programs on the target machine and monitoring and debugging of programs on the target machine from the host. The process communication has been designed in such a way that the possibilities are the same as those offered by the Ada programming language. The operating system is implemented on a MC 68000 based multiprocessor system in combination with a Unix(*) host.

(*) Unix is a trademark of Bell laboratories

1. Introduction

The motivation for the development of Fados is based on our experience with the use of multiprocessing systems for real-time data analysis in experiments in high energy physics, such as currently done in CERN (**). We will first give a short description of that application and the system in use at CERN.

At the NIKHEF (***) work started in 1979 to develop a multi microprocessor system for analyzing data from experiments in real-time. The time available to do this analysis depends on the actual experiment and varies from a few tenths of a millisecond to about ten milliseconds. In this short period of time only a partial analysis of the data can be perfromed. The result of this analysis determines whether the data should be preserved for further analysis or not. It is not possible to do this analysis with sufficient precision on a single microprocessor.

The Fast Amsterdam Multiprocessor (FAMP) system designed for this application, is a modular multiprocessor system based on the MC 68000 microprocessor. The two basic modules are:

- A cpu board which can address two independent busses, each with its own part of the address space of the processor;

(**) Centre Europeenne pour la Recherche Nucleaire
(***) Dutch National Institute for Nuclear and High Energy Physics

- A 16K dual ported memory which can be addressed from two busses (*).

Furthermore there are memory modules and special modules for data acquisition. A more complete description is available in[1]. The connections at bus level are illustrated in figure 1. The FAMP system is currently operational in configurations with three to seven processors.

For software development a host target approach has been used. Assembly and compilation of programs was done on a time-sharing system, whereafter the object code was transferred over a terminal line to the FAMP system. On the FAMP CPU's a modified version of the Motorola monitor MACSBUG was used. This monitor offers facilities for the loading of programs over a terminal line and machine level debugging. Extensions were added to make this monitor suitable for a multi-processor system. At first only assembler was used for programming, partly because nothing else was available, partly because the very time critical applications do not allow the overhead of a programming language. In a later stage Pascal was used for less time-critical applications.

This approach turned out to have serious pitfalls.

(*) Recently the FAMP system has been redesigned for 128K dual ported memories and the use of the Motorola standards VME and VMX for the two busses.

- It is desirable that files available on the host can be used as input for programs on the target and that programs on the target can write data logging information on files on the host.

- During the experiments, which run continuously for several month, it is necessary that the behaviour of the target machine is checked by an operator and that there are facilities for intervention. When several targets are in use there is a need for one single system taking care of the interface with the operator instead of having a separate interface (physically as well as logically) for every target machine.

  Especially when the number of processors is large, the time to load a program over a terminal line becomes excessive. Load times of half an hour our common.

Because of these problems it was decided to make a better programming and support environment for the FAMP system in particular and embedded real-time systems in general.

2. Requirements for a real-time programming environment

There are two different strategies to set up a programming environment for real-time systems. One is to use a seperate machine (the host) for editing and compilation of programs and then transfer the object code to the target machine. The alternative is to use the same machine for program development as will be used for the real-time application.

For software development a time-sharing operating system is by far the prefered choice, certainly if several people cooperate on the software development. This means that the consequence of combining development and real-time work is that a time-sharing operating system should be installed on the computer used for the real-time application. It was decided to stay with a host-target approach for two reasons. First there are serious practical difficulties in installing a time-sharing system on specially designed hardware, such as the Famp multiprocessor system. Secondly a development system imposes different requirements on as well hardware as software. Memory management, terminal(s), and disk(s) are indispensable, while many real-time applications have no need for them. The result is that after program development is finished a large part of the system remains unused. As far as the software is concerned it is difficult to guarantee that the development work will not interfere with the execution of the real-time programs. If this guarantee cannot be given then the system cannot be used for further software development once operational.

In the host target approach as described above the main problem is the poor interaction between the two systems. Therefore we looked for a way to combine the advantages of separate functions of target and development system, but not the disadvantages of a bad attunement. We have set the following requirements for the programming and user support environment for real-time systems to be developed:

1 The real-time operating system should be easily portable to different target machines and it should not block new hardware developments.

2 It should be possible to monitor and operate the target machine from a remote time-sharing system.

3 Programs on the target machine have to have access to the file system on the host.

The operating system should support several high level programming languages.

5 Special tools for symbolic debugging are required.

6 There should be a sufficiently fast connection between host and target. A local area network connection is the preferred choice.

Below a rationale for each of these points is given.

1) Especially ease of portability to other target machines is important since it is likely that the target machine will be changed rather frequently, e.g. to use an other type of microprocessor.

2) Currently it is customary that a real-time system has a direct interface with the user. We believe that this is neither necessary nor desirable. To construct a good user interface in itself is difficult enough and a real-time system is a considerably less suited environment than a

timesharing system. A better solution is for the real-time system to offer only basic facilities accessible over a local area network. One can then choose for the user interface a machine on which modern techniques for human-machine interaction can be realized (using windows, graphics, etc.). Also for the monitoring of the real-time system during production runs it is important that the interface with the operator is handled by another system. The interference with the real-time tasks is then reduced to a minimum and an operator interface can be constructed with which a number of independent systems can be monitored simultaneously.

3) Since program development is done on the host it is necessary that programs on the target can use files on the host for test input and save their output on the host. As soon as the target is operational access to a remote file system is also required to save data and for diagnostics. A local file system is only then necessary when the data cannot be transported fast enough to a remote file system. Furthermore it is not recommended to have a disk as a part of an embedded system unless that is absolutely necessary. The mean time between failure for a disk is considerable higher than for electronic components. This results in a decrease of the reliability of the system when a local disk is used.

4) When using an ordinary high level programming language to

write real-time paralel programs, it is necessary that facilities for inter process communication and interrupt handling are added. At first sight it would therefore be preferable to use a programming language where these facilities are built in. These languages (e.g. Modula, Concurrent Pascal, Ada), however, are not suitable for a multiprocessor system without a common memory that can be accessed by all processors, since they all allow different processes to access common data[2,3,4]. That is why it was decided not to support any of these languages specifically but to provide for inter process communication and interrupt handling by procedure calls to a nucleus.

5) A problem encountered when real-time programs are debugged by setting breakpoints and analyzing the memory image is that the setting of breakpoints can interfere with the correct functioning of the program. To solve this problem special adaptations are necessary. This can be done by setting the breakpoints per process so that the processes that are not time critical can be debugged without disturbing the time critical ones.

6) The use of a separate system for program development requires a good connection, logical as well as physical, with the target. What is needed is a high speed message oriented (and not character oriented) connection. Principally a character oriented connection can be used to transfer messages, but in practice such lines are always

slow (9600 baud, and usually less). Even if a higher speed is hardware wise possible then the overhead in the host for each character transfer will prohibit the use of this speed. This can be solved only by the use of a local area network controller capable to transfer a large amount of data at a high speed without assistance from the host cpu.

3. An overview of the FAMP distributed operating system (Fados)

The operating system is based on the idea that there is a nucleus which performs, together with a group of system processes, the functions traditionally performed by the kernel. Communication between system processes is performed by the exchange of messages. A crucial point in our design is that the inter process communication on the target has been integrated with the one on the host. The resit is that the host target interface is transparent for inter process communication. This has two major advantages:

- A large degree of freedom is obtained in the choice which machine (host or target) has to perform a specific task.
- The user interface to the target can be entirely provided for by processes on the host and embedded in the user interface of the host computer.

The design of an operating system on the basis of communicating processes has received attention before[5].

The most important advantage is the simplicity of the design. The resulting communication overhead between processes has always been considered the main disadvantage. For a multi-processor system, however, it is an inevitable choice since a large part of the activities of the operating system have to be performed locally and consequently the operating system has to be partitioned in communicating parts. As far as the part that is local per processor is concerned, the choice remains whether that is designed monolithically or divided in cooperating processes.

The structure of our design for the operating system is largely determined by the interaction between host and target. Three main areas of interaction can be identified.

1 The code generated on the host has to be transferred to the target. The programs on target and host which take care of this have to be capable to load a group of processes, possibly on different processors.

2 Programs on the target have to have access to the file system of the host.

3 For symbolic debugging of programs on the target it is convenient to have the interpretation of the memory image and the information about names produced by the compiler performed at the host. A process on the target will then only have to handle low level operations like the setting of breakpoints at physical

addresses and the transfer of (parts of) the memory image to the symbolic debugger on the host.

Apart from the interaction with the host the structure of the operating system is also largely determined by the requirement that it should function on hardware without memory management. Due to the absence of memory management it becomes impossible for the operating system to allocate pieces of memory to a process and guarantee that those pieces can be used only by the process to which they were allocated. Also it becomes inevitible that when object code is linked after compilation the actual piece of the memory is determined where the process will execute. This implies that the possibilities for using a memory allocation mechanism in the operating system are limited. Consequently, we have decided not to include any memory allocation mechamism. This greatly simplifies the structure of the operating system.

The operating system we have developed consists of the following parts:

On every processor of the target:

- A nucleus which provides process communication and the transfer of interrupts to user- and system processes.

- System processes which
  - load a user process into memory and start it: the "Load process".
  - handle breakpoints, addressing errors etc.: the "Debug Monitor".

On the host:

- a "command interpreter"
- a "debugger";
- a "file server".

The communication between these processes is pictured in fig 3. The nucleus on every target processor provides for the message exchange between processes on the target machine with one another and with processes on the host.

We have chosen for a 68000 based micro computer as host with Unix as operating system. Since the 68000 processor is also used in our multi processor system it is possible to use the compilers and program generators available on the host for the target as well. The host is connected to the target by means of a parallel interface to a dual port memory that is part of the target. This interface enables the host to address the dual port memory in the same way as it addresses it's own memory. This interface was developed when local area network controllers were not yet available. The

consequence of coupling the target and host by a dual port memory is that they have to be whithin a distance of two yards. This has several practical problems and therefore we intend to replace the dual port memory link between target and host by Ethernet (or an other local area network of similar performance) in the near future.

The Unix version which is used now, Bell Labs version 7, does not offer the possibility to make process communication between processes on host and target transparent without modifications to the Unix kernel. Furthermore it is also impossible to have several user processes communicate concurrently through the same physical link with the target. We have solved this by installing a small number of logical channels as different devices. This solution was relatively simple to realize. Our current implementation is realized in such a way that a 'read' operation on the device with which one is connected to the target returns an entire message, whereas a 'write' operation transmits an entire message. Depending on the device written to, the source field in the message is filled in. A read operation only returns messages with the correct destination address. It is also possible to send a message to one of the other devices. Hereby the processes on Unix can also communicate with one an other. In the near future we hope to switch to a Unix version with inter-process communication facilities such as System V or Berkeley 4.2[6,7].

Most of the software is written in C, with the execption of the entry of the nucleus after a system call or an interrupt, which had to be done in assembler. There were two arguments to use C, first because it is relatively simple to handle pointers on machine level, secondly because C has a direct interface to the UNIX system calls, wich is important for the programs on the host. Analysis of the code produced for the nucleus, where performance is critical, has demonstrated that with optimizations on the assembler level less than twenty percent increase in speed and decrease in size could have been achieved.

4. The Nucleus

The main task of the nucleus is to provide for inter process communication and the scheduling of processes. The primitives offered by the nucleus are intended for direct use by the programmer. To guarantee ease of use we have tried to model these primitives as closely as possible after those offered by a well structured programming language with facilities for multi tasking. First an analysis of the possibilities for doing so are given.

There are two inherently different strategies for inter process communication. The first one is characterised by separate communication and synchronization operations and communication through shared data. This can be implemented by semaphores, monitors, etc. In the second strategy process communication and synchronization are combined.

This can be thought of as the exchange of messages, where synchronization is done by waiting for a message and upon arrival of the message its content is available to the receiver. There are several programming languages which offer facilities for inter process communication. There is however no direct correspondence between the definition of process communication at the level of the programming language and one of the before mentioned strategies. Each strategy can simulate the other. Consequently by defining a syntactic construct an implementation of either of these strategies is not implied. In some situations implementation of one of these strategies can be considerably more efficient than the other. If the hardware makes it possible for processes to address an area of common memory, which is the case on a single processor system and on a multi processor system with a single common memory, then implementation of message passing on top of a semaphore/shared data facility is not or hardly less efficient than a direct implementation. On the other hand implementing shared data on top of message passing will lead to a important loss of performance. If the hardware does not allow processes to address an area of common memory then message passing is the lowest level of communication.

If a syntactic construct which can be implemented efficiently with message passing does not pose any restriction on the strategy for process communication, whereas a syntactic construct which cannot be viewed as a

form of message passing requires the presence of a nucleus offering a semaphore/shared data facility. Concurrent Pascal and Modula are examples of languages which can be implemented on top of a message passing nucleus only with a important time penalty. This in contrast to the Ada rendez-vous mechanism which resembles Hoare's[8] input/output commands (CSP)[9] and can therefore be efficiently implemented on top of a message passing nucleus.

The Fados inter process communication

We decided to offer message passing only. An important argument is that on the FAMP system communication through shared data is only possible between processes on the same processor or on two directly coupled by a dual-port memory. Since we wanted to experiment with different allocations of tasks on processors, we wanted the process communication to be independent of this allocation. Apart from that, it might be more attractive in a multi processor system to avoid the overhead of scheduling operations by busy polling on semaphores in a common memory. For doing so no support from the nucleus is needed. Since the primitives for process communication will be used directly by the programmer it is important that the primitives themselves encourage the use of a well defined communication protocol between the processes.

The alternatives we considered were CSP and Ada. Since Ada offers some additional features we decided to implement the

Ada rendez-vous mechanism as completely as possible to see how well it could be used on a multi processor system.

Process (Ada: task) communication is done in Ada by means of entry calls and accept statements which can be combined with select and when statements[4]. When a process executes an ordinary entry call it will be stopped until the process containing the corresponding accept statement has executed that accept statement. During such a 'rendez-vous' the in parameters of the accept statement are copied from the calling to the called process and the out parameters vice versa. To realize this with message passing the entry call can be considered the sending of a message containing the in parameters of the accept statement followed by the waiting for a message containing the out parameters of the accept statement. The accept statement corresponds to the waiting for a message at the start of the statement followed by the sending of a reply at the end of the accept statement. To do this the following three message passing primitives are available:

```
send    - send a message and wait for a reply;
receive - wait for a message;
reply   - send a reply.
```

These primitives, though with somewhat different arguments, have also been described by W. M. Gentleman[10].

An entry call in Ada requires the specification of both the task containing the accept statement as well the name of the entry. A task containing several entries can specify on

which entries an entry call will be accepted, by combining the select statement with accept statements. To realize this with messages it is necessary that when a message is send the destination process is specified, (equivalent to the specification of the task's name in the entry call) and that additionally the message is given a name similar to the name of the entry in Ada. When receiving a message there has to be a possibility to specify a list of names of messages which can be received at a particular place in the program. This is implemented by means of a bit map. The name argument of the send call is a 32 bit integer in which one bit may be set. The receive call has an integer argument that should contain a set bit for every message name to be accepted by that receive call.

Apart from the ordinary entry call Ada also has a conditional and a timed entry call. A timed entry call specifies the maximum period of time that can be waited whether the entry call will be accepted. A conditional entry call can be considered a timed entry call where the period that will be waited equals zero. For the realization of these two entry calls an extra parameter is necessary in the send call to specify the maximum time between transmission and acceptance of the message. Furthermore there exists in Ada a selective wait statement which makes it possible to limit the time that will be waited for an entry call. This requires that the receive call has a parameter to specify the maximum time the process will be

blocke The size of send and reply message is variable below a certain maximum (512 bytes). Currently it is not possible to send messages larger then the maximum physical message size.

This results in the following definition of the system calls (in the programming language C);

```
send - send a message and wait for a reply message
send (dest, name, timeout, msg, nbytes, rcvbuf, rcvbytes)
TASKID dest;      /* destination process */
int    name;      /* name of the message (one bit set) */
int    timeout;   /* maximum time a message will be queued before it is
                     accepted */
int    *msg;      /* address of message to be send */
int    nbytes;    /* size of message to be send (max 512) */
int    *rcvbuf;   /* address of buffer for reply */
int    rcvbytes;  /* size of buffer for reply */

receive - receive a message
receive (name, timeout, buffer, nbytes)
int    name;     /* bit map of names of messages to be received */
int    timeout;  /* maximum time the process will be blocked */
int    *buffer;  /* address of buffer for message */
int    nbytes;   /* size of buffer for message */

reply - send a reply message
reply (name,msg,nbytes)
int    name;     /* name of the reply message */
int    *msg;     /* address of reply to be send */
int    nbytes;   /* size of reply to be send (max 512) */
```

The name of the reply call has no further meaning but can be used to transfer an extra four bytes of information, for example to signal error conditions. When the receive or send call returns the size and name of the message are available in global variables. Messages which arrive before a receive call are queued. The same happens to messages whose name does not occur in the argument of the receive call. If the message is sent by a process on the same

processor no buffer space is needed since the message can remain in the data space of the blocked sender. If the message is sent by a process on an other processor the message remains buffered in the dual-port memory. The time that a message will remain queued is determined by the timeout argument of the send call. When this time is passed the nucleus returns a timeout message. The send call then returns the appropriate error code. A reply message is always sent in reply to the last arrived but not yet answered message. Hence there is no need for a destination field in the reply call. The destination address which has to be given in the send call consists of two parts, a processor identification and a local process number. Consequently the nucleus does not have to have knowledge about which process runs on which processor. During startup of a group of user processes the address of each is made available to all processes (see below). Server processes have fixed addresses. In figure three the code of a buffer process is given in Ada and in the language C using the primitives described here.

Interrupt handling

On the machine level an interrupt is an interruption forced by an external device of the running program followed by a subroutine call to an address indirectly supplied by the device. From a structured programming point of view this is highly unsatisfactory. The alternative is to let processes

wait for interrupts. The mechanism chosen for process synchronization will usually determine the way this is realized. If processes synchronize by means of semaphores it is natural to regard an interrupt as the freeing of a semaphore. A process will then wait on a semaphore of which is known that it is released by an interrupt. If synchronization is done by means of message passing an interrupt can be regarded as an empty message. In both cases the external device is regarded as an external process and an interrupt as a synchronization operation between processes.

Since we have followed the Ada philosophy with the process communication there is no reason to treat interrupts differently. In Ada interrupts are regarded as entry calls. Depending on the implementation and the type of interrupt such an entry call may be an ordinary, a timed or a conditional entry call. Interrupts which are queued correspond to ordinary entry calls and interrupts which are lost in case they cannot be immediately accepted correspond to conditional entry calls. An accept statement which is executed in response to an interrupt has a higher priority than other tasks.

An interrupt in our system is regarded as a send call executed by an external device. A reply message will not be sent in this case. Before a process can receive interrupts as messages it has to use the system call

```
recint (vector, name, timeout)
int vector;      /* number of vector */
int name;        /* name of message generated after an interrupt on vector *
int timeout;     /* time the generated message will be saved */
```

to inform the nucleus that it wishes to receive interrupts on the named vector as messages with the given name. After the call recint, interrupts on that particular vector will be translated by the nucleus into send operations with as destination the process which issued the call. The timeout argument specifies the time that the message associated with the interrupt should be stored for acceptance. When the timeout argument is zero interrupts are lost when the process for which they are intended is not waiting for them. See figure 4 for a comparison with Ada.

A process which handles interrupts should have a hardware priority which equals or exceeds the priority of the interrupts it handles. This is to avoid that it will be interrupted by interrupts it should handle itself. When an interrupt arrives the process for which it is intended is started immediately if it was waiting for it. That the interrupt is received by the processor implies that the running process has a lower priority.

Scheduling

The scheduling algorithm has been kept simple. A process will continue to execute until it decides to wait or until an other process with a higher priority wakes up. When a

process is started the priority can be stated. Priorities remain fixed during execution. The priority of a process consists of two parts, the hardware priority which determines the priority level of the processor and a software priority. The hardware priority is only relevant for processes which handle interrupts. A process with a higher hardware priority has a higher general priority. Processes with equal priority are scheduled round-robin.

5. File operations

The operations on files have to be realized in our system by messages to a file server on the host. This requires attention in three areas.

- The protocol between target program and file server.
- The connection between the i/o routines of the high level language used and the protocol with the file server.
- The installation of the file server on the host computer.

We wanted to use the compilers available on our Unix development system also for our multiprocessor system. These compilers have an i/o library which uses the Unix file system calls. To avoid that the i/o library routines for the different languages had to be adapted it was necessary to simulate the Unix system calls on the target. It was

decided to take the Unix file system calls as the basis for the protocol with the file server. This leads to a straight forward implementation of the file server and of the routines to simulate the Unix system calls on the target.

In Unix the interface with the i/o system call is formed by a group of C callable subroutines which actually do the system call. These routines are available in a standard library. When a program is compiled for use on Fados a modified version of this library should be used. In this modified version of the standard library the routines for the assembler interface with the system calls are replaced by routines implementing the protocol with the file server.

Unix dependencies

The strategy described above leads only on a few points to a Unix dependent protocol because most of the Unix system calls are very general and can be supported easily by a file server on any byte oriented machine. The reason is that the only file type Unix supports is an array of bytes with no record structure imposed on it. This file type is generally supported (e.g. on VAX/VMS)[11]. support apart from this file type also a variety of record structures in files. From the ten by the file server supported system calls the following can be easily supported on a non Unix system:

open; close; creat; read; write; lseek; access.

Creat makes a new file, lseek positions the file pointer.

The following calls, however, cannot be supported on systems other than Unix:

ioctl; stat; fstat.

Ioctl is used to set terminal options. This call is used to find out whether standard output is connected to a terminal. If that is the case output to that file will not be buffered by the program but written to the terminal immediately. This is certainly a useful function but when an operating system different from Unix is used it has to be implemented in a different way, since the call ioctl will be very hard to simulate correctly. The calls fstat and stat collect all system information about a file. These calls are not used by the C i/o library routines but only by the ones for Fortran. The Fortran i/o library uses them for two purposes. The first one is to discover whether the file pointer of a particular file can be positioned, in other words whether the lseek call will function. The only routine which uses fstat for this purpose is canseek, which should be adapted when a non Unix system is used to run the file server. The second reason the Fortran library uses stat and fstat is for the implementation of the INQUIRE statement, which passes all sorts of information about a file to the Fortran program. It seems questionable whether it is worth the effort to adapt the Fortran library in such a way that the INQUIRE statement will also work with a non Unix machine as file server.

6. Loading a group of processes

We have assumed that use would be made of a programming languages without built in support for the execution of parallel processes on a multi processor system. To write in such a language a program consisting of several parallel processes requires some contriving. Every process must have its own instance of the library routines since those have to be available in the local memory. This makes it a natural choice to link each process separately of the others which means to treat it as an independent program. If several processes are executed on the same processor this leads to a duplication of code but this is inevitable if one wants to determine at the moment of execution how processes should be distributed over the processors. The only information a process needs about the other processes is the addresses of those processes, to be used in the send calls. This information is not available before the programs are loaded into memory since only then the nucleus will return the process identification to the parent process. When a group of processes forming a single program are loaded the addresses of all processes are passed dynamically to all. These address are placed in a global array named task. The nuclei have no knowledge about processes on other processors. Apart from the addresses of the other processes also the file descriptors of standard input, standard output and standard error are passed.

In a multiprocessor system no guarantee can be given that all processes will start simultaneously. To avoid that a one process will already attempt communication with an other process that is not yet loaded it is necessary to do the loading on two phases. In the first phase the processes are made known to the nuclei of the processors, so that at the end of this phase all processes exist. In the second phase all process addresses which are now known can be passed and the processes can start to execute freely. Which process then starts first and what time it still takes for the last one to get started is no longer important.

Implementation

On Unix there is a command interpreter which obtains as input a list of executable files which are to be loaded as a group of cooperating processes (see figure 1). With every file the necessary stack space, hard and software priority and the processor where it is to be loaded are specified. The command interpreter then sends successively for every process to be loaded a message to the "Load process" on the appropriate processor. Such a load process is present on every processor. This load process waits for command messages containing the name of the file to be loaded, the necessary stack space and the priority. The file to be loaded is then read into the memory of the processor and then a new process is created by a call to the nucleus. As soon as this process is created it is placed in the ready

queue and can start to execute. The first code of a process is the same for everyone and the first thing this new process does is to wait for a message containing the addresses of the other processes and the file descriptors for standard input, output and error. The Load process returns the address of the new process to the process who issued the command to create it. After all processes are started this way and all are waiting for a message the command interpreter will send a message containing the addresses of the other processes and the descriptors of the files opened for those processes by the command interpreter.

7. Debugging on Fados

The interpretation of the memory image in terms of the names that are used in the high level program can be done on the host. This makes it possible to use the symbolic debuggers of the host (if available). On every target processor there is a system process that waits for the occurrence of bus/address errors, breakpoints and commands from the debug program on the host. When a bus/address error occurs or a breakpoint is reached a reply is given to the Unix debug program. This debugger can subsequently send requests to the debug monitor on the target to return system data about processes, saved registers or parts of the memory image. When a bus or address error occurs all user processes on that processor are stopped since due to the absence of memory management no guarantee can be given that their text

and data segments have been violated. When a breakpoint is reached only the process in which the breakpoint is placed is stopped.

The commands which can be given by the debugger on Unix to the debug monitor on Fados are:

- start or stop a process.
- Send a message when a bus/address error occurs or when a breakpoint is reached.
- Return or write system status information (Ready queue, etc.).
- Return or write data in memory.

These command are similar to the facilities offered by the ptrace system call in Unix. This implies that it is relatively simple to interface an existing debugger to Fados. Currently the low-level facilities are directly available to the user. It is, however, inteded to interface them with a symbolic debugger, as soon as one will become available for our compilers. It is expected that most work in adepting an existing debugger will be needed to allow for the debugging of several processes simultaneously.

8. Timing Results

We will give here the results of the timing of the process communication and the access to the file server. In table 1

the time needed for the exchange of a message and a reply are given for different sizes of the message and the reply. The time is given for the two processes allocated on the same processor and for two processes executing on two processors connected directly by a dual port memory. The measurements have been done with 8 Mhz 68000 processors and memory without wait cycli and without refresh.

Timing of message exchange (send followed by an immediate reply) between two processes.
All times in milliseconds (error +- 0,02 ms), sizes in bytes:

send size	reply size	time (A)	time (B)
0	0	1,04	2,24
3	3	1,08	2,28
512	0	1,53	3,40
512	512	2,03	4,23

A) on the same processor
B) on two different processors directly connected by a dual port memory

A multiprocessor system with a single common memory is a more common architecture the the one used in the FAMP system. Therefore the timing when the two processes are allocated to two different processors directly connected by a dual port memory is also given since it gives the time needed when a common memory is used. As can be seen the copying of 512 bytes once (the difference between rows 4 and 6 column A) takes .5 ms. The figures of column B are roughly twice as large as those of column A. This has two causes. One is that the copying has to be done twice, once to the dual-port memory and then from the dual-port memory

to the local memory of the second cpu. This alone however does not account for the doubling of time when sending empty message. This is caused by the additional execution of the code to put a message in a dual-port memory. This code has to identify in which dual-port memory the message should be placed, link a buffer from the free list of the dual-port memory, and copy the message plus header.

The response time for an interrupt is .5 ms. This is roughly half of the time needed for a message exchange between two processes on the same processor.

The time to read in a block from a file is entirely determined by the response from Unix. The time needed by the message passing is roughly 3 ms (compare table 1) whereas with our Unix implementation a program on Unix needs 20 ms to read a block (of 512 bytes) from a file. This is also the time a program on a Fados processor needs to wait for a block of data from the disk.

With non-blocking messages an increase in speed can be achieved, especially when the overhead of scheduling is thereby reduced or even avoided. This is however only then true when sender and receiver are carefully tuned, since otherwise the user would actually have to do the handshake which is now done automatically. We do not think that for real-time applications this increase of speed is worth the expense of safety. If one is willing to sacrifice safety for speed it is much more attractive to stop using the

standard inter process communication entirely and use the possibilities of the hardware directly, i.e. the common memory used as common data and for synchronization busy waiting to avoid scheduling overhead. In this way a considerable speed improvement can be achieved.

The time needed for communication is such that if a process does not spend most of its time communicating only once every 5 to 10 ms a message can be sent. If a particular process requires more frequent communication of small amounts of data, a couple of words, then the message operations described here cannot be used and instead direct use of the common memory by the process seems mandatory.

9. Conclusions

The main conclusion is that with the approach described here, where an integration between process communication on host and target is achieved, it is possible with a relatively small effort to make a real-time operating system for a embedded system which can be easily extended, operates on simple hardware, and is from a point of user friendliness in no way inferior to a combined system for development and real-time execution. It can be easily extended by adding new processes, either on the host or on the target. The initial experience is that especially the ease of transferring programs towards the target machine and the access from the target to the file system of the host are very valuable and considerably decrease the barrier to use a

dedicated system. The transparency of communication between different processors is valuable for experimenting with different allocations of tasks over processors. The system is easily portable to different target systems, since the code to be transported is small (roughly 800 lines) and written in a high level language.

As has been demonstrated, the message passing primitives described here are sufficient for the implementation of the Ada rendez-vous mechanism. Unfortunately, this does not imply that Ada can be used for a multiprocessor system without common memory. Ada allows common data for different tasks and this can only be implemented with reasonable efficiency on a mulitprocessor system with common memory. Secondly, Ada allows different tasks to call global routines, e.g. those in the IO package. This also will be hard to implement in a satisfying way on multiprocessor system.

Nevertheless we think that the possibility to experiment with Ada like task communication in a multiprocessor system is valuable. It makes it possible to investigate the usefulness of Ada like task communication in a multi processor system whithout being burdened by the problems of a full Ada implementation for such a system.

References

1. L. O. Hertzberger, "The Fast Amsterdam Multiprocessor (FAMP) system," Computer Physics Communication Vol. 26, pp.79-98 (1982).

2. P. Brinch Hansen, "The programming language Concurrent Pascal," IEEE Trans. Software Eng. Vol. 1(2), pp.199-207 (june 1975).

3. N. Wirth, "Modula: A programming language for modular multiprogramming," Software-Practice and Experience Vol. 7(1), pp.3-35 (jan 1977).

4. Reference Manual for the Ada Programming Language, ANSI/MIL - STD 1815A, Alsys, France (jan 1983).

5. Marvin H. Solomon and Raphael A. Finkel, "The Roscoe Distributed Operating System," pp. 108-114, dec in Proceedings on the 7th Symposium on Operating System Principles (1979).

6. Unix Programmers Manual, System V, Bell Telephone Laboratories (1983).

7. Unix Programmers Manual, 4.2 Berkeley Software Distribution, University of California, Berkeley (August, 1983).

8. C. A. R. Hoare, "Communicating Sequential Processes," Comm. of the ACM Vol. 21(8), pp.666-667 (1978).

9. Jim Welsh and Andrew Lister, "A Comparative Study of Task Communication in Ada," Software-Practice and Experience Vol. 11, pp.257-290 (1981).

10. W. M. Gentleman, "Message Passing between Sequential Processes: the Reply Primitve and the Administrator Concept," Software-Practice and Experience Vol. 11, pp.435-466 (1981).

11. VAX/VMS Manual, Digital Software.

Caption for figure 1

The connections between Famp processors at bus level.

Caption for figure 2

[1,2] see paragraph 7 for a list of commands [1] from the debugger and a list of responses [2] from the Debug Monitor

[3] stop or kill a process (argument is a process_id).

[4] Debug Monitor's confirmation of [3]

[5] start a process (arguments are an executable file, stacksize, priority).

[6] return process_id of a started process.

[7] read the file named in [5] in core.

[8] data from the file returned by the file server.

[9] the command interpreter opens standard input, standard output en standard error for a process to be started.

[10] returned file descriptor.

Caption for figure 3

The code of a buffer process in Ada and in the language C with the Fados communication primitives.

Caption for figure 4

Interrupt handling in Ada and in C with the Fados primitives

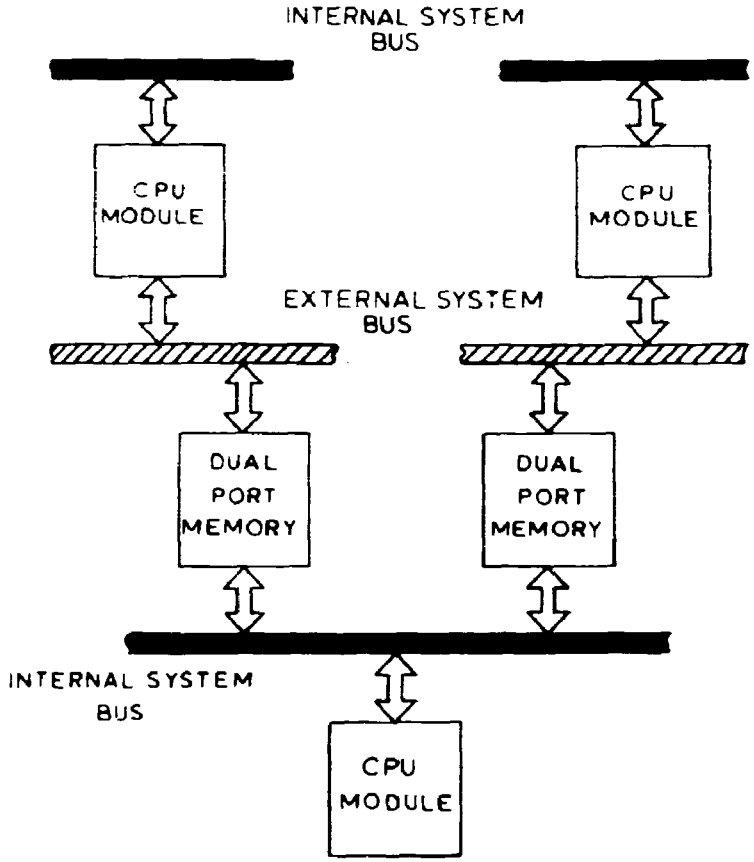
INTERNAL SYSTEM
BUS
CPU
MODULE
CPU
MODULE
EXTERNAL SYSTEM
BUS
DUAL
PORT
MEMORY
DUAL
PORT
MEMORY
INTERNAL SYSTEM
BUS
CPU
MODULE

Fig. 1

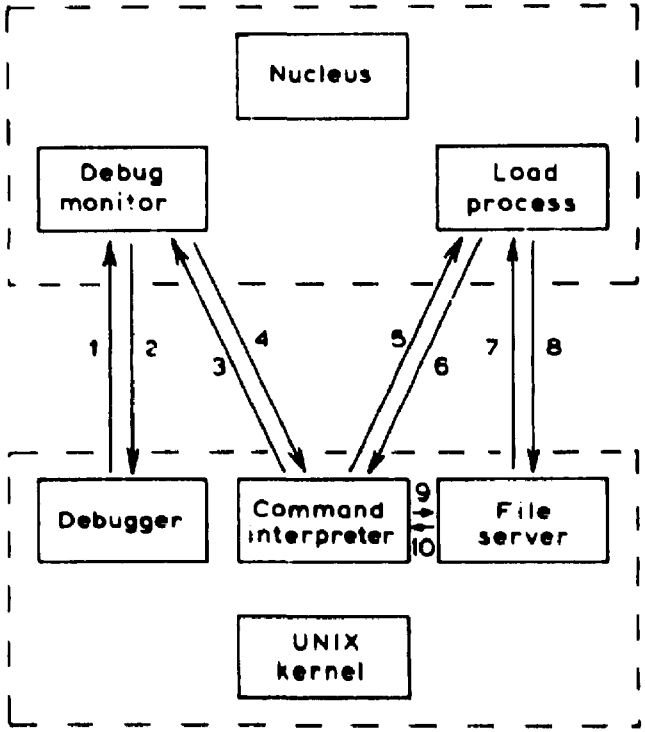
Nucleus
Debug monitor
Load process
1
2
3
4
5
6
7
8
Debugger
Command interpreter
9
10
File server
UNIX kernel
in ROM in every target processor
UNIX host

Fig. 2

FIGURE 3: Ada - C example 1

C version:

Called task

```
#include <fados/nucl.h>          /* file with extern declarations */
#define   POOL_SIZE   100
#define   READ        1
#define   WRITE       2

char pool[POOL_SIZE], c;
int count     =0;
int in_index  =0;                  /* arrays in C from 0 to size-1 */
int out_index =0;

main ()
{
   while ( 0 == 0) {
      if ( count < POOL_SIZE && count > 0)
         receive (READ+WRITE, -1, &c, 1); /* &c means address of c */
      else if (count == 0)
         receive (WRITE, -1, &c, 1);
      else
         receive (READ, -1, NIL, 0);
      if (m_name == READ) {
         reply (0, &pool[out_index], 1);
         out_index = (out_index + 1) % POOL_SIZE;
         count     = count - 1;
      } else {
         pool [in_index] = c;
         reply (0, NIL, 0);
         in_index = (in_index + 1) % POOL_SIZE;
         count    = count + 1;
      }
   }
}
```

Calling task

```
#include <fados/nucl.h>          /* file with extern declarations */

main ()
{
      char c;
      .
      send (task[0], -1, READ, NIL, 0. &c, 1);
      .     /* see section 6 for the explanation of the task array */
}
```

Ada version:

Called task

```
task buffer is
   entry read (c: out character);
   entry write (c: in character);
end;

task body buffer is
   pool_size    :constant integer:=100;
   pool         :array(1..poolsize) of character;
   count        :integer range 0..poolsize :=0;
   in_index, out_index :integer range 1..poolsize :=1;
begin
   loop
      select
         when count < poolsize =>
            accept write(c: in character) do
               pool (in_index) := c;
            end;
            in_index := in_index mod pool_size + 1;
            count    := count + 1;
      or when count > 0 =>
            accept read(c: out character) do
               c := pool(out_index);
            end;
            out_index := out_index mod pool_size + 1;
            count     := count - 1;
      end select;
   end loop;
end buffer;
```

Calling task

```
task body caller is
   c        :character;
begin
   .
   buffer.read (c);
   .
   .
end caller;
```

Fig. 3 Continued

FIGURE 4: Ada - C example 2

C version

```
#define   DONE   1

main ()

{
   int   i;

   recint (0x40, DONE, 0)          /* 0x40 is 40 hexadecimal */
                                   /* no qeueing of interrupts */
   while (0 == 0) {
      receive (DONE, -1, NIL, 0);
         /* do whatever is necessary */
   }
}
```

Ada version

```
task interrupt_handler is
   entry done;
   for done use at 16#40#;
end;

task body interrupt_handler is
   I    : integer;
begin
   loop
      accept done do
      end
      /* do whatever is necessary */
   end loop
end interrupt_handler;
```