# A coprocessor for hardware multitasking support

Verschueren, A.C.

*Award date:*
1987

Eindhoven University of Technology
Department of Electrical Engineering
Group Digital Systems (EB)

# A COPROCESSOR FOR HARDWARE MULTITASKING SUPPORT

Master's thesis
by A.C. Verschueren

This is the final report for the graduation work done at the Eindhoven University of Technology, department of Electrical Engineering, group Digital Systems

| | |
|---|---|
| By: | A.C. Verschueren |
| Coach: | prof. ir. M.P.J. Stevens |
| Time period: | may 14, 1986 to august 27, 1987 |
| Report date: | august 7, 1987 |
| Location: | Eindhoven University of Technology, Room EH 10.8 |

**Abstract:**

This report is the result of a brainstorm that lasted for over a year. It started with a simple question posed by my coach, Prof. Stevens:

**'Is it possible to build a multitasking operating system in hardware ?'**

A question from which I now think that it can be answered with a firm 'yes, and even more than that'.

Following a comparative study of several (real time) multitasking operating systems, I have conceived the basic design of an integrated circuit that offers the functions of a multitasking operating system to a computer if it is connected to this computer's processor.

The proposed chip takes a very general approach to communication between tasks running under the operating system. An on-chip local area network controller makes it possible to connect these chips together, so that a task running on one host processor can communicate with tasks running on the other processors in the network. The specification also includes the connection of these networks by means of bridges to other networks, so that the communication facilities stretch even further with the same ease of operation. For a task, there are no differences between communicating with a task running at the same processor and a task running at another processor.

In this graduation report, I take a look at the proposed chip from two points of view. Chapter two describes the functions that can be requested by the host processor, and thus gives the viewpoint of the programmer who has to write programs while making use of the chip. Chapter three gives a breakdown of these hardware functions into cooperating functional blocks, and thus gives the chip as it will be seen by the hardware designers (both for the surrounding logic and the chip itself).

Chapter one provides an introduction to (real time) multitasking operating systems and also tries to explain why the functions are proposed the way they are. The appendices contain a comparison between several operating systems and a comparison between the proposed functions and those provided by the iRMX 86 operating system.

Eindhoven, june 1987 Ad Verschueren.

Table of Contents

Appendices

## Preface

This graduation work started with an idea dreamed up by Prof. Stevens: 'If a multitasking operating system takes too much computer time itself, shouldn't it become time to do the multitasking in separate hardware ?'. He gave me a free hand to come up with something that would fit the description, and so I started delving into manuals of several existing operating systems.

While I was collecting bits and pieces from these operating systems to compile a set of functions to be performed by the multitasking hardware, something strange caught my eye. The multitasking was not the most important part within these operating systems, it was just a means to synchronise tasks that were communicating with eachother ! This insight shifted the emphasis from multitasking to communication and has helped a lot to expand the functionality of the hardware far beyond the first ideas.

If the multitasking hardware block that controls one processor can be made to communicate with other multitasking hardware blocks, then the tasks that run on these processors can communicate amongst eachothers as if they were running on a single processor. In this setup, communication, synchronisation and task switching are all built into a single hardware block, and work in unison.

This graduation work starts a new research project at the Eindhoven University of Technology. It is therefore largely investigative and feasibility research. For the current (1987) integrated circuit technology, building a chip implementing these functions might be just out of reach, but it should be possible within several years.

My graduation period was relatively long, over one year. This report is, as a result of this, rather bulky with over 150 pages. But it could easily be much bulkier if I had the time to put all my ideas regarding the implementation of this multitasking hardware into writing.

During the compilation of the functional description, I have constantly kept in mind that everything should be implemented in hardware. If I did not have the foggiest idea of how to implement a proposed function, then I would change the function or drop it altogether. For a complex design like this, I prefer this 'yo-yo' design approach over a strict 'top-down' design approach because it greatly reduces the risk of specifying functions that cannot be implemented.

As a result of this approach, I obtained a rough idea of the hardware and algorithms that will be needed to build the 'Multiprocessor Multitasking Coprocessor' (shorthand: 'MMTCP' - anyone out there who can think of a better name ?). This report contains most of my ideas for the hardware implementation, but almost none regarding the algorithms.

The algorithms must be implemented in hardware and 'software'. The line between hardware and software is very thin here - should a state machine controlled processor be regarded as the hardware form of the microprogram that drives a microprogram controlled processor ?

In some cases, I propose to build the algorithms in very specialised hardware like 'content addressable memories'. It is in these cases, that the description of the hardware provides a view of the underlying algorithms.

My apologies for being incomplete by not giving these algorithms. In most cases, I was satisfied if I could imagine the algorithm to be used to solve a problem, and I did not even bother to write it down. A large part of the algorithms are very straightforward (but bulky), and writing them down in this stage of the design process would have been a waste of time anyway. The more difficult algorithms can be found in the optimisation of the task switching and the communication protocols for the local area network. The former can only be worked out when the performance of the rest of the system is known, and the latter depend very much on existing standards. It is better to execute these algorithms with microprogrammed controllers, so that they need not be fixed until the last stages of the design process.

I would like to thank my coach, Prof. Stevens, for providing the opportunity to do this work. Most of the time, I was left on my own to come up with new ideas, which he then tried to talk out of my head at our next meeting. These meetings were sometimes very fiery, but they served well to keep only the good ideas upright and break down the bad ones.

I thank René Hoozemans for thinking with me and his help in understanding the UNIX operating system. I thank mr. Geurts who got me to understand multitasking operating systems at all, and pointed me in the direction of iRMX 86. I would like to thank my colleague Lucien Duijkers for his help with the LEX operating system, and Herman Vos, who is currently working on the local area network controller to be placed on the MMTCP chip. Many others contributed their bits and pieces to the project, I hope to see you all at my graduation party !

Eindhoven, july 6, 1987 Ad Verschueren.

## 1. Introduction

In this chapter I will give an overview of the basic ideas that have lead to the current specification of the MMTCP (Multiprocessor MultiTasking CoProcessor).

I will start with describing the division of programs into several more or less independent subprograms ('tasks', sometimes called 'processes') and focus on why this is done. These tasks must all be running 'at the same time' and this will lead me to the use of multitasking operating systems.

When a program is split up into several independent tasks, these tasks need to communicate with each other in several different ways. I will describe four methods of communication, each with their own specific use. I will also describe a general way to get the tasks to communicate with the outside 'world'.

The next topic will be the distribution of the tasks over different processors. There are several different ways to do this, I will describe some of them. This will lead me to the description of local area networks (LAN's) and networks of local area networks.

In the last part of this chapter I will map all the described topics onto the MMTCP system hardware and software, hoping that this will give the reader the necessary insight to understand what is going on in the other chapters of this report.

Readers with knowledge of multitasking operating systems (specifically iRMX and UNIX) can skip most of this chapter. The last two subchapters ('Multiprocessor architectures' and 'MMTCP Implementation') contain information that is necessary to understand the inner workings of the MMTCP, and should at least be glanced over.

## 1.1 Multitasking

High level languages provide the programmer with means to divide his/her program into routines that each handle part of a problem. The keywords used here are 'information hiding' and 'algorithm hiding', meaning that the exact form in which data is stored, and the way this data is handled may be 'invisible' to the programmer once these routines are written.

A routine may be directed into doing something specific by giving it parameters when the routine is 'called'. The effect after calling the routine may be that changes have occurred within (invisible) data structures and/or that result values are returned to the calling routine. A big program written in a high level language consists of a main routine that calls other routines, which in turn call other routines, etcetera. The lowest level routines do the actual work and can be seen as the leaves on a routine 'tree'.

In some programs, the structure of the problem to be solved is such that several operations can be going on at the same time. This is especially true when when it is known that a certain set of data values will be needed in the near future ('read ahead' on a disk drive) or when processed data values may be stored or output while a new set is being processed (printer 'spooling'). Sometimes it is easier to visualise the data processing as a series of transformations on the data, where each of the transformations is handled by a different program in a chain of programs running concurrently.

Computer systems that control 'real world' processes (for instance in a chemical plant) are often faced with very contradictory demands. They have to keep up with the external changes (work in 'real time', as it is called) and they should be able to control several external processes at the same time. Although the second problem can be solved by using a separate computer for each external process, this would present big problems when several of the processes are interacting with eachother. In that case, the computers should be able to communicate with eachother to exchange information about the state 'their' process is in.

Until recently, using several computers with an interconnection network has been more expensive than using one computer with the 'interconnections' in software (especially if it should work reliable). So, despite the fact that using several computers would be a more elegant solution, a single central computer is used. This central computer runs a control program that simulates the running of several programs at the same time and offers facilities for these programs to 'communicate' with each other. Such a control program is called a 'real time multitasking executive', and in most cases it can be bought from the manufacturer of the computer hardware.

Computer systems are also used to run several programs for different human users at the same time (in some situations, several programs for a single user). Depending on the situation, these programs may be using completely different sets of data, or they may be working on a common set of data (like an inventory control system with several steams of goods flowing into or out of a warehouse). Depending on the system setup, these programs may be sharing several pieces of hardware, while they may use other pieces of hardware for themselves.

Until recently, almost everything was shared (except for the keyboards and video displays), but nowadays, the users are each given an 'intelligent workstation' (a

small computer) to work with. These workstations run most of the user programs and communicate with eachother using some form of (local area) network. Coupled to this network are the more expensive pieces of equipment like fast printers, large background stores and dedicated fast computers to run programs that are too big for the workstations. These pieces of equipment are still shared amongst the users.

The control programs used for these computer systems do not differ too much from the real time multitasking executives introduced above. The demand for 'real time' handling is relaxed or completely gone, but now the emphasis has shifted to the protection of the users from eachother and the protection of the system as a whole from the users' mistakes. The same primitives used to let tasks run concurrently are now used to run complete programs 'at the same time', and the signalling between tasks is used to be able to share resources and data sets in a more or less invisible way (the programs simply request the control program the use of shared resources, and normally do not communicate with other programs themselves).

The hardware of most processors is incapable of running several programs/tasks at the same time, so the multitasking executives must switch the processor between those tasks that want to make use of it (like a normal resource). This is done by letting a program run for a while, then saving the 'volatile environment' of the program (those registers that will be changed when another program is run) and loading the volatile environment of the program that must be made running. The switching of the processor between different programs is appropriately called 'task switching', and may occur a lot within a real time multitasking system.

The multitasking executive decides which task should be made running after a task switch. Two main algorithms are used to make this decision:

**Event/Task priority** driven processor assignment. When this algorithm is used, all tasks are given a certain 'priority', indicating the importance of the task. If there are several tasks that want to use the processor, then the task with the highest priority is allowed to run. A task stops running when it has to wait for another task or for something to happen in the external 'world' (generally called an 'event'), giving way to tasks with a lower priority.

**Time slicing/'Round robin'** processor assignment. Here, a task is given a specific amount of time to run, after which a task switch is forced and the next waiting task is restarted. All waiting tasks are allowed to run one at a time, in a circular fashion (hence 'round robin'). The amount of time given to each task depends upon the amount of computer time a task needs, and how fast a task should complete it's job.

Both algorithms have their advantages and disadvantages. In most multitasking executives, both algorithms are used, sometimes in strange combinations. UNIX, for instance, implements time slicing by lowering the priority of a running task at a steady rate, while at the same time increasing the priority of tasks that are ready to run (priority based time slicing !). Some very strict *real time* multitasking executives do not implement time slicing themselves, but offer this as an option that can be implemented in higher operating system 'layers'.

Event/task priority driven processor assignment is used where tasks should be able to respond very quickly to external 'events'. The programmer can specify which event he/she considers most important by assigning high priorities to tasks that

must handle such important events. This way, handling an unimportant event can be postponed to handle more important events.

Time slicing is used when the tasks are all of more or less equal importance. It is also the only way to run tasks that may run for a long time without having to wait for external events (calculation intensive tasks, for instance).

If a task falls into an endless loop then the users of the other tasks in time sliced system only see a decrease of the system performance. If this happens to a high priority task in a priority driven system, then the system will crash ! This is the reason that event/priority driven systems are only used for those parts of a system that work reliably and cannot be (directly) influenced by users. Real time operating system applications generally fall into this category (sometimes there are no human users at all in such a system !). The control programs in multiuser environments also fall into this category, although these may provide certain event driven functions to the user programs under their own strict supervision.

A multitasking operating system keeps track of it's tasks in data structures known as 'task descriptors'. A task descriptor has fields to hold the current priority (if used), the 'state' of the task (running, ready to run, waiting for an event, etcetera) and (a reference to) the volatile environment. Note that these are only the very basic contents of a task descriptor, and that they sometimes need several hundreds of bytes ! The task descriptors of tasks that are ready to run are normally placed in a linked list called the 'ready-to-run' list. In a priority based system, this list is sorted on priority, so that the task at the head of the list has the highest priority, and is the prime candidate to be made running at the next task switch.

I have already used the word 'event' several times in this subchapter. An 'event' occurs when a task communicates with another task or when an external process changes it's state. In the next subchapter, I will describe these different forms of communication and the 'events' generated by them.

## 1.2 Communication

Communication is the main chore of multitasking systems. Tasks communicate with eachother or with the outside 'world' (with sensors, actuators or humans sitting behind a terminal), and this communication should be handled in a consistent way by the multitasking operating system.

The following four subchapters handle an equal number of different forms of communication, the fifth subchapter describes how external (hardware generated) events 'communicate' with tasks running in a multitasking system.

### 1.2.1 Synchronisation

The simplest form of communication is synchronisation. Although there is no exchange of messages of any kind, the mere fact that tasks can wait for eachother or external events is already a form of communication. For the moment, I will forget that external events are handled by the multitasking system as synchronisation events, as these will be described in the subchapter 'communication with external hardware' below.

Synchronisation always needs two parties to communicate. The task that wants to wait for the synchronisation to take place calls an operating system function specifying the operating system which synchronisation event it wants to wait for. The operating system places the requesting task 'on hold', searches for a task that is ready-to-run and makes this task running (otherwise the processor would be doing nothing).

The waiting task is made ready to run again when another task is running and decides to 'signal' the synchronisation event. To do this, the signalling task calls another operating system function, also specifying which synchronisation event is signalled. The operating system will make the waiting task ready to run again, and check it's priority. If the priority is higher than the signalling task's priority, then an immediate task switch is done, otherwise the signalled task will have to wait until there are no other tasks with a higher priority that are ready to run.

Such a synchronisation event is handled by the multitasking operating system in the form of what is generally called a 'semaphore'. The 'semaphore' is simply a name for the data structure that is used to keep track of the number of tasks waiting (in most cases a linked list of task descriptors) and the number of signals given (an up/down counter).

Numerous variations on this theme are in use. I will describe a few of them:

**'No memory'** semaphores. A semaphore is said to have no memory when signals sent to it have no effect if there are no tasks waiting. Normally, the signals sent are 'remembered' so that a task that does a wait request later will be restarted immediately.

**'Non-counting'** semaphores. A semaphore is said to be non-counting when sending more than one signal has no effect if there are no tasks waiting. Such a semaphore can only 'remember' a single signal being given in the absence of waiting tasks.

**'Queue-less'** semaphores. A semaphore is said to have no queue when all waiting tasks will be made ready to run as soon as a single signal has been given. Normally, only one (or possibly a limited number) of tasks will be made ready to run upon the reception of a signal.

**'Unit-ised'** semaphores. Semaphores of this type do not count the number of 'signal' calls in their counter, but rather the number of abstract 'units' that is specified in each 'signal' and 'wait' function call. This makes it possible to let a task wait until a specified number of 'signal' calls have been done. It also enables a single 'signal' call make a specified number of waiting tasks ready to run.

If a waiting queue is used for the waiting tasks, then this queue can be ordered in several different ways, each with their own uses (tasks at the head of the waiting queue are the first to be made ready to run). The two main queue forms are:

**FIFO** (First In First Out) waiting queues. Here the tasks await the 'signal' calls in the order in which they called 'wait' themselves. This means that the longest waiting task will be 'served' (made ready to run again) first.

**Priority based** waiting queues. Tasks are placed in the waiting queue in the order of their priority (FIFO order is used for equal priority tasks). This means that the task with the highest priority will be served first.

For very specialised functions, other exoteric waiting queue orders can be used. Some of them are available within the MMTCP, and will be described in subchapter 2.4: 'functional description: normal semaphores'.

Semaphores can be used to restrict access of tasks to specific pieces of hardware, software and/or data. This is known as 'resource locking', and a simple example is the restriction that only a single task at a time may use the system printer.

This is implemented by initialising the semaphore with a single unit, sometimes called the 'access token'. A task requesting access to the resource must first obtain this access token with a normal 'wait' call to the multitasking operating system. If the token was available at the time of call, then the task can immediately continue, otherwise it must wait until the token is returned to the semaphore by another task. Only a task that has obtained the token may use the resource, and when the resource is no longer needed, the token should be returned to the semaphore by a simple 'signal' call. The operating system will then give the token to the task at the head of the semaphore waiting queue (if any).

Problems arise when a task enters an endless loop, is involved in a deadlock situation or is removed from the system while in possession of one or more access tokens. If this is not detected by the operating system, then the token will never be returned to the semaphore, and the resource becomes unavailable (a printer may 'disappear' from the system).

To protect the system from these errors, a special kind of semaphore has been introduced which we will call a 'region'. A task that is in possession of an access token from a region semaphore cannot be removed from the system (at least not in the usual way), and is protected in some other ways too (see subchapter 2.7: 'functional description: regions').

### 1.2.2 Message exchanging

Sometimes, mere synchronisation of tasks is not enough for communication, and more information is needed by the waiting tasks (for instance: 'who restarted me ?' or 'where can I find my data ?'). In this case, short messages are exchanged between the tasks, and these are buffered in what is generally known as a 'mailbox'.

A task requesting a message from a mailbox calls an operating system function specifying which mailbox to use. If a message was available in the (FIFO organised) message buffer, then this message is given to the task immediately, otherwise the task must wait until one is written to the mailbox by another task. More than one task may be waiting for messages, these tasks are logically placed in a linked list like the one used for semaphores.

A task that wants to write a message to the mailbox does so by calling another function in the multitasking operating system. If there was a task waiting for a message, then this message is immediately given to this task, and the waiting task is made ready to run (removed from the head of the waiting list). If there were no tasks waiting, then the message is written to the FIFO message buffer that is part of the mailbox data structure.

There are two main forms of mailboxes, differing in the size of the message buffer FIFO. In the so called 'infinite' mailboxes, the message buffer is essentially unlimited in length, and a task can always write messages. In the so called 'fixed length' mailboxes, the message buffer has a limited number of 'slots' to hold messages, and a task will be placed in a waiting list if it requested to write in the mailbox while there were no slots available anymore.

Note that the actual order in which the messages enter the mailbox is not specified, nor is it possible to direct a message in a mailbox to a specific waiter (separate mailboxes will be needed to do that). This stems from the fact that in mailboxes the synchronisation of tasks is still the main issue, not the actual data that is transferred.

Most real time multitasking systems provide mailboxes directly as part of the inner layers of the operating system. In multiuser operating systems like UNIX, mailboxes are absent in these layers. If mailboxes are necessary, they can be build with a software simulated FIFO buffer and some semaphores to synchronise the reading and writing tasks.

Normally, messages may be written to a mailbox by different tasks, so that streams of messages coming from separate tasks will be mixed in the mailbox buffer. Also, several tasks may be waiting for the messages. Which message is given to a waiting task depends on the order in which the tasks requested the messages, the order in which the messages were written, and the organisation of the waiting list(s) for the tasks. The mixing and distribution of the messages can be seen as a more or less stochastic process that is very difficult to control.

We will see later that even a mailbox with a single writer and a single reader can change the order of the messages, so a mailbox should never be used if this order is important. The means of communication described in the next subchapter will

make sure that messages sent in a specific order will arrive in the same order, and will not be mixed with messages from other sources.

### 1.2.3 Character oriented data transport

Characters can be used to transport information between tasks, but are most important in the communication with human users. After all, we cannot interpret nor enter direct binary information (even a memory dump is presented to us as pages full of hexadecimal characters). Data is stored on disks and tape in the form of characters, because this makes it easier to inspect and change this information 'by hand' if something goes wrong in the system.

The transport of character data to and from a device (a keyboard, for instance) is always handled by a task within the system. There is always some kind of program that reads the keyboard data from a serial or parallel port, does some preliminary transformations (like the 'caps lock' function) and then sends the data to another task within the system for handling. So, in essence, character data transport within the system is always between two tasks, never directly between an internal task and an input/output device. How an input/output task communicates with external devices is described below in subchapter 1.2.5: 'communication with external hardware'.

With character data transport, the data itself has become more important than the synchronisation of the tasks involved. True, the sending and receiving tasks will be synchronised with eachother, but this is only necessary to keep them running at the same pace. In most systems, a FIFO oriented data buffer is inserted between the sender and receiver, making it possible to cope with temporary delays in the generation or 'consumption' of characters.

Most multitasking operating systems provide character data transport, but normally never in the lowest operating system software layer. In most systems, character data transport between tasks is implemented as a special form of input/output. UNIX calls it a 'pipe', which is also the name I will use in this report. Character transport to and from input/output devices is done by special tasks in the operating system, and the requests from internal tasks to do input/output actions are handled by higher layers of the operating system software.

Data written to a pipe should appear at the other end in exactly the same order (after all, what might happen if an escape sequence has the <ESC> character at the end ?). Also, a pipe should be used by only a single writer and a single reader at a time, otherwise the FIFO will turn into a GIGO (stands for 'Garbage In Garbage Out').

Keeping the data in the correct order is not such a problem, especially if the buffer is a simple software simulated FIFO between two tasks running at the same processor (this changes when the two tasks run at different processors !). The main difference between the operating systems lies in the way they restrict access to a pipe to a single reader and a single writer at a time.

UNIX does this by having the operating system place the pipe between two tasks at the time these tasks are created. The tasks do not even know that they communicate via the pipe, and no other task can interfere (only the operating system has knowledge about the existence of the pipe !).

iRMX86, on the other hand, treats pipes as a special form of files (and calls them 'stream files'), and gives them exactly the same possibilities as normal files. This means that it is possible to have several tasks share read and write access rights to a single pipe (although this is not recommended).

### 1.2.4 Block oriented data transport

The pipes described in the previous subchapter are used when data has to be transported between tasks at a more or less random rate. Normally, it is not known in advance how much data is going to be transported, and the pace of the transport itself depends upon the internal data handling rates of the participating tasks.

Within multiuser oriented operating systems, another form of data transport is used for what is generally known as 'file transfer'. In this case, all the data to be transported is already available at the start of the transport, so that the total amount of data is known in advance. The data transfer is initiated by tasks (in most cases within the operating system), but the actual transfer itself need not be synchronised to any task. It is only when the transfer is done that requesting tasks are notified (normally, they are allowed to continue running while the transfer is in progress).

In a single processor system or a system of closely coupled processors (processors connected to a common bus or backplane), file transfer is done with block moves between memory areas and DMA (Direct Memory Access) hardware to speed the transferring to and from input/output controllers. Hardware assistance is needed because the number of data bytes in the 'file' may be very large (up to several megabytes) and using the processors to do the transfer byte by byte would place a too heavy workload on the system.

In a loosely coupled system where the processors (or clusters of closely coupled processors) are interconnected via some form of communication link, it becomes necessary to do file transfers using these communication links. In a modern local area network, for instance, there are 'workstation' computers (without background storage) and 'file servers' (computers with large amounts of background storage). These computers need to communicate with eachother to be able to use the file server's background storage on each of the workstations.

### 1.2.5 Communication with external hardware

All computer systems need to do input and output actions to communicate with the outside 'world'. Most input/output actions contain two mayor phases - synchronisation and data transfer. In some cases, no data is actually transferred and such an action might be regarded as a special form of semaphore synchronisation.

The synchronisation phase is relatively simple. A simple binary digit suffices to tell the processor when the external process is ready to do an input/output transfer. In today's computer systems, three ways are used to handle these signals from the external processes:

1) **'Polling'**. The processor regularly reads input ports that reflect the state of the signal lines, and starts a transfer when it finds one of these lines active.

   This method is used when the signals arrive at a low speed and/or the processor has nothing else to do (very small systems with only a few tasks to perform, where hardware costs should be kept as low as possible).

2) **Interrupt**. The signal lines are connected to specialised hardware that has the capability to force the processor into the execution of a routine that performs the input/output transfer. After this routine is finished, the processor is allowed to continue it's normal operations, as if nothing had happened.

   This takes a little more hardware compared to the polling method, but is capable of handling up to several thousands of input/output transfers each second. Most microprocessors provide this hardware, either on the processor chip itself or on specialised support chips that can be connected to the processor.

3) **Direct Memory Access**. The signal lines are connected to specialised hardware that can force the processor to relinquish the system buses, and performs the input/output transfer directly between the system memory and a data port. When the transfer is done, the processor is allowed to use the buses again and only very little time is needed for each transfer (a single bus cycle suffices most of the time).

   DMA is used when the data needs to be transferred as very high speeds (up to a megabyte per second), but needs more hardware. It is very difficult to do any transformations on the data because these should then all be handled outside the processor. Sometimes special 'I/O processors' are used to relieve the host processor of the burden to initialise and monitor the input and output hardware (and also to handle the actual DMA data transfer), but this is an even more costly solution.

   DMA itself is not enough for the data transfer. Because the processor is not involved in the actual transfer, synchronisation of the transfer to running tasks should be done with one of the two methods mentioned above. In most cases, the DMA hardware or the input/output controller will use an interrupt line to signal the processor that a block of data has been transferred.

Some of the more complex input/output controllers use DMA to communicate with the processor via (chains of) command and result blocks in shared memory. These devices deserve the name 'coprocessor', because they can work more or less independently from the host processor. The synchronisation of such a device and the host is done by using 'interrupts' in both directions - a 'real' interrupt from the coprocessor to the host (indicating a result block should be handled) and a host-generated 'attention' line going to the coprocessor (indicating that a command block has been placed in memory and that this command should be executed). This method of communication is also used in multiprocessor systems where the processors are 'bus-ed' (a situation described in the next subchapter).

Within a multitasking environment, the ideal situation would be to have normal tasks communicating with the input/output devices. Most multitasking operating

systems provide means to do this by translating externally applied interrupt signals into the sending of a signal to a semaphore. If there was a task waiting at that semaphore, and this task has a higher priority than the task that was running at that time, then the interrupt routine will force a task switch to the waiting task. The interrupted task will be switched back to in the future, and experiences nothing except for a delay in execution.

Unfortunately, task switching sometimes takes much more time when compared to the original implementation with a simple interrupt routine (a task switch always saves and reloads the complete volatile environment, while some interrupt routines change only a few registers, and do not need to save these registers for the next invocation). In these cases, the old interrupt routines must be used because otherwise the processor will drown in the task switches (having no time left to process the data). This means that we are faced with two problems:

1) The interrupt routine is running asynchronously from the operating system, so it can interrupt tasks and operating system routines at any moment. If the routine were to send signals to a semaphore, then the operating system call that executes this request should be protected from interrupts because it contains critical sections (removing a task descriptor block from one linked list and inserting it in another).

2) Even if it were possible to send this signal, then it still cannot be effective, because no task switch can be executed. A task switch can only be done if the complete volatile environment is saved, and that is just what is skipped here to save time. Note that task switching within an interrupt routine is no problem if this interrupt routine has saved the complete volatile environment, or is capable of doing this when it has become known that a task switch must be done.

One way to solve this problem is by having the interrupt routine set a flag to request the sending of the signal. A special routine or task in the multitasking operating system is then set up to periodically check these flags and convert them into real semaphore signals. The periodicity is achieved by using the 'ticks' from a hardware clock interrupt to restart this routine or task at a steady rate (the clock interrupt routine is given the capability to switch tasks). The iRMX86 real time operating system manages to do without such a periodic task, but places heavy restrictions on the interrupt routine and forces it's associated task to use special operating system calls and a special kind of semaphore (which is not very flexible).

## 1.3 Multiprocessor architectures

With the advent of microprocessors, increasing the system throughput by boosting the speed of the processor has become more expensive than building a system comprised of several processors. In a multitasking environment, the tasks can now be distributed across several processors, so that we come a bit nearer to the ideal situation of running all tasks at the same time (one task per processor).

It is now also possible to differentiate the processors so that each of them excels in a specific form of computation. A processor can be equipped with floating point hardware to speed floating point calculations, array processors can be used to multiply arrays of numbers in a few microseconds and so on.

If the processors have a way to communicate with eachother, then parts of programs that may benefit from using this expensive hardware can be run on these extended processors. The programs run faster, and the expensive hardware can be shared so that it is used more cost effective.

Two forms of interprocessor communication will be described in the next subchapters.

I will not speak of the means of communication within array processors (like the Inmos 'Transputer'), because these communication channels are used for a special purpose and array processors are not likely to be used as general purpose processors in the near future. These processor systems obtain their speed from mapping the system hardware (read: communication channels) as much as possible onto the data flow between the computations in the problem at hand, which is not very flexible.

### 1.3.1 'Bus-ed' processors

I speak of bus-ed processors if their system buses are connected in such a way that they have access to a common area of working memory. This area of memory can then be used to exchange messages with eachother, thus forming a communication channel.

Synchronisation between the tasks is achieved by letting one processor generate an interrupt at another processor, which can then be handled in ways that have already been described above. This way of synchronisation is also needed to inform a task that data has been placed in (or is read from) the common memory area.

This way of communication can be made very fast, especially if the processors are allowed to write and read eachothers' memories directly. The only problem is that the distance between the processors is severely limited because it is very expensive to extend a high speed system bus over longer distances (and do this reliably).

The main difference between bus-ed processor systems and LAN interconnected systems (described below), is that in bus-ed processor systems the data transfers are done directly between the processors and data structures in memory. With LAN interconnected processors, the transfers are done with specialised input/output devices, and data must be transferred between the working memories and these devices to constitute the communication channel.

### 1.3.2 LAN interconnected processors

When a Local Area Network is used to interconnect processors (or clusters of bused processors), data is transferred between the processors using a network of interconnections controlled by specialised input/output controllers. I will use the term LAN very loosely here, because this definition encompasses also systems interconnected by simple RS-232 asynchronous connections as well as IEEE 488 and ESDI buses.

Data is transferred in 'packets' that contain control, address and data fields. The control fields are used to tell the receiver what the packet contains, and also contains information that controls and safeguards the data flow. The address field(s) are used to direct a packet of data to the correct receiver, and the data field contains the actual data.

Depending on the LAN structure, a packet may pass several processor systems before arriving at the destination, and packets sent from one source to a single destination may take different routes in the network. This last possibility may present huge problems if the packages do not contain control information to keep them in the correct order (sequence numbering), because they then can reach the receiver in a different order than the order in which they were originally transmitted.

Because the data transfer is handled in hardware (at least the most basic parts of it), synchronising it to the tasks is not such a big problem. The hardware can generate an interrupt when a packet has been received and transferred into working memory, so that the processor can inspect the packet and act upon it's contents (by restarting a task, for instance). Depending upon the amount of 'intelligence' present in the LAN controllers, the processor is more or less involved in the data transfers. In some cases, the processor must handle each byte received with an interrupt routine, at the other extreme, the LAN controller is capable of handling multiple buffers, checks the addresses (rejecting those packets not intended for the system) and also handles the lowest levels of the link management and error recovery procedures.

I will now narrow my definition of a LAN again to networks that are connected by a bus or ring, use packets by default (no byte asynchronous protocols) and have controllers that are more or less of the 'intelligent' kind described above.

### 1.3.3 Interconnected LAN's

Within a LAN, the protocols and transmission media are used by all the participating parties. The distance that can be spanned by a full size modern LAN varies but the maximum is usually several kilometers.

If two tasks need to communicate with eachother while they are running in systems that are connected to different LAN's, then a way must be found to transfer packets from the sending system's LAN to the receiving system's LAN.

If both LAN's have a computer system in common that can communicate with both of them, then this computer system can be set up to transfer packets back and forth between both LAN's. If there is no such system, then a computer system in one of the LAN's should be equipped with hardware to communicate with a similar

system in the other LAN. This hardware can take several forms, from shared memories to dedicated (satellite) data links or other LAN's.

The computer systems involved in the data transfer between LAN's are called 'bridges' or 'gateways', depending on whether or not they connect LAN's of the same type. In this report, the term 'bridge' will be used, because I do not think this distinction is very important. The MMTCP implements these connections by passing packets between the MMTCP LAN and the host memory, which would normally be called a gateway !

The basic function of a 'bridge' is to detect packets that are sent via the LAN that are intended for another LAN, check whether such a packet can be forwarded to it's destination with the means available to the bridge, and send the packet on it's way to the other LAN if so. It is possible to have several bridges connected to a single LAN, all with different other LAN's they can reach. It is also possible to have several parallel connections between two LAN's, which will make the system much more reliable (if one connection fails, the others can handle the packet traffic).

The detection of packets intended for other LAN's and the checking whether they can be forwarded should both be done in the LAN controller hardware, because otherwise all packets should be received and given to the host processor for checking in software.

The standard solution to this problem is to send all packets directly to the bridge, using a special 'bridge address'. The actual address of the receiver is then placed in the data portion of the packet, to be used by the bridge processor to choose the correct transmission path for the packet. In some cases, the complete path the message has to take must be specified within the original packet, which implies that the sender of a packet must have (up to date !) knowledge about the complete packet transmission system that is used to transfer the packet to it's destination.

Both the use of a special bridge address and the specification of the complete path in the packet make the system rather inflexible. According to the ISO-OSI standards, the complete path specification is not necessary, but the different LAN specifications are 'fuzzy' about where every address is placed within the packet structure. On one hand, the main address is subdivided into a LAN number and a number indicating which controller within the indicated LAN is addressed, on the other hand, a special address is used for the local bridge (which is supposed to handle these packets).

In my opinion, the best solution for the MMTCP would be to use the 'LAN and controller' address and have the bridges deciding which packets they can handle (based upon the LAN number). This would be easy to implement in hardware, keeps the packet length short and concentrates the necessary system knowledge within the bridges (where it belongs). My colleague Herman Vos is investigating all these possibilities, and has not come up yet with a definite solution.

## 1.3 - Introduction: Multiprocessor architectures

## 1.4 MMTCP Implementation

The MMTCP is intended to be the hardware form of a multitasking operating system, coupled with a LAN controller to exchange messages between MMTCP's. These messages will make it possible to let tasks communicate as if they were running at the same processor while they may be located in computer systems that are on the opposite side of the globe. The MMTCP will also be equipped with a local working memory, used to store the data structures that form the task descriptors, semaphores, mailboxes and so on. This working memory is also used to buffer LAN data packets and is inaccessible to the host processor (the memory is connected to the MMTCP via a special interface).

An MMTCP can be instructed to act as a 'bridge' for data packets that are not intended for the local network of MMTCP's. A list of accessible LAN numbers can be set up within the working memory of the MMTCP, and any packet intended for such an accessible LAN will then be transferred to the local host's memory (using DMA). The host will then be instructed to forward this packet to the correct remote LAN. At the receiving end, a bridge host uses DMA to transfer a received packet into a holding buffer in the working memory of it's MMTCP, after which this packet will be handled as if it were sent by a local MMTCP (and can be sent over the local LAN if necessary).

### 1.4.1 Hardware task switching

The (real time) multitasking operating system functions that are normally called as a software procedure are executed within the MMTCP hardware. The original function call is replaced by the following steps executed by the host processor:

1) Write the function number to a command register within the MMTCP.

2) Write the parameters to parameter registers, parameter values not written will be given default values by the MMTCP's command interpreters.

3) Save the volatile environment and write a pointer to this environment to a special register within the MMTCP.

4) Wait until the MMTCP indicates that a new environment pointer can be read, then read this pointer, load the indicated volatile environment and restart the associated task.

5) Read results from result registers within the MMTCP (not all results need be read), then indicate that this has been done by writing a special 'end of command' register. These results are meant for the task that was just restarted (which need not be the task that executed steps 1, 2 and 3).

The MMTCP can force a task switch by sending an interrupt signal to the host processor. The interrupt routine should execute steps 3 and 4 as indicated above. Note that in this case, it will be very likely that the task switch will restart a task that 'called an operating function', so that actually steps 3, 4 and 5 will be executed. A task switch back to the interrupted task will then consist of steps 1, 2, 3 and 4, because no results will be read. The easiest way to implement all this is to write the interrupt routine in such a way that it can be called as a normal subroutine while executing a system 'call'.

### 1.4.2 MMTCP semaphores, 'channels' and regions

The semaphores implemented by the MMTCP are 'unit-ised', and can operate in all the modes described above. They also have the capability to restart tasks that request the lowest number of units or those that request 'just the right amount' of units (can be used for optimisation purposes). The functions to wait for- or signal a semaphore can be used for semaphores that are located in other MMTCP's, as long as there is a connection between the remote MMTCP and the MMTCP of the host processor where the task is running (either directly via the on-chip LAN controllers or using bridges). As far as the running tasks is concerned, there is no difference between 'local' and 'remote' semaphores, only the addresses and response times differ.

To make it easier to port UNIX to the MMTCP, the 'channel' semaphores used in UNIX are available under certain restrictions (these 'channel' semaphores are of the queue-less no-memory type, and are addressed with 32 bit numbers defined by the tasks that use them). The most important restriction is that they can only be used by tasks that run on the same host processor.

Region semaphores are available within the MMTCP, complete with the necessary protection against task deletion. Within the MMTCP, the call that is used to end deadlock situations ('UNLINK') will also take care that all access tokens that the task possesses are automatically returned to their specific region semaphores. Note that the detection of a task being in an endless loop or a deadlock situation is not done by the MMTCP and must be implemented by the user (possibly in the form of a 'monitor' task). The MMTCP provides functions to check for deadlock situations, but has no direct means to detect an endless loop in a task program.

### 1.4.3 MMTCP mailboxes

Both types of mailboxes (infinite and fixed size) are available within the MMTCP, the type of the mailbox and the organisation of the waiting list(s) can be specified when the mailbox is created. Mailboxes can be used by local and remote tasks alike, just as with semaphores.

### 1.4.4 MMTCP pipes

Pipes are implemented within the MMTCP at the lowest level of the operating system (in this case, the chip hardware). The tasks themselves can initiate the use of pipes to transfer data, but a pipe is always 'owned' by the reading task (it cannot be given to another task for reading). At the writing end of the pipe, a kind of 'region' mechanism takes care that only a single task can write into the pipe at the same time (this task is called the 'current writer'). Again, pipes can be written by local and remote tasks alike.

Pipes do not offer hardware assistance in the transfer of data bytes between the host and the MMTCP, and so are not suited to transfer large blocks of data. This is done on purpose because setting up DMA hardware to transfer a packet of a few data bytes takes more time than doing the transfer with software running on the

host (direct memory access is impossible when pipe data is transferred into or out of the host processor's registers !).

### 1.4.5 MMTCP 'file transfer'

To be able to transfer large blocks of data efficiently using the MMTCP network, the MMTCP's should be equipped with hardware that connects to DMA controllers in the host system, making it possible to do 'end-to-end' DMA transfers of data blocks. Because the number of available DMA channels is limited (hardware !), a task in the operating system should arbitrate between tasks requesting data block transfers, and allocate the available DMA channels to them. This task should also monitor the data transfer, and signal the requesting tasks when the transfer is done.

For the MMTCP, data block transfer is something that is done 'in the background'. The packets used to transfer the data block are given a very low priority and must always wait if there is more urgent data to transport (sending units to a semaphore should not have to wait if a file of several megabytes is being transferred !).

I have chosen the term 'stream data' for the transfer of these data blocks. I do not use the term 'file transfer' because this type of data transfer is not always used for complete data files (it can be used anytime when a 'reasonably' large amount of data must be transported). I also do not use the term 'block transfer' because these blocks might be confused with the data packets used by the MMTCP's to transport the stream of data bytes.

The term 'stream data' is particularly well suited for the MMTCP implementation of 'file' transfer, because the data bytes to be transported flow like an uninterrupted stream from the sending host's memory to the receiving host's memory. The MMTCP's will buffer the data packets at each end of the link, take care of all transmission protocols and make sure that bytes are sent to the receiving host's memory in exactly the same order as they were fetched at the other end.

### 1.4.6 MMTCP communication with external hardware

The MMTCP has interrupt inputs that are scanned by on-chip hardware. If one of these inputs becomes active, this activity is internally translated to the sending of a unit to a predefined (but normal) semaphore. Optionally, the unit may be reverted to another semaphore if the first one had already accumulated a specified number of signals (interrupt 'overflow' to a so-called 'alternate interrupt semaphore', which is a normal semaphore too). The semaphore to receive the unit need not be located in the MMTCP that received the interrupt signal, so it is easy to have a signal at a specific MMTCP restart a task several kilometers away on a remote computer (as long as this computer has an MMTCP with a direct network connection to the first one). The MMTCP's will handle this conversion and sending of packets (if necessary) without any host processor intervention.

The communication between a high speed interrupt routine and tasks within the multitasking environment is made very simple by the MMTCP. A special register within the MMTCP is always ready to be written with the line number of an external interrupt input. As soon as this is done by an interrupt routine, units are

sent to semaphores as if an external interrupt input had gone active, possibly with a forced task switch request as result. By making this forced task switch request the lowest priority interrupt input on the host processor, we can make sure that the high speed interrupt routine finishes as usual (restoring changed registers just before enabling the lower level interrupts and returning to the interrupted task), following which the forced task switch interrupt routine is called. There are no restrictions placed upon the high speed interrupt routine, and the semaphore to be signalled may again be located within a remote MMTCP.

### 1.4.7 Functions not covered yet

Lots of functions provided by the MMTCP have not been described in the previous subchapters, simply because there was no space. I will now provide a short (and still incomplete) description of these missing functions.

**Suspension of tasks.** Tasks may be suspended (placed in a kind of 'extra deep sleeping state') by other tasks. During suspension, a task can change state to ready to run, but it will never be made running while in this state.

**Delays and Real time clock.** A task may be put into a sleep state for a specified number of clock 'ticks', where the number of clock ticks per second can be specified when the MMTCP is started. Alternatively, a task may be put asleep until a specific time and date, for which an on-chip real time clock is used (which can also be read to obtain the current time and date).

**Virtual memory/Process transfer support.** The MMTCP provides functions that support 'scheduler' tasks running on the host processors. These schedulers can remove 'silent' tasks from the host memory and use the space freed to load 'active' tasks. Tasks may be transferred back and forth between the host memory and background storage, which is used to form a virtual memory system. Alternatively, tasks may be transferred from 'busy' processors to 'less busy' processors connected to the local MMTCP LAN to get a more even distribution of the workload in the system.

**State timers.** For each of the states 'running', 'suspended' and 'not running nor suspended' a timer is kept for all the tasks. These timers can be used to optimise system performance in systems with virtual memory and/or process transfer.

**Deletion protection.** Nearly all system entities may be protected against deletion by the normal delete function 'calls'.

**Task permissions.** A 16 bits 'permission mask' word is present in each task descriptor, containing bits that enable or disable a task to do certain operations. If a task creates a 'child' task, then the child task can never have permissions the parent does not have itself.

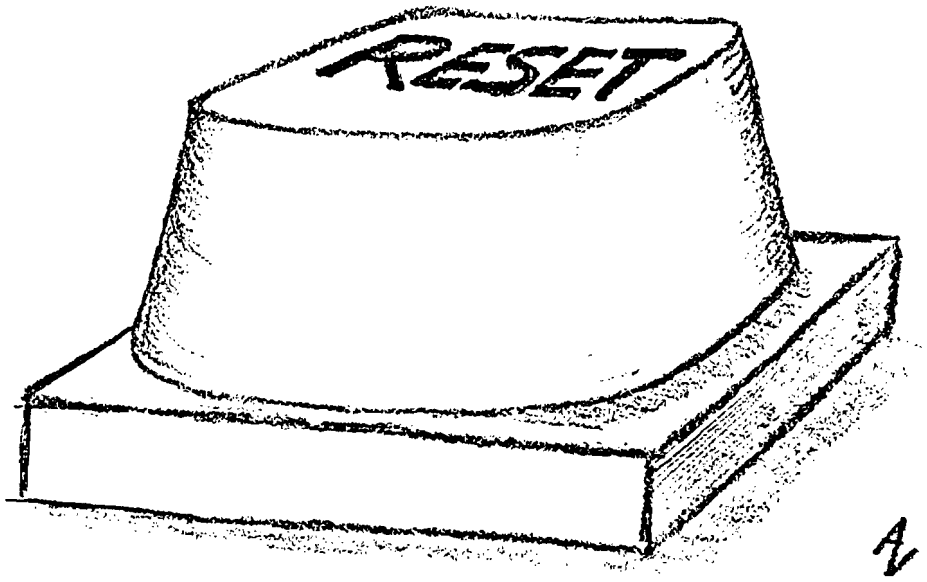
**Error handling/System startup.** Errors related to a function 'call' are reported in one of the result registers. Errors not related to any function call are reported by writing a message to a special mailbox, to be read by one of the tasks in the operating system. A similar mailbox is available to exchange initial messages following system startup (both these mailboxes have fixed

identification numbers, while all other entities are given an identification number when they are created within the MMTCP).

**User-defined status.** A 16 bits status word belonging to the running task is always readable (and can be changed), a variable sized area within each task descriptor can be read and changed by some of the tasks within the operating system. The contents of the status word and the size and contents of the task descriptor area are completely user defined. Two words in the task descriptor area can be used for the virtual memory and process transfer support functions.

**Status information.** Tasks can obtain status information for all the system entities, including the tasks themselves.

**LAN maintenance.** Functions are available to connect and disconnect a MMTCP and it's LAN, to determine the location of a LAN failure and to check out which stations are present within the local LAN.



## 2. Functional Description

This chapter contains preliminary ideas for the functions to be provided by the multitasking coprocessor chip.

In the following subchapters, the calls to control the multitasking coprocessor are given. Due to the preliminary nature of this report, no bit patterns or even numbers of bytes can be given for the data to be written into- or read from the coprocessor. I do have some ideas, though:

* The environment pointers have to be at least 48 bits long (a segment + offset specification for an 80386 takes that amount of bits). To be prepared for the future, I propose to use 64 bit environment pointers.

* All entities in the system (processes, semaphores etcetera) will be known under a logical 'name', which is a simple binary number. These 'names' will be provided by the coprocessor when the entity is created. The only exception to this rule are the so-called 'channel' semaphores (as used in UNIX), where the host processor provides a 32 bit number indicating the 'channel'.

The coprocessor has defaulting hardware build into its parameter registers. This defaulting hardware is used (amongst other things) to provide short addresses for local entities (if they are not processes, pipes, DMA channels or interrupt line numbers). Normally, an entity is addressed with a three part address:

**Ring Number**: an identification number for the ring to which the MMTCP containing the entity is connected.

**Ring Address**: an identification number to specify which MMTCP on the specified ring contains the entity.

**Local Address**: the location of the entity within the working memory of the specified MMTCP.

The defaulting hardware will provide the ring number of the local ring if the host did not write the corresponding parameter register. Likewise, the ring address is filled with 'this MMTCP' if it is not written by the host (it is considered an error to write the ring number and omit the ring address !). Note that this feature is only provided to be used by tasks that are locked into a specific MMTCP, because tasks that can be moved between coprocessors (within a ring) always have to provide at least the ring and local addresses of the entities they want to use (except for the pipes, the entities do not follow the tasks around when they are moved !).

For each of the coprocessor functions, the necessary parameters (with their default values) and the results will be given.

## 2.1. System Initialisation and Maintenance

This subchapter contains the function calls needed to initialise the coprocessor and the interconnecting Local Area Network communication links. Also, functions needed for LAN (re-)configuration and checkout are given.

The multitasking coprocessor has a built-in real time clock. Keeping this clock running during power down requires building it in CMOS technology (to keep the power consumption low during battery backup). Building this part in CMOS makes it possible to build other important registers in CMOS too, so that they only have to be initialised once, after the system is first powered up. The following call will do this:

### 2.1.1 INIT_SYSTEM

INIT_SYSTEM initialises important system parameters after power-up. This function call can only be called once after the multitasking coprocessor has been reset (unless the previous call resulted in an error). Because of the large number of parameters, some parameters may have to be packed together in a single word, while they are shown separately here.

- Parameters for INIT_SYSTEM:

  START_OF_LOCAL_MEMORY (no default) is the lowest address the coprocessor may use in the local memory.

  END_OF_LOCAL_MEMORY (no default) is the highest address the coprocessor is allowed to use in the local memory.

  TYPE_OF_LOCAL_MEMORY (no default) gives the type of memory used for the local memory, and controls such things as address multiplexing, refreshing enable/disable, refreshing intervals etcetera.

  INTERRUPT_CONFIGURATION (default: no scanning done - 12 or 20 interrupt inputs depending on host bus width) controls the number of coprocessor interrupt inputs, and the way they are scanned (all interrupt inputs are disabled after power up).

  CHANNEL_HASH_TABLE_SIZE (default: 0) sets the size of the hash table for the channel semaphores. If the size is set zero, then the channel semaphores are disabled. The user has the option to choose between 0, 32, 64 or 128 entries (preliminary !).

  USER_AREA_SIZE (default: 2 words) sets the size of the user-defined data area for each process. The lower bound for this variable is 2 words. An upper bound is not defined yet (possibly somewhere between 16 and 32 words).

  CLOCK_TICK_SPEED (no default) controls the number of clock ticks per second. Bounds are not defined yet, but a possible upper bound could be 1000 ticks/second, a lower bound depends on the hardware used. The clock ticks are derived from the crystal that controls the real time clock, so that it is possible to use a system wide clock speed divider constant while running

the multitasking coprocessor hardware at different clock speeds. Assuming the real time clock crystal runs at 32 KHz (watch crystal), a 16 bit programmable divider gives a lowest tick frequency of around 0.5 Hz, which should be low enough. Note that timeouts and delays generated for the Local Area Network controllers should be independent of the clock tick speed specified for the user processes.

DEF_MAILBOX_MODE (no default) sets the default MAILBOX_MODE to be used in the INIT_MAILBOX call (FIXED for a fixed number of mailbox 'slots' or INFINITE for an 'infinite' number of slots).

DEF_FIXED_PART_SIZE (no default) sets the default FIXED_PART_SIZE to be used in the INIT_MAILBOX call. This sets the number of mailbox slots if the MAILBOX_MODE is set FIXED, it sets the number of mailbox slots accessible at high speed when the MAILBOX_MODE is set to INFINITE.

DEF_SLOT_SIZE (no default) sets the default SLOT_SIZE to be used in the INIT_MAILBOX call (2, 4 or 8 bytes).

SYSTEM_MAILBOXES (no default) sets the sizes for the 'system errors' and 'system interconnection' mailboxes (both are described in the section on mailboxes). This parameter also sets the mode for the 'system interconnection' mailbox, and controls whether or not this last one is created. All process waiting lists for these mailboxes are organised as FIFO's.

DEF_PIPE_LENGTH (no default) sets the default pipe buffer length to be used in the INIT_PIPE call. For the lower and upper bounds see the description of the INIT_PIPE call.

COPROCESSOR_ID (no default) is the logical number of this coprocessor on the local interconnecting ring. It is used for addressing messages sent between the coprocessors. If this parameter has the value normally used as 'all stations address', then it is assumed that no interconnecting ring is available (stand alone operation), and the parameters PROCESS_TRANSFER_ENABLE, STREAM_CONFIGURATION and STREAM_INIT_TIMEOUT will be ignored. In stand alone operation, all function calls that normally would generate traffic on the local ring become illegal. Note that the coprocessor is initially disconnected from the network, so that CONNECT_NETWORK will have to be called. Also note that it is impossible to change the ring address once it is assigned, because the ring address is part of the addresses for most of the multitasking system entities located in a MMTCP.

PROCESS_TRANSFER_ENABLE (default: FALSE) controls whether or not this coprocessor will participate in the transferring of processes in a multiprocessor system. If this flag is set FALSE, then the BROADCAST and CLAIM commands become illegal, and process blocks broadcasted by other coprocessors are not stored in the private memory (saves a lot of memory). This parameter is ignored for stand alone operation.

* This flag can be set FALSE for fixed function processors in the multiprocessor system, like storage-, (remote) terminal cluster- or remote input/output controllers. Note that 'locking' processes into a coprocessor can also be done for each process individually (with the LOCAL_LOCK flag for the INIT_PROCESS function call).

STREAM_CONFIGURATION (default: no stream data handling possible) sets the number of DMA channels used to transfer stream data into or out of the coprocessor. This parameter also specifies the maximum number of data bytes in a stream data packet to be sent or received, and - if needed - the number of buffers to be used for stream data transfer. Depending on the buffering algorithm used, it may be possible to fix the number of data buffers ('twice the number of DMA channels', for instance). This parameter is ignored for stand alone operation.

STREAM_INIT_TIMEOUT (no default) sets the number of clock ticks that will be the default timeout period for a stream control process waiting for a stream initialisation packet. This timeout period cannot be infinite (would be denoted by specifying zero clock ticks). This parameter is ignored for stand alone operation.

LOCAL_RING_NUMBER (no default) is the ring number of the ring this multitasking coprocessor is connected to. All the coprocessors on a ring should have this parameter set to the same value (the token ring standard contains a protocol to do this automatically - Literature 2). The range of valid values is 1..16382 (0 is shorthand for 'this ring', 16383 is the 'all rings' address). This parameter must be given if the coprocessor is connected to a MMTCP LAN network, a host processor with main LAN interconnection capabilities, or both.

BRIDGE_ADDRESS_START (default: 0) is the lowest remote ring number that may be addressed by this coprocessor's host. The range of valid values is 0..16382 . If the value 0 is given, then it is assumed that this multitasking coprocessor is connected to a host without main LAN interconnection facilities.

BRIDGE_ADDRESS_END (default: 16382) is the highest remote ring number that may be addressed by this coprocessor's host. The range of valid values is BRIDGE_ADDRESS_START up to and including 16382. This parameter is ignored if BRIDGE_ADDRESS_START is 0.

FIXED_BRIDGE (default: FALSE) indicates whether or not the range of ring addresses given above is a fixed range. If this parameter is FALSE, then an 'accessible remote ring numbers' table is build. If this parameter is TRUE, then it is assumed that all the remote ring numbers in the given range are accessible and no such table is build (the CHANGE_BRIDGE_TABLE function call becomes illegal). This parameter is ignored if BRIDGE_ADDRESS_START is 0.

- Results returned by INIT_SYSTEM:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

Directly after initialising the system, 'root' processes should be created in each coprocessor by calling INIT_PROCESS. This will automatically make these processes the first running process in each coprocessor. The 'ghost' processes running after system reset have all their permission flags set TRUE, so that these can be set TRUE for the root processes too (the list of permission flags is given in the description of the INIT_PROCESS function call).

If BRIDGE_ADDRESS_START is 0, then it is assumed that the local host processor has no main LAN interconnection facilities, and the CHANGE_BRIDGE_TABLE function call becomes illegal (no table is stored in the working memory).

If BRIDGE_ADDRESS_START is non-zero, then it is assumed that the local host processor has one or more main LAN interconnection facilities, and can act as a bridge for connection to remote MMTCP rings. In this case, the following events occur:

* If FIXED_BRIDGE is FALSE, then an 'accessible remote ring numbers' table is build in the working memory (initialised empty = no remote rings accessible). The length of this table depends upon the difference between BRIDGE_ADDRESS_START and BRIDGE_ADDRESS_END.

* Two of the DMA channels are assigned to read from- and write into the host memory respectively. These channels will be used to transfer data packets between the multitasking coprocessor and the host processor's memory. The read channel is used to fetch packets received via the main LAN, the write channel is used to store packets intended to be sent using the main LAN. The channel numbers for the stream data transfer are offset by two (and the maximum number of stream data channels is decreased by the same amount).

* Two special purpose 'mailboxes' are initialised in the multitasking coprocessor's working memory. These mailboxes have fixed and known 'addresses' and have a slot size of 2 (two) bytes. The use of these mailboxes is explained in subchapter 2.2: 'Functional description: Bridge handling'. Note that these mailboxes only appear to be normal mailboxes, while the message queueing algorithm might differ from a plain FIFO (it will probably be based on the priority of the messages).

To keep the ring operational during fault conditions, rerouting should be possible. This may require multiple serial input and output ports for the ring controller to choose from when rerouting the data stream around the faulty ring segment(s) and/or coprocessor(s). This can be done with the SERIAL_PORTS parameter for the CONNECT_NETWORK function call.

Ring checking can be performed automatically by the master coprocessor. It can also be done with special commands to be issued by processes running on the host processor. I will propose some function calls below.

To be able to optimise the use of the Local Area Network, the different packet types should have different priorities. In general, I propose the following priority scheme to be used for the packet transmission:

| | | |
|---|---|---|
| Highest priority: | (1) | error management |
| | (2) | semaphores |
| | (3) | mailboxes, regions |
| | (4) | general management |
| | (5) | pipes |
| | (6) | process transfer |
| Lowest priority: | (7) | stream data |

Within these priority levels, there are sublevels. Normally, messages restarting a process should have a higher priority than messages that are involved in stopping a process. Also, messages containing an acknowledge for the reception of a data packet of any type should have a very high priority because in most cases the sender of the original message is waiting for such an acknowledge. In the token ring protocol, early acknowledging is possible by setting a bit in the frame closing flag, to indicate the packet has been received without errors (unfortunately it is not possible to store response data in the closing flag).

Because the connection link is to be used by many stations, there should be a priority scheme for arbitration between these stations. I would urge to use some kind of rotating priority scheme, so that there is no station capable of blocking all the other stations. For the IBM token ring, this can be done by disallowing a sending station to send more than one packet in a row (it should release a free token after it has removed it's packet from the token ring). If the sender expects an acknowledge packet, then the free token released onto the token ring should have such a priority that it can only be used for such a packet. If the receiver cannot reply immediately, then the free token will return to the sending station, which then can release a free token with the 'requested priority' found in the received token (allowing normal messages to be sent again). The token ring protocol has a build-in feature to control this sending of tokens with requested priorities (a kind of 'stack' mechanism is used) - Literature 2.

Errors detected by the ring controller are formatted into a message and written to the 'system errors mailbox', described in the section on mailboxes. This mailbox can be read by a special error control process running on the host processor. This process can then decide which actions should be taken to correct the error and can work with the user processes error detection layer in the operating system (and possibly the user processes themselves). The system errors mailbox will also be used to receive the error messages generated by the coprocessor's hardware interrupt input scanner (see the section on interrupts).

I propose to use the following function calls for LAN maintenance and checkout (VERY preliminary, and will most probably be changed !):

### 2.1.2 CONNECT_NETWORK

CONNECT_NETWORK connects the local multitasking coprocessor to the Local Area Network interconnection ring. This function is normally called only once following system startup with the INIT_SYSTEM function call. This function call is allowed only if the multitasking coprocessor is not operating in stand alone mode, and can only be called by a process which has the 'System_Task' permission bit set TRUE.

This function can also be called to change the active network connection ports (on the fly), to route the data stream around a faulty segment and/or coprocessor. Note that this part of the function call depends upon the precise hardware configuration for the low level LAN input and output controllers.

- Parameters for CONNECT_NETWORK:

MASTER_COPROCESSOR (default: FALSE) controls whether or not this coprocessor is the initial master on the local interconnecting ring (only one of them can be master, and this one is responsible for monitoring and safeguarding the message flow). If this flag is set FALSE, then the coprocessor will automatically assume the role of 'standby' ring master, as specified for the token ring network. If none of the interconnected processors has the MASTER_COPROCESSOR flag set, then one of them will become ring master automatically after a specified timeout period (no free token circulating the ring). The low level ring protocol responsible for this is described in literature 1 & 2.

ERROR_RETRY_COUNT (default as specified for the token ring standard, if this function is called for the first time, otherwise the previous setting) is used to let the Local Area Network controllers decide whether a transmission error is a 'hard' one (cable broken) or a 'soft' one (electrical interference or congestion). This parameter specifies the number of times a packet transmission may be attempted following error detection before it is considered a 'hard' error.

TRANSMISSION_TIMEOUT_PERIOD (default as specified for the token ring standard, if this function is called for the first time, otherwise the previous setting) sets a limit for the waiting time between the instant a packet is set up for transmission and the time when the transmission really starts. This has to be done because an overloaded ring can severely increase the time the coprocessors may have to wait before they can send a message, which may pose problems with processes waiting for these messages being sent or received. If the given timeout period is exceeded, a special ERROR_CODE will be given to a waiting process, or an error message will be written in the system errors mailbox.

CONGESTION_WAITING_TIME (default as specified for the token ring standard, if this function is called for the first time, otherwise the previous setting) is the waiting time inserted by the LAN output controller when the receiver of a packet has signalled that it can no longer keep up with the incoming data stream. For the token ring protocol, this signalling is done by setting the 'address recognised' bit in the closing flag, while the 'message copied' bit is not set.

SERIAL_PORTS (default: main ports for both input and output) controls the serial input and output ports that will be used for the Local Area Network connection.

- Results returned by CONNECT_NETWORK:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.1.3 DISCONNECT_NETWORK

DISCONNECT_NETWORK disconnects the local multitasking coprocessor from the Local Area Network interconnection ring. This function is normally only called if there is something seriously wrong with the interconnection network, to isolate the coprocessor from the external malfunctions. This function call is allowed only if the multitasking coprocessor is not operating in stand alone mode, and can only be called by a process which has the 'System_Task' permission bit set TRUE.

Following this function call, the local coprocessor will be logically disconnected from the Local Area Network. If the parameter SOFTWARE_DISCONNECT is set FALSE, then the coprocessor will make no attempt to signal the disconnect to the other coprocessors, which means that any remote error recovery will have to be done by error recovery software running on the remote host processors. SOFTWARE_DISCONNECT can be set FALSE for two purposes:

* An unrecoverable interconnection failure has been detected, so that there is no chance at all of getting a message through. In this case, it makes no sense to send a 'cleanup' message, as it will be lost anyway.

* The local multitasking coprocessor will be re-connected to the network within a reasonable time period, in which case it may be better to leave the remote processes waiting until this has happened. This may, however, cause problems because timeout periods may expire for these waiting processes.

This function call may be issued several times in a row, which will normally be done only to try getting 'cleanup' messages through to the other coprocessors on the network.

- Parameters for DISCONNECT_NETWORK:

SOFTWARE_DISCONNECT (default: TRUE) controls whether or not the multitasking coprocessor will try to send a 'cleanup' message to the other coprocessors on the local network. This message will automatically restart all remote processes waiting for an entity which is contained in the coprocessor for which this function is called. These processes will get an ERROR_CODE telling them that the entity they waited for no longer exists. The 'cleanup' message will also abort all process and stream data transfers into or out of this coprocessor. In case of process transfer, this means that the process remains in the 'source' coprocessor (a process transfer

is done by first copying the process data structure to the new location, after which the original data structure is deleted). A stream will be placed in the ABORT_READ or ABORT_WRITE states.

- Results returned by DISCONNECT_NETWORK:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.1.4 NETWORK_CHECKOUT

NETWORK_CHECKOUT is a general function call to invoke special protocols on the interconnection network for error detection and network management. This function may only be called by a process which has the 'System_Task' permission bit set TRUE. These functions are described in the literature for the IBM token ring protocol (Literature 1 & 2). The two main functions are:

* Sending 'beacon' messages to the other coprocessors to aid in finding ring hardware errors. The reception of beacon messages will be reported in the system error mailboxes in the other coprocessors. Beaconing may be done while the coprocessor is disconnected from the network.

* Finding out which processors are on the ring by sending address resolution packets around the ring. The responses will be stored in a mailbox, specified by the RESPONSE_MAILBOX parameter for this function call.

All subfunctions return immediately to the calling process.

- Parameters for NETWORK_CHECKOUT:

  SUBFUNCTION (default: GET_TRAFFIC_STATISTICS) specifies the subfunction to execute:

  **GET_TRAFFIC_STATISTICS** does not perform any special functions, and can be used if a process only needs to read the traffic statistics counters.

  **RESET_TRAFFIC_STATISTICS** does not perform any special function, except that it will reset the traffic statistics counters that will always be returned in response to this function call (must be done periodically to prevent them from overflowing). Note that the counters are reset after they have been copied to the result holding area or registers.

  **START_BEACONING** starts the sending of beacon packets immediately. During beaconing, the normal transmission of packets is impossible !

  **END_BEACONING** ends the transmission of beacon packets after finishing the current beacon packet. Normal transmission becomes possible again (if the network is connected).

**CHECK_ADDRESSES** initiates the sending of address resolution packets around the ring. The coprocessor has to be connected to the network if this function is to be used. This function will normally only be used by the ring master.
The protocol can be sketched as follows:

* A special packet is transmitted with the address of the source processor and an 'all receivers' destination address.

* The first coprocessor downstream will set the 'address recognised' bit and prepare to send a similar packet when a free token arrives. Other downstream coprocessors will detect the 'address recognised' bit is already set, and do not react to such a packet. Handling this part of the protocol will be done fully automatically by the coprocessors (no host intervention). Coprocessors which are disconnected from the ring will not react at all to these messages !

* The initiating coprocessor will receive all the address resolution packets, and store the source addresses it finds in them in the mailbox indicated by the RESPONSE_MAILBOX parameter. This way, the response mailbox will be filled with the addresses of all the connected coprocessors, in the order in which they are connected to the ring network.

* The initiating coprocessor knows it has obtained all the addresses when a packet is received from the first upstream processor with the 'address recognised' bit still reset. This will be indicated by storing the local coprocessor's ring address in the RESPONSE_MAILBOX.

RESPONSE_MAILBOX (no default - this parameter is necessary only for the CHECK_ADDRESSES subfunction) is the mailbox to receive the ring addresses of the coprocessors present on the interconnection ring (see the description for the CHECK_ADDRESSES subfunction described above). This mailbox should be large enough to contain all the addresses that are expected to be received, (slot width is not important, as the addresses are only 8 bits long - they will always fit). If the mailbox overflows for one reason or another, then addresses will not be written into it (the best way to prevent this is to use an 'infinite' mailbox).

- Results returned by NETWORK_CHECKOUT:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

RING_MASTER is a flag indicating whether or not this coprocessor is the current ring master. This result parameter is a 'snapshot' of the status of the LAN controllers during the interpretation of the command, and may therefore not represent the actual status at the time the requesting process starts interpreting this status (this also holds for the other result parameters).

PACKETS_SENT is a (16 bits) counter that counts the number of packets sent by this coprocessor without errors occurring. The counter will lock at 65535.

PACKETS_RECEIVED: ditto, but counts the number of packets received without errors.

SEND_CONGESTION: ditto, but counts the number of times a packet was sent that could not be handled at the receiving end (the sending of this packet will be tried again after a specified waiting time).

RECEIVE_CONGESTION: ditto, but counts the number of times a received packet could not be handled.

SEND_TIMEOUTS: ditto, but counts the number of times the sending of a packet has been aborted because of heavy bus traffic caused a timeout.

IMMEDIATE_ERRORS: ditto, but counts the number of times a packet was received containing an error, while the 'error detected' flag was not yet set (meaning that this coprocessor was the first to detect the error).

REMOTE_ERRORS: ditto, but counts the number of times a packet was received with the 'error detected' flag set (meaning the packet contains an error, and that this error has already been detected by another coprocessor).

ABORTS: ditto, but counts the number of times an 'abort' flag has been received (which is always sent before a master generates a new token on the ring). This also includes the abort flags generated by this coprocessor (if this one is the RING_MASTER). Note that aborting a stream does not generate an abort flag (any packets in transit for this stream will be finished).

TOKENS_GENERATED: ditto, but counts the number of times a free token has been forced on the ring by this coprocessor (which therefore must be the RING_MASTER).

## 2.2. Bridge Handling

Normally, an MMTCP addresses an entity located in the local system with the ring address of the MMTCP containing the entity and the location of this entity in the working memory of that MMTCP (some entities are addressed by a logical rather than by a hardware address, but this is only done to request the MMTCP's to search for the hardware address).

If the communication has to cross MMTCP ring boundaries, then the addressing range has to be expanded to include the logical ring number of the receiving MMTCP. In principle, this is not such a difficult operation - simply add some command registers to hold the ring address (with as default the local ring address). Deciding where to send the messages now goes as follows:

```
IF (ring number) = (local ring)
THEN
  IF (ring address) = (this coprocessor)
  THEN
    { Handle locally. }
  ELSE
    { Transmit information packet to other coprocessor on
      the local ring for direct handling. }
ELSE
  IF (this coprocessor connected to a bridge host) AND
     (the ring number can be reached by this bridge)
  THEN
    { Transfer the information packet to the local host
      memory and request sending it on the main LAN. }
  ELSE
    { Transmit information packet on the local ring to be
      handled by other bridge coprocessor. }
```

The last transmission poses some problems. If there are several bridge processors connected to the MMTCP LAN, each capable of transmitting messages to other bridge processors, we are facing several possibilities:

* Suppose that they are not connected to the same main LAN, and so there is the possibility that the ring numbers that they can reach do not overlap in some instances (for instance: bridge 'A' can only reach rings 1, 2 and 7, while bridge 'B' can only exchange messages with rings 3, 4 and 8).

* Suppose that some of the bridge processors are connected to the same LAN, so that they can exchange messages with the same rings. In this case, we are free to choose which of these bridges we are going to use.

In case there is a choice, it might be better to divide the local network into portions that each use a different bridge to reach the same remote ring (in an effort to divide the workload across several bridge hosts).

If there is only a single bridge processor present on the local MMTCP ring, then all the messages intended for other rings must be handled by this bridge host.

The exact routing of the messages across the main LAN network(s) has to be done by the host processor controlling a bridge.

MMTCP's should be capable of redirecting messages to the correct bridge MMTCP automatically. This is done by storing a table in the working memory of MMTCP's connected to a bridge host. This table contains the remote ring numbers that can be reached by that bridge host. This way, an MMTCP only has to send the message out on the ring, where it will be handled automatically by the bridge MMTCP capable of sending the message to the indicated remote ring.

If there are more MMTCP's capable of handling such a message, then the first upstream MMTCP will capture and handle it. There are two ways to divide the workload between the bridge processors. The first one is by physically moving bridge host MMTCP's around the MMTCP LAN ring (which might pose some problems). The second one is by disabling and enabling the handling of messages intended for specific rings periodically, so that a more or less statistical division of the workload is reached (we must make sure, though, that always at least one coprocessor can handle the outgoing traffic !).

The maintenance of the translation tables will be part of the network management software running on the host processors. This is done with the function call described below.

Only the MMTCP's connected to a bridge host need a table, and this table can be a bitmap of 2 kilobytes length (the bridge MMTCP only needs to know whether or not it can handle a message directed to a specified ring number). This bitmap will make checking so easy that it can be done while a packet is received from the MMTCP LAN, making it possible to set the 'address recognised' and 'message copied' bits in the trailing flag. Note that this checking has to be done by the intermediate level LAN input controller on the MMTCP chip, which will require extra hardware in that functional block (no, this is not a mixup of functions because this functional block was already responsible for deciding which packets were to be transferred to the high level LAN input controller and which packets should be ignored).

Informing the bridge host that a message has to be read from the coprocessor and transmitted over the main LAN can be done by sending a message containing the number of data bytes to read to a special mailbox (generating a normal task switch). The task restarted in this way can initialise the host's DMA controller for reading, and start the actual DMA transfer. If the host's DMA and LAN controllers are capable of 'chaining' data blocks, then this can be done easily using this protocol (suited for the 82586 Ethernet controller and 82285 advanced DMA controller, for instance).

Informing the MMTCP that a message has arrived and is ready to be transported into the MMTCP's working memory can be done in a similar way. A second special mailbox can be used to receive messages containing the number of bytes to be read from the packet that was received by the main LAN controller and must be read by the MMTCP. Using a mailbox will make it possible to overlap the LAN receiving and MMTCP writing tasks again by using the same chaining technique.

Both special mailboxes introduced above can only be accessed by tasks running on the bridge host. These tasks should have the 'System_Task' permission bit set TRUE. Making changes in the 'accessible ring numbers' table should also be permitted only to system tasks running on the connected bridge host.

## 2.2 - Functional Description: Bridge Handling

The following function call can be used to inspect and update the 'accessible ring numbers' table:

### 2.2.1 CHANGE_RING_ACCESS_TABLE

CHANGE_RING_ACCESS_TABLE is used to update and read the accessible ring numbers table in the working memory of a multitasking coprocessor connected to a bridge host. This function call can only be issued to the local coprocessor by a task executing with the 'System_Task' permission bit set TRUE.

Note that the local ring number (set by the LOCAL_RING_NUMBER parameter for the INIT_SYSTEM function call) is a special case. If this number is in the range of ring numbers contained in the table, then the 'accessible' setting of this ring number in the table is completely ignored (the table in not even consulted if a packet is received on the MMTCP LAN with this ring number as destination).

- Parameters for CHANGE_RING_ACCESS_TABLE:

SUBFUNCTION (default: CHECK_IF_PRESENT) can be one of the following:

CHECK_IF_PRESENT checks if at least one of the ring numbers in the range LOW_NUMBER..HIGH_NUMBER can be accessed, and returns the result of this check in the CHECK_RESULT status flag.

CHECK_IF_ABSENT checks if any of the ring numbers in the range LOW_NUMBER..HIGH_NUMBER is inaccessible, and returns the result of this check in the CHECK_RESULT status flag.

MAKE_ACCESSIBLE makes the ring numbers in the range LOW_NUMBER..HIGH_NUMBER accessible. This will make this bridge coprocessor respond to messages with a remote ring number in the given range.

MAKE_INACCESSIBLE makes the ring numbers in the range LOW_NUMBER..HIGH_NUMBER inaccessible. This bridge coprocessor will stop responding to messages sent over the MMTCP LAN network with a remote ring number in the given range.

LOW_NUMBER (default: BRIDGE_ADDRESS_START, as given for INIT_SYSTEM) indicates the start of the range of remote ring numbers for which the SUBFUNCTION is to be used. This parameter should be in the range BRIDGE_ADDRESS_START .. BRIDGE_ADDRESS_END as given for INIT_SYSTEM.

HIGH_NUMBER (default: BRIDGE_ADDRESS_END, as given for INIT_SYSTEM) indicates the end of the range of remote ring numbers for which the SUBFUNCTION is to be used. This parameter should be in the range LOW_NUMBER .. BRIDGE_ADDRESS_END as given for INIT_SYSTEM, and never be numerically below LOW_NUMBER.

- Results returned by CHANGE_RING_ACCESS_TABLE:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  CHECK_RESULT is a status flag (TRUE or FALSE), which is returned if the SUBFUNCTION was either CHECK_IF_PRESENT or CHECK_IF_ABSENT.

**2.2 - Functional Description: Bridge Handling**

## 2.3. Processes

Processes are the operational entities in the multitasking system. They execute the programs and interact with eachother and the outside world. Processes can be moved between MMTCP's connected to the same token ring network (not between networks).

The identification numbers for processes differ from the other entities. The ring address has become useless, because the identification number should remain the same while the process can move between different MMTCP's on the same ring. Therefore, the identification number consists of a ring number (default: 'this ring'), a dummy ring address (actually the ring address of the MMTCP where the process was created) and a 16 bits identification number (that has nothing to do with addresses within the MMTCP working memory).

A process that is locked within an MMTCP (either because of stand alone operation or the LOCAL_LOCK flag being set TRUE) can be addressed by other processes in the same MMTCP with only the 16 bits identification number, but if the addressed process is not locked, it is regarded an error to do so.

### 2.3.1 INIT_PROCESS

INIT_PROCESS initialises a process, and, if SUSPENSION_DEPTH is 0, places this process in the ready to run queue. If SUSPENSION_DEPTH is non-zero, then the new process will be 'ready-suspended', and can be placed in a waiting state by the creating process before releasing the suspension. This makes it possible to create a process that will wait for an event, handle the event when it occurs, and immediately afterwards terminates itself (one way to implement 'callouts' in UNIX).

- Parameters for INIT_PROCESS:

  ENVIRONMENT (no default) is the pointer to the initial environment for the new process.

  PRIORITY (default: MAX_PRIO given below, 0 for the first process created) is the initial priority for the new process. PRIORITY cannot be set outside the MAX_PRIO..MIN_PRIO range given by the next two parameters. 0 (zero) is the highest priority.

  MAX_PRIO (default: MAX_PRIO of creating process, 0 for the first process created) is the maximum priority the new process may ever be set to. MAX_PRIO cannot be set above (numerically below) the MAX_PRIO of the creating process.

  MIN_PRIO (default: MIN_PRIO of creating process, 65535 for the first process created) is the minimum priority the new process may ever be set to. MIN_PRIO cannot be set below (numerically above) the MIN_PRIO of the creating process, and should always be equal to or lower than (numerically above) the MAX_PRIO parameter given above.

  LOCAL_LOCK (default: FALSE) disables the BROADCAST function call for the new process. To be able to set this flag TRUE, the creating process

should have the 'Local_Lock_Enable' permission bit set TRUE. Locked tasks will never be found by the scheduler list search function calls if the FIND_LOCKED_TASKS bit is not set TRUE.

PERMISSIONS_MASK (default: all permissions of the calling process) is the mask used to reset permission bits for the new process. These permission bits are packed in a 16 bits word, and are all set TRUE for the process created when the system is started up (they are also set TRUE for the 'process' that created this first process). If a permission bit in this mask is FALSE, then the corresponding permission bit for the new process is set FALSE too (logical AND action on the PERMISSIONS_MASK and the PERMISSIONS_STATUS for the calling task). There is no way whatsoever to set a permission bit TRUE for a task once it is set FALSE, and a task with a permission bit set FALSE cannot create a task with this bit TRUE (if a task tries to do this, it is regarded an error and no child task will be created). The following permission bits are incorporated in the mask (preliminary !):

**System_Task** should be TRUE for a task that wants to use some of the special function calls available like UNLINK or FORCE_DELETE, or requests access to the 'system errors' and 'system interconnection' mailboxes.

**Scheduler** should be TRUE for a task that wants to search the schedulers tasks list, manipulate the 'Dirty' or 'Alt_Ready_Queue' flags, or wants to use the BROADCAST and CLAIM function calls.

**Stream_Control** should be TRUE for a task that wants to use the function calls related to stream data handling.

**Chan_Sem_Use** should be TRUE for a task that wants to make use of channel semaphores (if they are enabled at all).

**Offspring_Generation** should be TRUE for a task that wants to create a 'child' task with the INIT_PROCESS function call.

**Create_Enable** should be TRUE for a task that wants to create anything (including a 'child' task).

**Non_Owned_Delete** should be TRUE for a task that wants to delete a multitasking operating system 'entity' it did not create itself (like a semaphore, mailbox, 'child' task, etcetera).

**Stop_Other_Tasks** should be TRUE for a task that wants to call a function that stops a task (like WAIT), while specifying another task to be stopped, where this other task is not it's own offspring.

**Non_C_W_Flush** should be TRUE for a task that wants to flush a pipe it is not the owner, nor the 'current writer' for.

**Global_Operations** should be TRUE for a task that wants to use a function call accessing a non-local operating system 'entity' (except for pipes).

**Local_Lock_Enable** should be TRUE for a task that wants to create a 'child' task with the LOCAL_LOCK flag set TRUE.

SPECIAL_STATUS (default: 0) is the initial value for the 'special status' word, which is always readable in a special coprocessor register while a process is running. This field can be changed by SET_SPECIAL_STATUS, and is completely user-defined.

SUSPENSION_DEPTH (default: 0 = not suspended) is the initial suspension depth for the process. If this parameter is set non-zero, then the new process is not placed in the ready to run queue immediately after the INIT_PROCESS call ends, and the new process can be placed in some other state (and/or waiting list).

TIME_SLICE (default: 0 = no maximum running time) is the maximum number of clock 'ticks' a process can run without being placed in the ready queue behind all other process with the same priority. This is a 'hard' time slicing algorithm. If the time slice timer times out and there are no other processes with the same priority, then an internal flag is set indicating that a process with the same priority may preempt the running process (otherwise it would be placed in the ready to run queue).

- Results returned by INIT_PROCESS:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

PROCESS is the identification number to be used in the future when referencing to the new process.

### 2.3.2 TERM_PROCESS

TERM_PROCESS terminates a process if the process is not running within a region and the 'deletion holdoff' counter (controlled by DISABLE_DELETION and ENABLE_DELETION) is zero. If the process is 'current writer' for one or more pipes, then a RELEASE_PIPE call is done for all these pipes. If the process is the reading process for one or more pipes, then these pipes will be deleted too, regardless of the state of the deletion holdoff counters for these pipes (it makes no sense to have a pipe that 'ends in mid-air').

- Parameters for TERM_PROCESS:

PROCESS (default: running process) is the process to terminate.

- Results returned by TERM_PROCESS:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is. This error code is only returned if the terminated process was not the calling process !

### 2.3.3 DELAY

DELAY lets a process wait for a specified number of clock ticks (the process is removed from the ready to run list during that time).

- Parameters for DELAY:

  PROCESS (default: running process) is the process to be delayed. If not the running process is specified, then this process should be in the 'ready to run' state (in any ready queue). The process may currently be suspended !

  TIMEOUT (default: 1 clock tick) is the number of clock ticks the process is delayed. The actual delay may vary between (n-1) and n times the clock tick period, if n is the number given here. If the process does not have enough priority to preempt the process running when the delay times out, it will have to wait longer to become the running process again. 0 clock ticks is not allowed.

- Results returned by DELAY:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.3.4 SUSPEND_PROCESS

SUSPEND_PROCESS increases the 'suspension depth' counter for the given process and makes it impossible for the process to be placed on any ready queue (removes the process from a ready queue if it is already there - note that the actual process state does not change). Suspending a process is impossible if the process is currently running in a region. There is a maximum suspension depth (probably 255), which limits the number of SUSPEND_PROCESS calls in a row.

There are 2 ready queues, the normal ready queue and the alternate ready 'queue'. These are described in the section on virtual memory support.

- Parameters for SUSPEND_PROCESS:

  PROCESS (default: running process) is the process to suspend.

- Results returned by SUSPEND_PROCESS:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is. A self-suspend normally returns the 'no error' code.

  SUSPENSION_DEPTH is the new state for the 'suspension depth' counter of the suspended process. This result will always be 0 if the calling task suspended itself !

### 2.3.5 RESUME_PROCESS

RESUME_PROCESS decreases the 'suspension depth' counter for the given process, if this counter was not already 0. If the counter reached zero by this call, then the process is no longer suspended, and can be placed in any ready queue (and will be placed there if the process is not waiting for an event to happen).

- Parameters for RESUME_PROCESS:

PROCESS (no default) is the process to resume. The running process cannot be specified !

- Results returned by RESUME_PROCESS:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

SUSPENSION_DEPTH is the new state for the 'suspension depth' counter of the 'resumed' process.

### 2.3.6 CHANGE_PRIORITY

CHANGE_PRIORITY sets or reads the priority for the specified process.

- Parameters for CHANGE_PRIORITY:

PROCESS (default: running process) is the process from which the priority is asked.

PRIORITY (default: current priority for the specified process) is the new priority to be given to the specified process. If the default is kept, then the priority is not changed.

- Results returned by CHANGE_PRIORITY:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

PRIORITY is the (new) priority of the specified process. This will never return the temporary priority given to the process by the POLL function.

### 2.3.7 CHANGE_USER_AREA

CHANGE_USER_AREA changes the user-defined data space for the specified process. The size of this data space is set by the USER_AREA_SIZE parameter for the INIT_SYSTEM function, and it contains at least user defined 'PROCESS_SIZE' and 'USER_PROCESS_STATE' entries. The user-defined data space is initialised to 'all zeroes' by the INIT_PROCESS function.

- Parameters for CHANGE_USER_AREA:

  PROCESS (default: running process) is the process from which the data area is to be changed.

  USER_AREA_DATA (default: current data in the user area) gives the data to be written into the user area. Data is written on a word-default base, that is, if a word is not written, it keeps the original contents.

- Results returned by CHANGE_USER_AREA:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  USER_AREA_DATA are the contents of the user area for the indicated process (after the changes are made).

### 2.3.8 YIELD

YIELD places the first process in the ready queue with a priority equal or below the specified priority behind all other process in the ready queue with the same priority. For this function, the running process is considered to be on the top of the ready queue. This function can be used for user-defined time slicing.

- Parameters for YIELD:

  PRIORITY (default: priority of the running process) is the maximum priority for the process to swap places in the ready queue.

- Results returned by YIELD:

  ERROR_CODE is set to 'no error', because this function cannot generate an error condition.

### 2.3.9 POLL

POLL assigns a new priority to the indicated process which is lower than the current priority of the process. This priority cannot be lower (numerically higher) than the MIN_PRIO parameter given in the INIT_PROCESS call which created this process. The normal priority of the process is re-instated when the process is made running again, or when the specified timeout time has elapsed. This function can be used by processes which have to wait for an external event which does not generate an interrupt, if processing this external event has a relatively low priority.

- Parameters for POLL:

  PROCESS (default: running process) is the process to receive a temporary lowered priority.

POLL_PRIORITY (default: MIN_PRIO given by INIT_PROCESS) is the temporary priority for the specified process.

TIMEOUT (default: 0 = infinity) is the maximum number of clock ticks this lowered priority will exist.

- Results returned by POLL:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

TIME_ELAPSED gives the number of clock ticks this lowered priority existed (65535 = 65535 or more).

### 2.3.10 GET_STATISTICS

GET_STATISTICS gives an overall indication of the state of a process. It can be used together with CHANGE_USER_AREA and CHANGE_TIMERS by the scheduler process(es) to make decisions whether or not to swap processes out or in, and can also be used for system optimalisation and resource profiling.

- Parameters for GET_STATISTICS:

PROCESS (default: running process) is the process for which the statistics are requested.

- Results returned by GET_STATISTICS:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

PROCESS_STATE is a collection of flags, indicating the overall process state (preliminary !):

Wait_For indicates what the process is waiting for (if anything). These are the possibilities:

* **Nothing** (process is not waiting)
* **Normal Semaphore**
* **Channel Semaphore**
* **Region**
* **Mailbox Reading**
* **Mailbox Writing** (non-infinite mailbox only)
* **Pipe Reading**
* **Pipe Writing**
* **Pipe Claiming**
* **Real Time Clock Match**
* **Stream Initialisation**
* **Stream Ending**
* **Poll State Ending**
* **Scheduler Process Search**
* **LAN Reply** (acknowledgement or status packet)

**Delay_Running** indicates whether or not a delay or timeout is running. If a process called DELAY, then this flag is TRUE, with 'Wait_For' set to 'Nothing'.

**Remote_Wait** indicates the process is waiting for a remote entity.

**BroadCast_TX** indicates the process has been broadcasted to the other MMTCP's on the local token ring to request it to be transferred away from this coprocessor.

**BroadCast_RX** indicates the process has been received by the local coprocessor following a BROADCAST function call in one of the other coprocessors on the local token ring.

**Local_Lock** indicates this process is not to be moved between coprocessors. The setting of this flag is specified when a process is created with INIT_PROCESS.

**Alt_Ready_Queue** indicates this process cannot be placed in the normal ready to run queue (in most cases this flag is set because the process has been swapped out to background storage in a virtual memory system).

**Dirty** indicates this process is being used by (one of) the scheduler processes. This also means this process has limited capabilities to change state (cannot be made running, for instance).

PERMISSIONS_STATUS gives the collection of permission bits that control which functions this task may call. These bits are packed in a 16 bits word, and are controlled by the PERMISSION_MASK in the INIT_PROCESS function call.

QUEUE gives the identification number of the entity the process is waiting for (if any). This can be a (channel) semaphore, mailbox, region etcetera.

ENVIRONMENT is the pointer to the environment of the process. If the environments are saved dynamically (on the stack, for instance), then this value is meaningless for the running process.

CURRENT_COPROCESSOR gives the number of the coprocessor where the process is currently running (can be used in a system with a central scheduling algorithm, or to check for errors).

PRIORITY is the current priority for the process. If the process has called POLL, then PRIORITY indicates the normal priority, not the priority given with POLL.

SUSPENSION_DEPTH is the current contents of the suspension depth counter for the process. If non-zero, then the process is suspended, even if the flags in the PROCESS_STATE indicate the process is not waiting for an event.

REGIONS_ENTERED gives the number of regions the process is currently running in.

CURRENT_WRITER_STATUS gives the number of pipes for which the process is the 'current writer'.

READER_STATUS gives the number of pipes from which the process is reading (these are logically 'owned' by the process).

DELETION_HOLDOFF gives the current contents for the 'deletion holdoff' counter for this process. If non-zero, the process can only be deleted by FORCE_DELETE.

SPECIAL_STATUS is the special status word for the given process.

TIME_REMAINING gives the number of clock ticks the process has to wait until the timeout condition occurs (if a timeout is running).

### 2.3.11 CHANGE_TIMERS

CHANGE_TIMERS sets and/or reads the three process timers (one each for the running, waiting and suspended states). These three timers are 32 bits wide, increment once for each clock tick, are initialised to 0 by INIT_PROCESS and lock before overflow.

The times returned in the result registers are the times AFTER they have been changed by the parameters in this call.

Normally, this call will return immediately to the calling process with the results. If a process with higher priority preempts the calling process while this call is processed, then the results will reflect the state of these timers as they were at the entry of this call (the actual preemption will be done after the result has been written to the process block in memory).

- Parameters for CHANGE_TIMERS:

PROCESS (default: running process) is the process for which the timer values are to be set and/or read.

CPU_TIME (default: current value) is the new value for the (cumulative) running time timer.

SUSPENDED_TIME (default: current value) is the new value for the (cumulative) suspended state time timer.

WAITING_TIME (default: current value) is the new value for the (cumulative) waiting state time timer.

- Results returned by CHANGE_TIMERS:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

CPU_TIME gives the (cumulative) number of clock ticks the process has run on the processor.

SUSPENDED_TIME gives the (cumulative) number of clock ticks the process has been in suspension.

WAITING_TIME gives the (cumulative) number of clock ticks the process has been waiting in any kind of queue (including the ready queue !). This timer is also running if the process is waiting for a reply from the LAN.

### 2.3.12 SET_SPECIAL_STATUS

SET_SPECIAL_STATUS sets a new value for the special status word for the specified process. The special status word is always readable in a special coprocessor register while a process is running.

- Parameters for SET_SPECIAL_STATUS:

PROCESS (default: running process) is the process which is to receive the new special status word.

SPECIAL_STATUS (default: 0) is the new special status word for the specified process.

- Results returned by SET_SPECIAL_STATUS:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.3.13 SET_ENVIRONMENT

SET_ENVIRONMENT changes the environment pointer stored in the process descriptor block of the specified process. This call should be used if the environment of the given process has been moved in memory (because the process has been swapped out or because the process has been moved between processors).

- Parameters for SET_ENVIRONMENT:

PROCESS (no default) is the process to receive the new environment pointer. This cannot be the running process.

ENVIRONMENT_POINTER is the new environment pointer for the given process.

- Results returned by SET_ENVIRONMENT:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.3.14 UNLINK

UNLINK forces a process to the ready state (out of any event waiting queue, with suspension depth 0), and is primarily used to end deadlock situations. The task calling this function should have the 'System_Task' permission bit set TRUE.

If the target process has entered regions, then an EXIT_REGION call is executed for all these regions. If the target process is 'current writer' for one or more pipes, then a RELEASE_PIPE call is done for these pipes. A special ERROR_CODE is returned to the UNLINK-ed process. If a target process is already in the ready state, not suspended, not in a region and not a 'current writer' for a pipe at the time of the UNLINK call, then this process experiences nothing, but a special ERROR_CODE is returned to the caller of UNLINK.

- Parameters for UNLINK:

PROCESS (no default) is the process to make ready. This should not be the running process.

- Results returned by UNLINK:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is. This word contains bits indicating whether or not the process was released from a waiting queue and/or un-suspended.

PIPES_RELEASED gives the number of pipes released by the target process.

REGIONS_EXITED gives the number of regions exited by the target process.

## 2.4 Normal Semaphores

The semaphores are used to synchronise processes running in the multitasking environment. The synchronisation is achieved by having processes request a number of abstract 'units' from a semaphore, which will stop them from continuing their work if these units are not available at that time. Other processes can send units to semaphores, which will restart processes waiting there. To make the multitasking coprocessor's semaphores as versatile as possible, they can be set to operate in several different 'modes'. Also, the number of units requested and sent can be specified for each call (some multitasking operating systems only allow sending or receiving one unit at a time). By using semaphore modes like 'BEST_FIT', the semaphores themselves can be used for optimisation purposes.

It is possible to send units to a semaphore resident in another multitasking coprocessor, if the local and remote MMTCP's are connected by a LAN and/or bridges. It is also possible to wait for units at a non-local semaphore. The multitasking coprocessors will automatically and transparently exchange messages to initiate and end the waiting of the process. This makes it possible to build a multiprocessor multitasking system where tasks operating on different host processors synchronise to eachother using semaphores, just like they would if they were running on a single host processor.

### 2.4.1 INIT_SEMAPHORE

INIT_SEMAPHORE creates and initialises a semaphore. This function can only be called by a task which has the 'Create_Enable' permission bit set. If the MODE parameter is NO_QUEUE_FIFO or NO_QUEUE_PRIO, then the semaphore has no queue and all processes waiting are released as soon as there are enough units in the semaphore's units counter to satisfy the request for the first waiter. With proper use of the condition settings for SIGNAL, the semaphore can be made non-accumulating or even without memory. The units counter is 16 bits wide.

- Parameters for INIT_SEMAPHORE:

  INITIAL_UNITS (default: 0) gives an initial number of units to the semaphore's units counter.

  MAX_UNITS (default: 65535) sets the maximum number of units that can ever be accumulated in the semaphore's units counter.

  MAX_WAITERS (default: 65535) sets the maximum number of waiting processes allowed in the semaphore's waiting queue.

  GLOBAL (default: TRUE) indicates whether or not the semaphore has to be made globally accessible throughout the (multiprocessor) system.

  MODE (default: FIFO) sets the queueing mode for the semaphore's waiting list. This mode can be:

  **FIFO**: processes wait in first-come first-served order.

**PRIO**: processes wait in priority order, new processes are placed behind other processes with equal or higher priority.

**NO_QUEUE_FIFO**: processes are placed in the queue in first-come first-served order, and are all released at the same time as soon as the request for the process at the head of the queue can be satisfied.

**NO_QUEUE_PRIO**: processes are placed in the queue in priority order (as for PRIO), and are all released at the same time as soon as the request for the process at the head of the queue can be satisfied.

**FIRST_FIT**: processes requesting few units are placed in front of other processes in the queue (with processes which have the same number of requested units in order of decreasing priority). If units are sent, then they are given to the process placed at the front of the queue. This means that processes which request few units are favoured over processes that request more units.

**BEST_FIT**: the processes are placed in the queue ordered on decreasing amount of requested units (with processes which have the same number of requested units placed in order of decreasing priority). If units are sent, then the queue is searched for the first process that requests a number of units less than or equal to the number of units accumulated in the units counter. This process is restarted, and, if there are units left, searching is continued to find another process to consume (part of) the remaining units. This means that processes which request 'just enough' units are favoured over all other processes. If, for instance, there are processes with requests for 7, 4, 3 and 2 units waiting (no units in the semaphore's counter), then sending 5 units will release the process requesting 4 units, leaving 1 unit in the semaphore's counter. Sending another 10 units will release the processes requesting 7 and 3 units, leaving the counter with 1 unit.

Note that a semaphore with a BEST_FIT queueing mode may be somewhat slower in response for a SIGNAL function call (described below), because the list of waiting processes has to be searched in this case (all the other queueing modes always release the processes from the head of the waiting list).

- Results returned by INIT_SEMAPHORE:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

SEMAPHORE is the identification number to be used in the future when referencing to the new semaphore.

### 2.4.2 TERM_SEMAPHORE

TERM_SEMAPHORE deletes a semaphore if the 'deletion holdoff' counter is 0. This function can only be called by a task which has created the semaphore itself, or by a task which has the 'Non_Owned_Delete' permission bit set. All waiters are released and receive a special ERROR_CODE to indicate the semaphore no longer exists.

- Parameters for TERM_SEMAPHORE:

SEMAPHORE (no default) indicates which semaphore is to be deleted.

MAX_UNITS (default: 65535) gives the maximum number of units allowed in the semaphore's units counter to enable the deletion.

MAX_WAITERS (default: 65535) gives the maximum number of waiters allowed to be in the semaphore's waiting queue to enable the deletion.

- Results returned by TERM_SEMAPHORE:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.4.3 WAIT

WAIT is called by a process to 'receive' units from a semaphore's units counter. The general idea is to place the process in the semaphore's waiting queue, and release the process as soon as it is placed at the head of the waiting queue and there are enough units in the units counter to satisfy the request (or if the first process in the queue is released and the MODE is NO_QUEUE_FIFO or NO_QUEUE_PRIO). The BEST_FIT algorithm described above works differently.

- Parameters for WAIT:

SEMAPHORE (no default) is the semaphore to use.

PROCESS (default: running process) is the process to wait at the semaphore. This process should be in the ready to run state (the running process always is), and not in a region if the default is not used. If the running process is not specified here, then the specified process will not receive an ERROR_CODE resulting from this call, but will receive the ERROR_CODE it was already set to receive (if any). The Stop_Other_Tasks permission bit should be set if the specified process is not a direct child of the calling task, nor the calling task itself.

NR_OF_UNITS (default: 1) is the number of units requested from the semaphore. A WAIT call which requests 0 (zero) units is allowed, and the request can then always be satisfied (process released as soon as it reaches the head of the ready queue, the process is always released if the semaphore's queueing mode is FIRST_FIT).

MIN_UNITS (default: 0) gives the minimum number of units to be in the semaphore's units counter to enable the WAIT function.

MAX_UNITS (default: 65535) gives the maximum number of units in the semaphore's units counter to enable the WAIT function.

MIN_WAITERS (default: 0) gives the minimum number of waiting processes in the semaphore's waiting queue to enable the WAIT function.

MAX_WAITERS (default: 65535) gives the maximum number of waiting processes in the semaphore's waiting queue to enable the WAIT function.

TIMEOUT (default: 0 = infinite) gives the maximum number of clock ticks the process will wait in the semaphore's waiting queue.

- Results returned by WAIT:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

CURRENT_WAITERS gives the number of waiting processes in the semaphore's waiting queue, at the time this process was released from the waiting queue (or the number of waiters at the time of exit from the WAIT call, if the function was not enabled).

CURRENT_UNITS gives the contents of the semaphore's units counter, at the time this process was released from the waiting queue (or the number of units at the time of exit from the WAIT call, if the function was not enabled).

TIME_ELAPSED gives the number of clock ticks the process was placed in the waiting queue. This is a 16 bit value, which blocks before overflow. TIME_ELAPSED is only returned if the running process called WAIT.

### 2.4.4 SIGNAL

SIGNAL sends a specified number of units to a semaphore's units counter. This function returns an error if adding the number of units to the units counter would increase the number of units in the counter above it's maximum set by INIT_SEMAPHORE. Note that the units are first added to the counter, and then dispatched to the waiting processes !

- Parameters for SIGNAL:

SEMAPHORE (no default) is the semaphore to receive the units.

NR_OF_UNITS (default: 1) is the number of units to send to the semaphore.

* If NR_OF_UNITS is set 0 (zero), then this function call does a 'force release' for the first waiter in the semaphore's waiting list (units counter unchanged). In this case, the first waiter will receive a special ERROR_CODE (if this process placed itself in the waiting

queue). If the MODE is set NO_QUEUE_FIFO or NO_QUEUE_PRIO then all waiters are released.

MIN_UNITS (default: 0) gives the minimum number of units to be in the semaphore's units counter to enable the SIGNAL function.

MAX_UNITS (default: 65535) gives the maximum number of units in the semaphore's units counter to enable the SIGNAL function.

MIN_WAITERS (default: 0) gives the minimum number of waiting processes in the semaphore's waiting queue to enable the SIGNAL function.

MAX_WAITERS (default: 65535) gives the maximum number of waiting processes in the semaphore's waiting queue to enable the SIGNAL function.

- Results returned by SIGNAL:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

CURRENT_UNITS gives the number of units in the semaphore's units counter after the NR_OF_UNITS were added, but before there were waiters released.

CURRENT_WAITERS gives the current number of waiters in the semaphore's waiting queue ('current' = at the time the SIGNAL call was issued).

### 2.4.5 CHECK_SEMAPHORE

CHECK_SEMAPHORE returns the number of units and waiters for a specified semaphore, and can also be used as a check for the existence of the semaphore.

- Parameters for CHECK_SEMAPHORE:

SEMAPHORE (no default) is the semaphore to check.

- Results returned by CHECK_SEMAPHORE:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

CURRENT_UNITS is the state of the semaphore's units counter at the time of the CHECK_SEMAPHORE call.

CURRENT_WAITERS is the number of waiters in the semaphore's waiting queue at the time of the CHECK_SEMAPHORE call.

## 2.5 Channel Semaphores

Channel semaphores have to be enabled by the INIT_SYSTEM function, before the following functions can be used. Also, the calling process should have the 'Chan_Sem_Use' permission bit set.

Channel semaphores are all non-counting without memory.

The CHANNEL is a user-defined 32 bit number.

Channel semaphores are always local, and their sole purpose is to make it more easier to emulate UNIX on the coprocessor (they are much less versatile than the normal semaphores).

Processes waiting for a channel semaphore are put into the ready to run state immediately when they are transferred to another coprocessor (they lose their 'connection' to the channel semaphore), and receive a special ERROR_CODE to indicate this has happened.

### 2.5.1 WAIT_CHANNEL

WAIT_CHANNEL places a process in a channel's waiting queue, until a SIGNAL_CHANNEL function is executed for the specified channel.

- Parameters for WAIT_CHANNEL:

  CHANNEL (no default) is the channel to wait on.

  PROCESS (default: running process) is the process to wait at the channel semaphore. This process should be in the ready to run state (the running process always is), and not in a region if it is not the running process. If the running process is not specified here, then the specified process will not receive an ERROR_CODE resulting from this call, but will receive the ERROR_CODE it was already set to receive (if any). The Stop_Other_Tasks permission bit should be set if the specified process is not a direct child of the calling task, nor the calling task itself.

  TIMEOUT (default: 0 = infinite) gives the maximum number of clock ticks the process will wait in the channel semaphore's waiting queue.

- Results returned by WAIT_CHANNEL:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  TIME_ELAPSED gives the number of clock ticks the process was placed in the waiting queue. This is a 16 bit value, which blocks before overflow. TIME_ELAPSED is only returned if the running process called WAIT_CHANNEL.

### 2.5.2 SIGNAL_CHANNEL

SIGNAL_CHANNEL releases all processes waiting in a channel semaphore's waiting queue. Does nothing if there are no processes waiting (non-counting/no memory !). Due to implementation restrictions, this call may be somewhat slower in response than the SIGNAL call used for normal semaphores (a list of waiting processes will have to be searched).

- Parameters for SIGNAL_CHANNEL:

  CHANNEL (no default) is the channel to receive the signal.

- Results returned by SIGNAL_CHANNEL:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  NR_OF_PROCESSES_STARTED gives the number of processes that were released from the waiting queue by this function call.

## 2.6 Mailboxes

All mailbox messages consist of a user-defined 16, 32 or 64 bit number. Mailboxes come in two forms:

* The so-called 'infinite' mailboxes, where the number of messages that can be stored is only bounded by the MAX_MESSAGES entry in the INIT_MAILBOX call (and the available coprocessor memory, of course). For 'infinite' mailboxes, the FIXED_PART_SIZE parameter in the INIT_MAILBOX function call is an optimalisation problem. If FIXED_PART_SIZE is made too large, then coprocessor memory will be wasted, if it is made too small, then time will be wasted in the dynamic allocation and de-allocation of coprocessor memory to hold the 'overflow' mailbox messages (lowering the coprocessor's performance).

* The so-called 'fixed' mailboxes, where the number of mailbox messages that can be stored is given by the FIXED_PART_SIZE parameter in the INIT_MAILBOX function call. A 'fixed' mailbox has two waiting queues, one for the readers (if the mailbox is empty) and one for the writers (if the mailbox is full). For a 'fixed' mailbox, the number of messages is considered to be the number of messages actually in the mailbox plus the number of waiting processes in the mailbox writers' waiting queue. This total is always checked against the MAX_MESSAGES parameter given with the INIT_MAILBOX function call.

There can be a maximum of four special purpose mailboxes in each multitasking coprocessor:

* The first is the 'system errors mailbox', which is always created following the INIT_SYSTEM function call. This mailbox has 8 byte (64 bit) messages, and is a 'fixed' mailbox with a user specified number of slots (given by the SYSTEM_MAILBOXES parameter for the INIT_SYSTEM function call). The system errors mailbox has a fixed identification number (the ring address of the local coprocessor concatenated with a fixed - dummy - 'memory address'), and can only be accessed by tasks which have the 'System_Task' permission bit set. It is possible to read from - and write into all the system error mailboxes in an interconnected system, although it would be wise to restrict access to reading only from the local system errors mailbox. This mailbox will receive error messages generated automatically by the internal functional blocks in the multitasking coprocessor (if the mailbox is full, then error messages will be lost !).

* The second special purpose mailbox is the 'system interconnection mailbox', where the message size is 8 bytes and the number of slots is user definable, as is the type of the mailbox (FIXED or INFINITE). It also has a known and fixed identification number (different from the system errors mailbox), and can only be accessed by tasks with the 'System_Task' permission bit set. Creating this mailbox is optional, the specifications for it are given by the SYSTEM_MAILBOXES parameter for the INIT_SYSTEM function call. This mailbox is meant to exchange initial messages between system processes running on different multitasking coprocessors (mainly for system startup and management).

* The third and fourth mailboxes are used to exchange messages with processes that control the receiving and transmitting of packets for the 'bridge' functions. These mailboxes have a slot size of 2 bytes and are only created in a MMTCP when the host processor has bridge capabilities. These mailboxes are described in subchapter 2.2: 'Functional Description: Bridge Handling'. Both mailboxes have fixed and known identification numbers and can only be read by local processes that have the 'System_Task' permission bit set TRUE.

### 2.6.1 INIT_MAILBOX

INIT_MAILBOX creates and initialises a mailbox (empty). This function can only be called by a task which has the 'Create_Enable' permission bit set.

- Parameters for INIT_MAILBOX:

MAILBOX_MODE (default given in the INIT_SYSTEM function call) gives the type of mailbox:

* **INFINITE** for an 'infinite' mailbox.

* **FIXED** for a 'fixed-size' mailbox.

FIXED_PART_SIZE (default given in the INIT_SYSTEM function call parameter DEF_FIXED_PART_SIZE) gives the number of slots allocated for the mailbox messages, as described above. The minimum value for this parameter is 1, the maximum value is not yet defined (may depend on the SLOT_SIZE parameter).

SLOT_SIZE (default given in the INIT_SYSTEM function call parameter DEF_SLOT_SIZE) gives the size of the slots for this mailbox (2, 4 or 8 bytes).

GLOBAL (default: TRUE) indicates whether or not the mailbox has to be made globally known throughout the (multiprocessor) system.

READ_QUEUE_MODE (default: FIFO) gives the queueing algorithm used for the mailbox readers' waiting queue:

* **FIFO** for a first-come first-served queueing algorithm.

* **PRIO** for a priority based queueing algorithm, where new waiters are placed in the queue following all waiters with higher or equal priority.

WRITE_QUEUE_MODE (default: FIFO) gives the queueing algorithm for the mailbox writers' queue, with the same possibilities as the mailbox readers' queue (FIFO or PRIO). This parameter is ignored if MAILBOX_MODE is set to INFINITE.

MAX_MESSAGES (default: 65535) gives the number of messages the mailbox may ever store (or the actual number of messages plus the number of waiters in the writers' waiting queue for a FIXED mailbox). This value should not be below the FIXED_PART_SIZE value.

MAX_READERS (default: 65535) sets the maximum number of processes that are allowed to wait for messages in the mailbox readers waiting queue.

- Results returned by INIT_MAILBOX:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

MAILBOX is the identification number to be used in the future when referencing to the new mailbox.

### 2.6.2 TERM_MAILBOX

TERM_MAILBOX deletes a mailbox if the 'deletion holdoff' counter is 0. This function can only be called by a task which has created the mailbox itself, or by a task which has the 'Non_Owned_Delete' permission bit set. All waiters are released and receive a special ERROR_CODE to indicate the mailbox no longer exists.

- Parameters for TERM_MAILBOX:

MAILBOX (no default) indicates which mailbox is to be deleted.

MAX_MESSAGES (default: 65535) gives the maximum number of messages (or the actual number of messages plus the number of waiters in the writers' waiting queue) allowed to enable the deletion.

MAX_WAITERS (default: 65535) gives the maximum number of waiters allowed to be in the mailbox readers' waiting queue to enable the deletion.

- Results returned by TERM_MAILBOX:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.6.3 RECEIVE_MESSAGE

RECEIVE_MESSAGE is called by a process to receive a message from a mailbox. If messages are available, then the message at the head of the message queue is given directly to the calling process, and this message is removed from the mailbox. If there are no messages available, then the process is placed in the mailbox readers' queue. The process is released as soon as it reaches the head of this waiting queue and there is a message sent to the mailbox.

- Parameters for RECEIVE_MESSAGE:

  MAILBOX (no default) is the mailbox to receive the message from.

  MIN_MESSAGES (default: 0) gives the minimum number of messages to be stored in the mailbox to enable the RECEIVE_MESSAGE function.

  MAX_MESSAGES (default: 65535) gives the maximum number of messages allowed in the mailbox to enable the RECEIVE_MESSAGE function.

  MIN_READERS (default: 0) gives the minimum number of waiting processes in the mailbox readers' waiting queue to enable the RECEIVE_MESSAGE function.

  MAX_READERS (default: 65535) gives the maximum number of waiting processes in the mailbox readers' waiting queue to enable the RECEIVE_MESSAGE function.

  TIMEOUT (default: 0 = infinite) gives the maximum number of clock ticks the process will wait in the mailbox readers' waiting queue.

- Results returned by RECEIVE_MESSAGE:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  MESSAGE is the message received from the mailbox. This result is invalid if an error occurred.

  CURRENT_READERS gives the number of waiting processes in the mailbox readers' waiting queue, at the time this process was released from the waiting queue (or the number of waiters at the time of exit from the RECEIVE_MESSAGE call, if the function was not enabled).

  CURRENT_MESSAGES gives the number of messages in the mailbox (or the actual number plus the number of waiters in the writers' waiting queue), at the time this process was released from the waiting queue (or the number of messages at the time of exit from the RECEIVE_MESSAGE call, if the function was not enabled).

  TIME_ELAPSED gives the number of clock ticks the process was placed in the readers' waiting queue. This is a 16 bit value, which blocks before overflow.

### 2.6.4 SEND_MESSAGE

SEND_MESSAGE sends a message to the indicated mailbox. If a fixed-size mailbox is used, and this mailbox is full, then the process is placed in the mailbox writers' waiting queue. The process is released if it is at the head of this queue and a message is read from the mailbox.

- Parameters for SEND_MESSAGE:

  MAILBOX (no default) is the mailbox to send the message to.

  MESSAGE (default: 0) is the message to be placed in the mailbox.

  MIN_MESSAGES (default: 0) gives the minimum number of messages to be stored in the mailbox to enable the SEND_MESSAGE function.

  MAX_MESSAGES (default: 65535) gives the maximum number of messages allowed in the mailbox to enable the SEND_MESSAGE function.

  MIN_READERS (default: 0) gives the minimum number of waiting processes in the mailbox readers' waiting queue to enable the SEND_MESSAGE function.

  MAX_READERS (default: 65535) gives the maximum number of waiting processes in the mailbox readers' waiting queue to enable the SEND_MESSAGE function.

  TIMEOUT (default: 0 = infinite) gives the maximum number of clock ticks the process will wait in the mailbox writers' waiting queue (this parameter is ignored for infinite mailboxes).

- Results returned by SEND_MESSAGE:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  CURRENT_READERS gives the number of waiting processes in the mailbox readers' waiting queue, at the time of the SEND_MESSAGE call.

  CURRENT_MESSAGES gives the number of messages in the mailbox (or the actual number plus the number of waiters in the writers' waiting queue), at the time of the SEND_MESSAGE call.

  TIME_ELAPSED gives the number of clock ticks the process was placed in the writers' waiting queue (0 if it was never placed there). This is a 16 bit value, which blocks before overflow.

### 2.6.5 CHECK_MAILBOX

CHECK_MAILBOX returns the number of messages in a mailbox (or the actual number of messages plus the number of waiters in the writers' waiting queue) and the number of waiters in the readers' waiting queue. This function can also be used to check for the existence of the mailbox.

- Parameters for CHECK_MAILBOX:

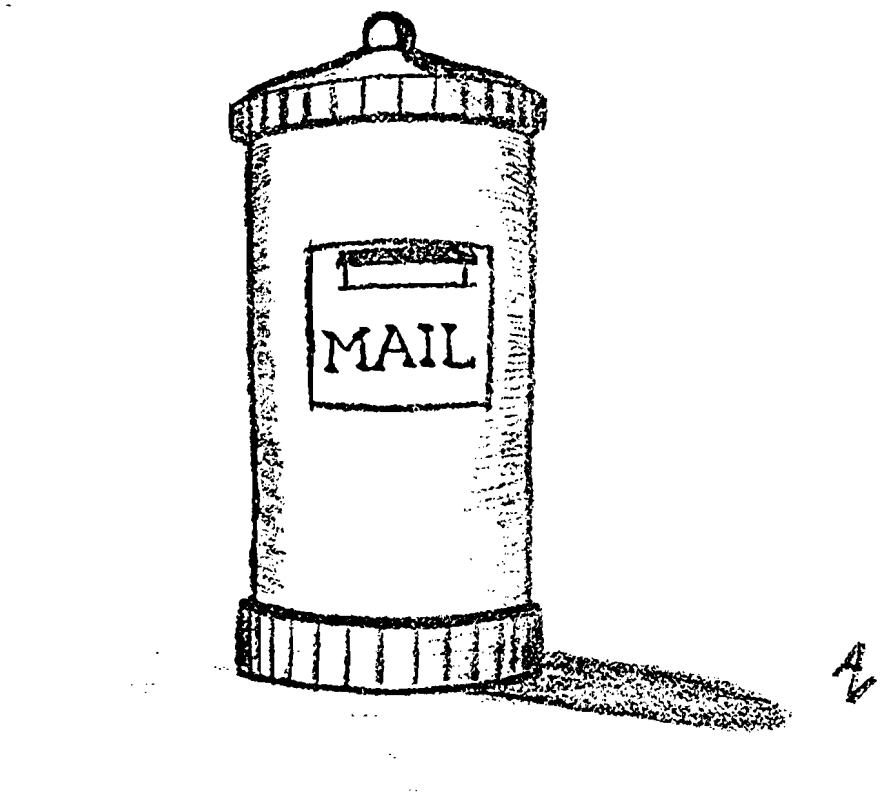
  MAILBOX (no default) is the mailbox to check.

- Results returned by CHECK_MAILBOX:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  CURRENT_MESSAGES is the number of messages in the mailbox (or the actual number of messages plus the number of waiting processes in the mailbox writers' waiting queue) at the time of the CHECK_MAILBOX call.

  CURRENT_READERS is the number of waiters in the mailbox readers' waiting queue at the time of the CHECK_MAILBOX call.



## 2.7 Regions

Regions are a special kind of semaphores. Processes running in a region cannot be deleted, suspended or placed in a waiting queue by another process. Time slicing is disabled for a process executing in a region. Regions should be used with great care, as deadlock can easily occur (UNLINK will then have to be called). Regions can be used by one process at a time. If another process wants to make use of a region, it is placed in the region's requesters waiting queue. The user of a region is always counted as one waiter to facilitate more flexible conditional ENTER_REGION calls.

### 2.7.1 INIT_REGION

INIT_REGION creates and initialises a region. This function can only be called by a task which has the 'Create_Enable' permission bit set.

- Parameters for INIT_REGION:

  MAX_WAITERS (default: 65535) gives the maximum number of waiters (plus the user) for the new region. MAX_WAITERS cannot be 0 !

  GLOBAL (default: TRUE) indicates whether or not the region has to be made globally known throughout the (multiprocessor) system.

  MODE (default: FIFO) gives the queueing algorithm used for the region's requesters waiting queue:

  * **FIFO**: processes wait in first-come first-served order.

  * **PRIO**: processes wait in priority order, new processes are placed behind other processes with equal or higher priority.

- Results returned by INIT_REGION:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  REGION is the identification number to be used in the future when referencing to the new region.

### 2.7.2 TERM_REGION

TERM_REGION deletes a region if the 'deletion holdoff' counter is 0. This function can only be called by a task which has created the region itself, or by a task which has the 'Non_Owned_Delete' permission bit set. All waiters are released and receive a special ERROR_CODE to indicate the region no longer exists. The current region user (if any) does not experience anything (a 'hidden' EXIT_REGION function is executed), but will get the special ERROR_CODE if the EXIT_REGION function is called.

- Parameters for TERM_REGION:

  REGION (no default) indicates which region is to be deleted.

  MAX_WAITERS (default: 0) gives the maximum number of waiters (plus the current user) allowed for the region to enable the deletion. If the default is kept, then the region can only be deleted if the region is not in use.

- Results returned by TERM_REGION:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.7.3 ENTER_REGION

ENTER_REGION is called by a process that wants to use a region. A process cannot request to enter a region it has already entered (trying to do this will return a special ERROR_CODE). If the region is in use, then the number of waiters is checked against the MAX_WAITERS parameters in this call and the INIT_REGION call, and, if the number of waiters is equal to one or both of them, then the function is not executed (returning a special ERROR_CODE). Otherwise, the process is placed in the region requesters waiting queue, and released as soon as the process is at the head of the queue and the user calls EXIT_REGION. If this call is successful, then the process' 'regions entered' counter is incremented by 1.

- Parameters for ENTER_REGION:

  REGION (no default) is the region to use.

  MAX_WAITERS (default: 65535) gives the maximum number of waiting processes in the region requesters' waiting queue (plus the region user) allowed to enable the ENTER_REGION function. If MAX_WAITERS is set to 0, then the region is only entered if immediately available.

  TIMEOUT (default: 0 = infinite) gives the maximum number of clock ticks the process will wait in the region requesters waiting queue.

- Results returned by ENTER_REGION:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  CURRENT_WAITERS gives the number of waiting processes in the region's waiting queue (plus 1, if the region is in use), at the time this process became the region user (or the number of waiters at the time of exit from the WAIT call, if the function was not enabled).

  TIME_ELAPSED gives the number of clock ticks the process was placed in the waiting queue. This is a 16 bit value, which blocks before overflow.

### 2.7.4 EXIT_REGION

EXIT_REGION makes a process exit the specified region, and gives the region to the next waiter in the region's requesters waiting queue (if there are any). If this call is successful, then the 'regions entered' counter for the process is decremented by 1.

- Parameters for EXIT_REGION:

REGION (default: last region entered - LIFO stack !) is the region to exit.

- Results returned by EXIT_REGION:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

CURRENT_WAITERS gives the number of waiting processes in the region's waiting queue (plus 1, if the region is in use), at the time of exit from this call.

### 2.7.5 CHECK_REGION

CHECK_REGION returns the number of waiters (plus 1, if the region is in use) and the task which currently uses the region (if any). This call can also be used to check for the existence of the region.

- Parameters for CHECK_REGION:

REGION (no default) is the region to check.

- Results returned by CHECK_REGION:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

CURRENT_WAITERS gives the number of waiting processes in the region's waiting queue (plus 1, if the region is in use), at the time of exit from this call.

CURRENT_USER gives the task which currently uses the region (if any).

## 2.8 Pipes

Pipes are used to transfer (byte oriented) data between processes. Each pipe can have only one reader (which is always the process that initialised the pipe), but a single process may be reading from several pipes.

There can be several processes that want to write into a pipe, but only one of them can really be the 'current writer' (this is to prevent garbage to be sent into the pipe by simultaneous writes). Processes wanting to write to the pipe should first obtain access by calling CLAIM_PIPE. If the pipe is in use, then the process can optionally be placed in a waiting queue. When a process has finished writing, it can 'pass the pipe' to the next process by calling RELEASE_PIPE.

A single process may be writing into several pipes, and combinations of writing into- and reading from different pipes are also allowed. A process may write into a pipe it is also reading from, but care should be taken to prevent deadlock in this case (which is probably used for testing only).

In a multiprocessor system, these pipes are the data transfer medium of choice, because they work completely transparent (a process does not need to know where the data is coming from or going to). Also, read and write synchronising is fully automatic (the protocol can be completely handled by the coprocessor, task switching synchronises readers and writers). If a process is moved between processors, the data is automatically redirected to the new coprocessor, without the writing processes knowing this.

The UNLINK call does a RELEASE_PIPE for all pipes where the target process was the current writer. Deleting a process does the same, and also deletes all pipes the process was reading from.

Data is sent and received in 'packages' which all have a maximum length, the 'maximum data block length', which probably will be around 32 bytes. This length depends on hardware considerations (number of parameter and result registers needed) and on the protocol used on the interconnecting link between the coprocessors (where a maximum packet length restriction may be in effect).

To allow more flexible conditional calls, the 'current writer' is counted as one pipe requester.

Pipes are addressed differently from other system entities. On the reading side, a pipe has only a logical number, to be used by the RECEIVE_DATA call. This same number can also be used by the reading task for the TERM_PIPE call.

On the writing end, two identifiers are used. For the CLAIM_PIPE call, the pipe to claim is identified by the task that owns and reads the pipe, concatenated with the short identifier used for the pipe by the reading process. CLAIM_PIPE returns a short identification number, to be used for the SEND_DATA call. This same number can also be used by the current writer for the RELEASE_PIPE call.

FLUSH_PIPE and CHECK_PIPE need the same identification number needed by the CLAIM_PIPE call. This number must also be given for RELEASE_PIPE if this call is used by a process which is not the current writer and for the TERM_PIPE call if this call is not used by the reader/owner task.

The number of pipes a task is reading from and the number of pipes a task is the current writer for may be restricted by implementation in the MMTCP. We should reckon with a maximum of 32 pipes for both directions (which should be more than enough).

The following calls are available for pipes:

### 2.8.1 INIT_PIPE

INIT_PIPE initialises a pipe for reading by the calling process.

- Parameters for INIT_PIPE:

  PIPE_LENGTH (default given by the DEF_PIPE_LENGTH parameter in the INIT_SYSTEM call) is the maximum number of bytes the pipe buffer can hold. There is a minimum of at least twice the maximum data block length, and a maximum which has not been determined yet (maybe several kilobytes).

  GLOBAL (default: TRUE) indicates whether or not the pipe has to be made globally accessible throughout the (multiprocessor) system.

  REQUEST_QUEUE_MODE (default: FIFO) gives the queueing algorithm for processes requesting to write into the pipe. This can be FIFO (first come, first served) or PRIO (priority based, as usual). Note that a low priority process can be writing into the pipe while a high priority process is requesting to use the same pipe, writing into a pipe is not automatically denied to allow a higher priority process access to the pipe.

  MAX_REQUESTERS (default: 65535) gives the maximum number of waiting processes in the pipe requesters queue. Because the 'current writer' is counted as a requester, this parameter cannot be 0. If this value is set to 1, then the pipe can only be claimed successfully if it is not in use at the time of the CLAIM_PIPE call.

- Results returned by INIT_PIPE:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  READ_PIPE_ID is an identification number to be used by the pipe reader/owner for RECEIVE_DATA and TERM_PIPE calls. This number is also a part of the full pipe identification number to be used by other tasks.

### 2.8.2 TERM_PIPE

TERM_PIPE is used to delete a pipe. This function can only be called by a task which has created the pipe itself, or by a task which has the 'Non_Owned_Delete' permission bit set. The 'deletion holdoff' counter for the pipe should be 0. All waiters are released and receive a special ERROR_CODE to indicate the pipe no longer exists. The current contents of the pipe buffer is lost. This function is called for each pipe a process is

reading from, if this process is deleted (in this case, the deletion holdoff counter for the pipe is ignored - the process takes precedence).

- Parameters for TERM_PIPE:

  PIPE (no default) indicates the pipe to delete. Note that there are two possibilities to specify the pipe, as described above. The reader/owner need only specify it's short identification number for the pipe (READ_PIPE_ID), all other tasks must add the task identification number of the reader/owner task.

  MAX_REQUESTERS (default: 65535) gives the maximum number of requesters (PLUS the 'current writer') allowed for the pipe to enable the deletion. If 0 is given, then the pipe can only be deleted if noone is writing into it.

  MAX_CONTENTS (default: 65535) gives the maximum number of bytes allowed in the pipe buffer to enable the deletion. Note that this does not include the number of bytes in a data package waiting to be written into the buffer.

- Results returned by TERM_PIPE:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.8.3 CLAIM_PIPE

CLAIM_PIPE is called by a process that wants to write to a pipe. If the pipe is in use, then the process is optionally placed in a waiting queue, as a 'pipe claimer'. Each time RELEASE_PIPE is called for a pipe, the process at the head of the waiting list becomes the 'current writer' and is released from the list.

- Parameters for CLAIM_PIPE:

  PIPE (no default) is the pipe to claim. This parameter consists of the task identification of the pipe reader/owner concatenated with the short identification number (READ_PIPE_ID) used by the reader/owner task to access the pipe.

  MAX_REQUESTERS (default: 65535) gives the maximum number of requesters (PLUS the 'current writer') allowed to enable the call. If this parameter is set to 0, then the pipe is only claimed if it is available immediately.

  TIMEOUT (default: 0 = indefinite) gives the maximum number of clock ticks the process is willing to wait until the claim succeeds. This timeout runs while the process is placed in the requester's waiting queue for the pipe, and does NOT take into account the time needed to send requesting and acknowledging messages back and forth between coprocessors residing on the communication link.

- Results returned by CLAIM_PIPE:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  WRITE_PIPE_ID is a short logical number to be used by the writing task to write into- and release the pipe.

  TIME_ELAPSED gives the number of clock ticks that passed between the moment this call was given and the instance when the process was placed back in the 'ready to run' state. This is a 16 bit number, which blocks before overflow.

### 2.8.4 RECEIVE_DATA

RECEIVE_DATA is called to receive a data block from the specified pipe. This call can only be executed by the process which initialised the pipe with INIT_PIPE. If there are not enough data bytes in the pipe buffer to satisfy the request then the calling process is placed in a waiting state.

- Parameters for RECEIVE_DATA:

  READ_PIPE_ID (no default) is the pipe to receive the data from. This parameter can only be the short identification number used by the reader/owner task (which must be the task that called this function).

  MIN_LENGTH (default: 1) gives the minimum number of data bytes to be received. If there are less data bytes available than specified with this parameter, then the calling process is placed in a waiting state until there are at least this specified number of bytes available in the buffer.

  MAX_LENGTH (default: maximum data block length) gives the maximum number of data bytes the calling process is willing to receive from the pipe buffer. This call will never return more data bytes than specified by this parameter. This parameter should be at least 1, never below MIN_LENGTH, and never more than the maximum data block length.

  TIMEOUT (default: 0 = indefinite) gives the maximum number of clock ticks the calling process is willing to wait until there are enough bytes in the buffer to satisfy the request.

- Results returned by RECEIVE_DATA:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  BYTES_RECEIVED gives the actual number of bytes received. If this value is 0, then it indicates that there were no bytes available at the time of the call (this is only possible if the MIN_LENGTH parameter was set to 0, or if an error occurred).

  DATA_BLOCK contains the data bytes retrieved from the pipe buffer.

TIME_ELAPSED gives the number of clock ticks the calling process was placed in a waiting state waiting for bytes to arrive. This is a 16 bit number, which blocks before overflow.

### 2.8.5 SEND_DATA

SEND_DATA is used to send data to a specified pipe. If the pipe is filled to such a level that the data block sent by this call cannot be stored in the pipe buffer, then the calling process is placed in a waiting state until the reading process has read enough bytes from the buffer to store the complete data block in the buffer. Note that the calling process must be the 'current writer' for the specified pipe (should have called CLAIM_PIPE successfully).

- Parameters for SEND_DATA:

  WRITE_PIPE_ID (no default) is the pipe to send the data to. This parameter is the short identification number returned by the CLAIM_PIPE function call.

  DATA_BLOCK_LENGTH (no default) gives the number of data bytes to be sent to the pipe with this call. The minimum value for this parameter is 1, the maximum value is the maximum data block length.

  DATA_BLOCK (no default) is the data block to be sent to the specified pipe.

  TIMEOUT (default: 0 = indefinite) gives the maximum number of clock ticks the process is willing to wait until the data block is placed in the pipe. This time does NOT include the time the coprocessor needs to send the data package out on the communication link to the other coprocessors, if the data is sent to a pipe residing in another coprocessor.

- Results returned by SEND_DATA:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  TIME_ELAPSED gives the number of clock ticks that passed between the issuing of the call and the moment that the data was stored in the pipe buffer (in the case of a remote buffer, the moment that the local coprocessor received notification from the other side that this was done). This is a 16 bit number, which blocks before overflow.

### 2.8.6 RELEASE_PIPE

RELEASE_PIPE is called to release the pipe and allow the process at the head of requester's queue to become the new 'current writer' for the pipe. Note that this call can be executed by <u>any</u> process in the system. This call it is executed for all pipes a process is current writer for if this process is deleted or is the target process for an UNLINK call.

- Parameters for RELEASE_PIPE:

  PIPE (default: last pipe requested - LIFO algorithm) is the pipe to be released. The current writer need only specify it's WRITE_PIPE_ID number to identify the pipe to release, all other tasks should give the task identifier for the reader/owner task concatenated with the READ_PIPE_ID number used by that task to access the pipe.

- Results returned by RELEASE_PIPE:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.8.7 FLUSH_PIPE

FLUSH_PIPE is called to remove all data from the specified pipe's buffer. If there was a data block waiting to be written to the buffer, it will be written immediately after this call has done it's work. This call should be regarded as a diagnostic/corrective tool, not as a standard call that is used regularly. This call can be executed only by processes that are either the owner of the pipe, the current writer for the pipe or have the 'Non_C_W_Flush' permission bit set TRUE.

- Parameters for FLUSH_PIPE:

  PIPE (no default) is the pipe to have it's buffer cleared. The full pipe identification number should be used for this parameter (reader/owner task identifier concatenated with the corresponding READ_PIPE_ID number).

- Results returned by FLUSH_PIPE:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.8.8 CHECK_PIPE

CHECK_PIPE is used to retrieve the current status of a specified pipe. It can also be used to check for the actual existence of a pipe.

- Parameters for CHECK_PIPE:

  PIPE (no default) gives the pipe to be checked. The full pipe identification number should be used for this parameter (reader/owner task identifier concatenated with the corresponding READ_PIPE_ID number).

- Results returned by CHECK_PIPE:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

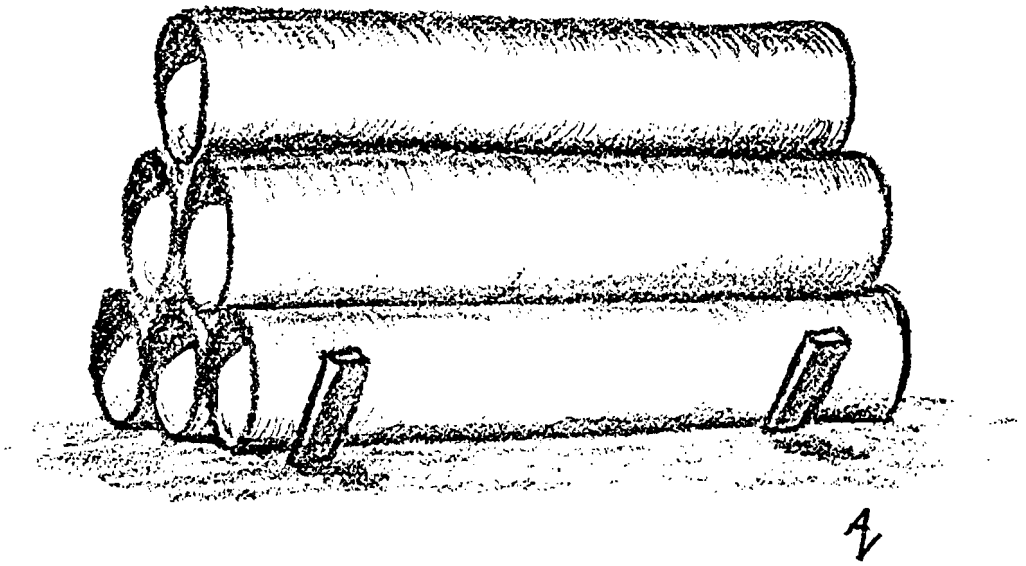
CURRENT_CONTENTS gives the number of bytes in the pipe buffer when this CHECK_PIPE call was received by the coprocessor which controls the pipe.

WAITING_CONTENTS gives the number of bytes in a data block which has been sent, but is not yet stored in the pipe buffer (if this value is 0, then there is no such data block). This status parameter is checked at the same time as the CURRENT_CONTENTS parameter.

PIPE_READER is the identification code for the reading process for the specified pipe.

CURRENT_REQUESTERS gives the current number of waiting processes in the requesters queue, PLUS 1 if the pipe has a 'current writer' process.

CURRENT_WRITER gives the identification code for the process which has write access to the specified pipe. This result is only valid if CURRENT_REQUESTERS is at least 1.



## 2.9 Interrupts

Interrupts come in two different types:

1) **'Direct coupled'** interrupts are connected directly to the host processor's interrupt structure. These interrupt inputs are meant for very time critical tasks and processor hardware related functions. The interrupt routines started by these interrupts need only save and restore those parts of the processor environment which they change, as no direct task switching can be done. The TRIG_INT_SEMAPHORE function can be used by these routines to send a single unit to a specified semaphore (using the same method as used for the indirect coupled interrupts described below), all other coprocessor functions are not available.

2) **'Indirect coupled'** interrupts are connected to the multitasking coprocessor's interrupt inputs. These interrupts send a unit to one of two internal semaphores. The normal interrupt semaphore receives the unit if there are not more than a specified maximum number of units accumulated in the semaphore's units counter, or if there is no 'alternate' interrupt semaphore specified. The 'alternate' interrupt semaphore receives the unit if the associated normal interrupt semaphore has received the maximum number of units (this can be used to start error recovery processes). The priority of an interrupt input is simply the priority of the process waiting at the interrupt semaphores. If an indirect coupled interrupt releases a process with sufficient priority to preempt the running process, then the 'force task switching' interrupt output to the host processor is activated.

NOTES:

Interrupts are ignored if they would increment the units counter of the specified normal or alternate interrupt semaphores above the MAX_UNITS parameter given with the INIT_SEMAPHORE function call that created these semaphores.

A normal or alternate interrupt semaphore should not be deleted (the existence of a semaphore is only checked during the SET_INT_SEMAPHORE and SET_ALT_INT_SEMAPHORE calls). Units generated by an interrupt which are sent to non-existent semaphores will be lost. It may be an idea to prevent deletion of an (alternate-) interrupt semaphore by calling DISABLE_DELETION.

It is possible to designate non-local semaphores to be the normal or alternate interrupt semaphores, as long as these semaphores are located within MMTCP's connected to the local token ring. In this case, the MMTCP will automatically send a message to the remote MMTCP to increment the remote semaphore's units counter. Sending this message will introduce an extra delay, which should be reckoned with !

If errors are encountered by the coprocessor while processing an interrupt, it has no way to inform the user software what happened. Partly for this reason, the alternate semaphores were introduced, but these do not provide much help if they encounter errors themselves. For this reason, errors will be formatted into messages and written into the 'system errors' mailbox described in subchapter 2.6: 'Functional Description: Mailboxes'.

These functions can only be used to change the local coprocessor's interrupt input assignments.

### 2.9.1 SET_INT_SEMAPHORE

SET_INT_SEMAPHORE assigns a normal semaphore to a specified interrupt input (LINE_NUMBER). This function can only be called by a task which has the 'System_Task' permission flag set TRUE.

- Parameters for SET_INT_SEMAPHORE:

  LINE_NUMBER (no default) is the interrupt input to which the semaphore is to be connected.

  SEMAPHORE (default: NONE) is the semaphore to be connected to the interrupt input. If the default is kept, then the interrupt input is left without a normal semaphore to send units to.

  MAX_UNITS (default: 0) is the maximum number of units allowed in the semaphore's counter to send the unit to the normal interrupt semaphore - this maximum has no effect if there is no alternate interrupt semaphore specified.

- Results returned by SET_INT_SEMAPHORE:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.9.2 SET_ALT_INT_SEMAPHORE

SET_ALT_INT_SEMAPHORE assigns an alternate semaphore to the specified interrupt input. This function can only be called by a task which has the 'System_Task' permission flag set TRUE.

- Parameters for SET_ALT_INT_SEMAPHORE:

  LINE_NUMBER (no default) is the interrupt input to which the alternate semaphore is to be connected.

  SEMAPHORE (default: NONE) is the alternate semaphore to be connected to the interrupt input. If the default is kept, then the interrupt input is left without an alternate semaphore to send units to.

- Results returned by SET_ALT_INT_SEMAPHORE:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.9.3 ENABLE_INTERRUPT

ENABLE_INTERRUPT enables the specified interrupt input and optionally specifies the active input level and/or edge triggering. This function can only be called by a task which has the 'System_Task' permission flag set TRUE. Note that an interrupt line need not be enabled for the TRIG_INT_SEMAPHORE function call.

- Parameters for ENABLE_INTERRUPT:

LINE_NUMBER (no default) is the interrupt input to be enabled.

TRIGGER_MODE (default: old setting, setting after INIT_SYSTEM is level triggered, active high) gives the hardware trigger mode for the specified line number. If this parameter is given and edge triggering is specified with it, then the edge detect flipflop for the specified input line is reset automatically (to prevent false interrupts). The following trigger modes are possible:

| | | |
|---|---|---|
| 1) | Level triggered, | active level high (default). |
| 2) | Level triggered, | active level low. |
| 3) | Edge triggered, | active edge low -> high. |
| 4) | Edge triggered, | active edge high -> low. |

- Results returned by ENABLE_INTERRUPT:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.9.4 DISABLE_INTERRUPT

DISABLE_INTERRUPT disables the specified interrupt input. This function can only be called by a task which has the 'System_Task' permission flag set TRUE.

- Parameters for DISABLE_INTERRUPT:

LINE_NUMBER (no default) is the interrupt input to be disabled.

- Results returned by DISABLE_INTERRUPT:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.9.5 TRIG_INT_SEMAPHORE

TRIG_INT_SEMAPHORE sends a single unit to one of two specified semaphores, using the same protocols as used by the hardware interrupt inputs on the MMTCP. This command can always be issued, and is issued by writing to a special register in the coprocessor's interface register structure. No result is returned to the task executing this function.

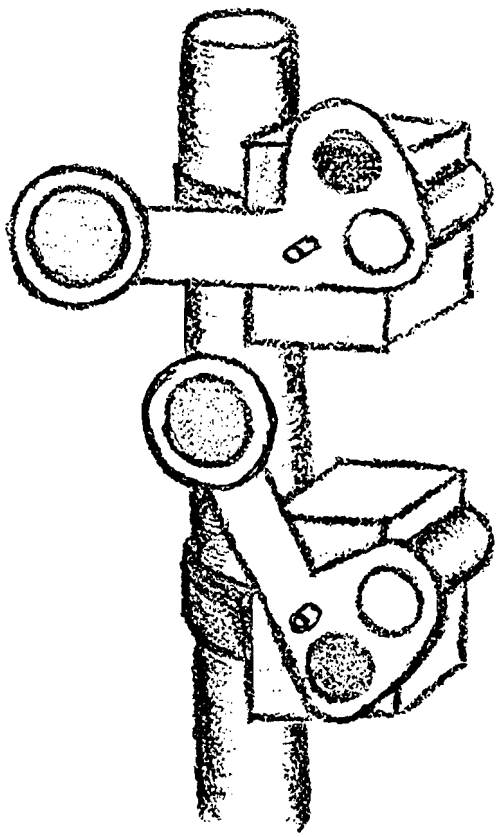
This function call is handled internally as if one of the external interrupt lines was made active. This means that TRIG_INT_SEMAPHORE has the same possibilities as the normal hardware interrupts, and uses exactly the same protocols as described above. Any unused interrupt input line can be used for the handling of this function, and need not even be enabled to do so. Keeping the interrupt line disabled prevents spurious interrupts caused by input noise, and may also shorten the cycle time for the interrupt scanner.

- Parameters for TRIG_INT_SEMAPHORE:

  LINE_NUMBER (no default) is the number of the external interrupt line input that is to be (software) triggered, so that the normal MMTCP hardware interrupt processing can take place. If the LINE_NUMBER is out of range or other errors occur, then an error message is written to the system errors mailbox.

- Results returned by TRIG_INT_SEMAPHORE:

  NONE



## 2.10 Real time clock/Power handling

Assuming a real time clock with date and time can be integrated on the coprocessor chip (together with some logic to control a 'power on' output), the following functions can be added:

### 2.10.1 SET_RTC

SET_RTC sets the date and time in the real time clock counters. This function can only be called by a task which has the 'System_Task' permission flag set TRUE.

- Parameters for SET_RTC:

  DATE (no default) is the new time to load in the real time clock counters.

  TIME (no default) is the new date to load in the real time clock counters.

- Results returned by SET_RTC:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.10.2 READ_RTC

READ_RTC obtains the current date and time from the real time clock counters.

- Parameters for READ_RTC:

  NONE

- Results returned by READ_RTC:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  DATE, which is the date at the time of the call.

  TIME, which is the time at the time of the call.

### 2.10.3 WAIT_RTC

WAIT_RTC places the specified process in a waiting queue until the real time clock counters match the given date and time (making it possible to schedule events years before they happen). If the power is turned off at that time, then the 'power on' output will be activated by the MMTCP.

- Parameters for WAIT_RTC:

  PROCESS (default: running process) is the process to wait for the real time clock match. This process should be in the ready to run state (the running process always is), and not in a region if it is not the running process. The Stop_Other_Tasks permission bit should be set if the specified process is not a direct child of the calling process, nor the calling process itself. If the running process is not specified here, then the specified process will not receive an ERROR_CODE resulting from this call, but will receive the ERROR_CODE it was already set to receive (if any - otherwise this call will behave as a very long normal hardware interrupt).

  DATE (default: current date) is the date on which the specified process will be released from the waiting queue.

  TIME (no default) is the time on which the specified process will be released from the waiting queue.

- Results returned by WAIT_RTC:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.10.4 POWER_OFF

POWER_OFF can be used to de-activate the 'power on' output pin from the coprocessor. The coprocessor goes 'asleep' to preserve power, and power can be re-instated by a real time clock timer match or external hardware (using a bidirectional 'power on' pin or one of the interrupt inputs). This function can only be called by a task which has the 'System_Task' permission flag set TRUE.

- Parameters for POWER_OFF:

  NONE

- Results returned by POWER_OFF:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is. This error code is returned after the power is turned on again, and indicates why the power is turned on. If this was due to a timer match, then the host processor can find the environment pointer for the process to start in the normal environment pointer registers.

## 2.11 Deletion control

All processes, (normal) semaphores, mailboxes, regions and pipes have 'deletion holdoff' counters. If these counters are non-zero, then the entities they belong to cannot be deleted by the normal delete functions. In the following function calls, 'ID' is the identification for any one of such entities (contains the type of entity and the identification number).

### 2.11.1 DISABLE_DELETION

DISABLE_DELETION increments the 'deletion holdoff' counter for the specified entity. An error occurs if the counter would wrap around as a result of this action. This function can only be called by a task which has created the entity itself (including child tasks), by a task for itself, or by a task which has the 'System_Task' permission bit set.

- Parameters for DISABLE_DELETION:

  ID (no default) is the entity to have it's 'deletion holdoff' counter incremented.

- Results returned by DISABLE_DELETION:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.11.2 ENABLE_DELETION

ENABLE_DELETION decrements the 'deletion holdoff' counter for the specified entity. An error occurs if the counter would wrap around as a result of this action. This function can only be called by a task which has created the entity itself (including child tasks), by a task for itself, or by a task which has the 'System_Task' permission bit set.

- Parameters for ENABLE_DELETION:

  ID (no default) is the entity to have it's 'deletion holdoff' counter decremented.

- Results returned by ENABLE_DELETION:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.11.3 FORCE_DELETE

FORCE_DELETE deletes an entity if the 'deletion holdoff' counter's value is equal or below the given maximum. Processes executing in a region cannot be deleted by this call (they have to be 'UNLINK'-ed first). This function can only be called by a task which has created the entity itself (including child tasks), by a task for itself, or by a task which has the 'System_Task' permission bit set.

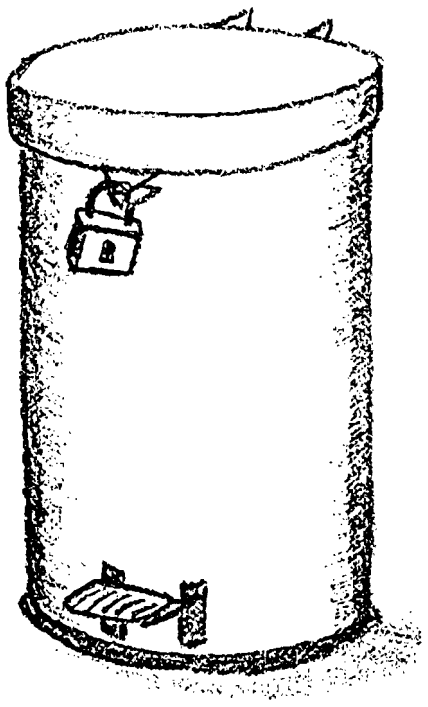
- Parameters for FORCE_DELETE:

ID (no default) is the entity to delete.

MAX_LEVEL (default: 1) is the maximum count allowed in the 'deletion holdoff' counter of the entity to enable the deletion.

- Results returned by FORCE_DELETE:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.



## 2.12 Stream Data Transfer

To be able to use the MMTCP controlled token ring networks to transfer large blocks of data, some kind of special hardware/software interface will have to be provided. For this purpose, I have introduced the concept of 'stream data' transfer in subchapter 1.4.5: 'MMTCP 'file transfer'.

Stream data transfer can cross ring boundaries, making transparent use of bridges and interconnecting networks (the stream control tasks need not be aware of this). If possible, we will make use of standard high level protocols, so that it will be possible to exchange stream data between an MMTCP and a non-MMTCP LAN controller.

Stream data transfer can be used to move process texts and data areas between host processors in case of a process transfer. In this case, the request originates from the scheduler process(es), which are part of the local operating system.

Because data stream handling involves hardware located in the host processor system (standard DMA controller chips), special tasks should be used to control and monitor the transfers. These tasks have to arbitrate between the stream data transfer requests of other tasks, initialise the DMA controller chip(s), set up and monitor the actual transfer of the data, and report the results back to the requesting tasks. These tasks (which I call a 'stream control tasks') have to use the function calls described below, and should have the 'Stream_Control' permission flag set TRUE.

Note that stream data should be enabled by the INIT_SYSTEM function call, otherwise all functions in this subchapter become illegal.

Stream data transfer goes through several phases:

**Initialisation.** The stream control tasks at both ends should inform eachother about the number of data bytes to transfer and the identification number they are going to use for that particular data stream (the 'stream token'). Then they can call SEND_STREAM and RECEIVE_STREAM. The MMTCP will then place these tasks in a waiting state while a connection is build and initialisation packets are exchanged. The tasks are made ready to run again as soon as the transfer is started or an error is detected.

**Data transfer.** The actual data transfer is done without any involvement of the stream control tasks. At the sending end of the data link, the MMTCP reads data from it's host memory with a standard DMA interface, divides the data stream into data packets, and handles all the protocols attached to the transmission. At the other end of the link, the receiving MMTCP re-assembles the data packets into a continuous data stream and uses DMA to store the data bytes in the receiving host's memory.

**Progress monitoring and result handling.** The stream control tasks can call CHECK_STREAM to monitor the progress of the data transfer. Alternatively, they can call AWAIT_STREAM_END to place themselves in a waiting state until the data has been transferred or a fatal error has occurred. The result of the data transfer can be reported back to the requesting tasks in the most suitable form, either synchronously (using mailboxes) or asynchronously (setting status flags).

The following function calls are used for stream data management:

### 2.12.1 SEND_STREAM

SEND_STREAM opens a data stream for sending.

- Parameters for SEND_STREAM:

  CHANNEL (no default) gives the number of the DMA channel to use. The range for this parameter is determined by the number of hardware DMA channels available to the coprocessor in the local host processor system (this number is set by the STREAM_CONFIGURATION parameter for the INIT_SYSTEM function call). If the channel is already in use at the time this command is given, then an error will result.

  DESTINATION (no default) gives the identification number of the coprocessor (or other LAN controller) the data stream should be sent to.

  STREAM_TOKEN (default: 0) is a number identifying this particular data stream if there are more data streams going from this MMTCP to the destination. The default stream token can be used if there are no parallel data streams between this coprocessor and the destination.

  STREAM_LENGTH (no default for the low word, default for the high word = 0) is the (32 bits) count of data bytes that comprise the total length of the data stream. A value of zero for STREAM_LENGTH is not allowed. Using 32 bits gives a maximum STREAM_LENGTH of 4 gigabytes !

  TIMEOUT (default given by the STREAM_INIT_TIMEOUT parameter for the INIT_SYSTEM function call) gives the number of clock ticks the stream control task is willing to wait until the stream data transfer link is set up. The waiting period cannot be infinite.

- Results returned by SEND_STREAM:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.12.2 RECEIVE_STREAM

RECEIVE_STREAM opens a data stream for receiving.

- Parameters for RECEIVE_STREAM:

  CHANNEL (no default) gives the number of the DMA channel to use. The range for this parameter is determined by the number of hardware DMA channels available to the coprocessor in the local host processor system (this number is set by the STREAM_CONFIGURATION parameter for the INIT_SYSTEM function call). If the channel is already in use at the time this command is given, then an error will result.

SOURCE (no default) gives the identification number of the coprocessor (or other LAN controller) the data stream should be received from.

STREAM_TOKEN (default: 0) is a number identifying this particular data stream if there are more data streams received by this MMTCP coming from the specified source. The default stream token can be used if there are no parallel data streams between this coprocessor and the source.

STREAM_LENGTH (no default for the low word, default for the high word = 0) is the (32 bits) count of data bytes that comprise the total length of the data stream. A value of zero for STREAM_LENGTH is not allowed.

TIMEOUT (default given by the STREAM_INIT_TIMEOUT parameter for the INIT_SYSTEM function call) gives the number of clock ticks the stream control task is willing to wait until the stream data transfer link is set up. The waiting period cannot be infinite.

- Results returned by RECEIVE_STREAM:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.12.3 CHECK_STREAM

CHECK_STREAM can be used to provide status information regarding the data streams sent or received by the local MMTCP. Note that the results returned by this function call are valid for the time this call was given, and that the status of a stream may have changed since then (if a higher priority task has interrupted the calling task).

- Parameters for CHECK_STREAM:

CHANNEL gives the DMA channel number for which the status is checked. The range for this parameter is determined by the number of hardware DMA channels available to the coprocessor in the local host processor system (this number is set by the STREAM_CONFIGURATION parameter for the INIT_SYSTEM function call).

- Results returned by CHECK_STREAM:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is. The only error indicated here is 'channel number out of range'.

STATUS is a code indicating the current status of the data stream:

**IDLE:** not in use at this moment.

**INIT_READ:** waiting for stream initialisation at the reading end of the data link.

**INIT_WRITE:** waiting for stream initialisation at the sending end of the data link.

**TRANSFER_READ:** transfer in progress, this is the receiving end of the data link.

**TRANSFER_WRITE:** transfer in progress, this is the sending end of the data link.

**END_READ:** data stream received without error, AWAIT_STREAM_END has not been called.

**END_WRITE:** data stream sent without error, AWAIT_STREAM_END has not been called.

**ERROR_READ:** data stream reception aborted due to fatal error, AWAIT_STREAM_END has not been called.

**ERROR_WRITE:** data stream sending aborted due to fatal error, AWAIT_STREAM_END has not been called.

**ABORTED_READ:** data stream reception aborted with ABORT_STREAM, AWAIT_STREAM_END has not been called.

**ABORTED_WRITE:** data stream sending aborted with ABORT_STREAM, AWAIT_STREAM_END has not been called.

If the status is any of the 'end', 'error' or 'aborted' states, then the stream status will automatically be changed into 'idle' following this call.

STREAM_LENGTH gives the total number of data bytes originally requested to be transferred. This result should be ignored if STATUS is IDLE.

BYTES_TO_GO gives the number of data bytes still left to be transferred at the time this function was called. This result should be ignored if STATUS is IDLE.

STREAM_TOKEN gives the token for the data stream used by the requested channel. This result should be ignored if STATUS is IDLE.

LINK_PARTNER gives the identification number for the MMTCP (or other LAN controller) that is handling the 'other end' of the data stream. This result should be ignored if STATUS is IDLE.

### 2.12.4 AWAIT_STREAM_END

AWAIT_STREAM_END lets the stream control task wait until the stream transfer is finished. This function call is disallowed if the status of the channel is IDLE (see above). There can only be one process waiting for the end of a data stream (there is no queueing mechanism). A timeout cannot be given here, automatic timeouts will be generated by the MMTCP hardware stream protocol handlers in case of transmission errors.

- Parameters for AWAIT_STREAM_END:

  CHANNEL gives the channel for which the end of the data stream is awaited. The range for this parameter is determined by the number of hardware DMA channels available to the coprocessor in the local host processor system (this number is set by the STREAM_CONFIGURATION parameter for the INIT_SYSTEM function call).

- Results returned by AWAIT_STREAM_END:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  BYTES_TRANSFERRED gives the actual number of data bytes transferred in the data stream. Normally, this will equal the number of data bytes requested by the STREAM_LENGTH parameters in both the SEND_STREAM and RECEIVE_STREAM function calls.

### 2.12.5 ABORT_STREAM

ABORT_STREAM is used to abort a data stream in case of 'emergency' (whatever the user defines with this word). This function call should be used with great care, and is not intended to be used as part of the normal operations. Any stream data packets in transit will be finished (no hardware packet abort done).

- Parameters for ABORT_STREAM:

  CHANNEL gives the DMA channel number for which the data stream is aborted. The range for this parameter is determined by the number of hardware DMA channels available to the coprocessor in the local host processor system (this number is set by the STREAM_CONFIGURATION parameter for the INIT_SYSTEM function call).

- Results returned by ABORT_STREAM:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

## 2.13 Virtual memory/Multiprocessor support

'Support' should be written in capitals, as the MMTCP makes no attempt to do the virtual memory or multiprocessor management itself. It is assumed that one or more 'scheduler' processes are running in the host processor's kernel. These scheduler processes decide which processes to swap out of memory and which processes to swap in from the background storage. They also decide when to transfer processes from one processor to another processor in a multiprocessor multitasking system.

To help these scheduling processes to make their decisions, the following constructs are provided:

* All processes stored in the memory of a single coprocessor are stored in a linked list. This list is sorted on decreasing priority and - in each group of equal-priority processes - on increasing (user-defined) 'size'. The schedulers can traverse this linked list (in both directions) with several functions described below. All processes have a 'dirty' flag, which is set when a process is being manipulated by a scheduler. If the dirty flag is set, then the process cannot be made running.

* All processes have a 'use alternate ready queue' flag. This is not a real queue, but is only used to signal that the process is not executable (swapped out, or in the process of swapping out or in). The scheduler waiting for processes to swap into memory can wait at a pseudo-semaphore for processes that are made ready while they have this flag set.

* All processes have 'sent-broadcasted' and 'received-broadcasted' flags. These are used in a multiprocessor system, to indicate a process is available to be moved between processors. A scheduler deciding that the local processor gets overloaded can instruct the MMTCP to 'broadcast' a task descriptor to the other MMTCP's on the local ring. The task descriptor is locally marked 'sent-broadcasted', while it is stored in the process queues of all other MMTCP's with the 'received-broadcasted' flag set. A scheduler deciding to load the broadcasted process in the local memory can 'claim' the process. If this claim is honoured, then the task descriptor is removed from all other MMTCP process lists, and all MMTCP data structures belonging to the task are transferred to the new location (including all pipes owned by the task - this is done automatically). The process text and data areas should be transferred too, after which the process can be made running on the new host processor.

* Several entries in the task descriptor can be used to control the searching conducted by the coprocessor in the process list. These entries are collectively named 'STEERING_FLAGS' in the functions described below. The following entries can be specified:

  - **RETURN_IF_NOT_FOUND** (default: FALSE) makes the function return immediately if no process is found.

  - **TIMEOUT** (default: 0 = infinite) gives the number of clock ticks the scheduler is willing to wait until a process in the process list gets the required set of parameters.

- **STOP_AT_DIRTY_PROCESS** (default: TRUE) stops the search if a process is found that is marked 'dirty' (otherwise this process is skipped).

- **FIND_LOCKED_TASKS** (default: FALSE) must be set TRUE to be able to find processes with the LOCAL_LOCK status bit set TRUE.

- **MIN_PRIORITY** (default: 0) is the minimum allowed process priority.

- **MAX_PRIORITY** (default: 65535) is the maximum allowed priority.

- **MIN_SIZE** (default: 0) is the minimum allowed used-defined process 'size'. This value is stored in the user data area of the task descriptors (see 2.3.7: 'CHANGE_USER_AREA').

- **MAX_SIZE** (default: 65535) is the maximum allowed user-defined process 'size'.

- **MIN_STATE** (default: 0) is the minimum allowed user-defined process 'state', stored in the user data area of the task descriptors (see 2.3.7: 'CHANGE_USER_AREA'). This is not the 'special status' word.

- **MAX_STATE** (default: 65535) is the maximum allowed user-defined process 'state'.

- **MIN_WAITING_TIME** (default: 0) is the minimum allowed number of clock ticks the process has to wait until a timeout occurs (if any is set, otherwise this entry is ignored).

- **MAX_WAITING_TIME** (default: 65535) is the maximum allowed number of clock ticks the process has to wait until a timeout occurs (if any is set, otherwise this entry is ignored).

Some bit flags, all default to FALSE:

- **NORMAL_READY_QUEUE_ONLY**
- **ALTERNATE_READY_QUEUE_ONLY**
- **MUST_BE_SUSPENDED**
- **SHOULD_NOT_BE_SUSPENDED**
- **LOCAL_PROCESSES_ONLY**
- **BROADCASTED_PROCESSES_ONLY**
- **READY_PROCESSES_ONLY**
- **NON_READY_PROCESSES_ONLY**

The following functions are provided for virtual memory and multiprocessor support (a process calling these functions must have the 'Scheduler' permission flag set TRUE):

### 2.13.1 SET_ALTERNATE_READY_QUEUE

SET_ALTERNATE_READY_QUEUE sets the 'use alternate ready queue' flag for the specified process. The process is removed from the normal ready queue if it was located there. Error occurs if the process had the 'use alternate ready queue' flag already set.

- Parameters for SET_ALTERNATE_READY_QUEUE:

  PROCESS (no default) is the process to be transferred to the alternate ready queue. This cannot be the running process.

- Results returned by SET_ALTERNATE_READY_QUEUE:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.13.2 SET_NORMAL_READY_QUEUE

SET_NORMAL_READY_QUEUE resets the 'use alternate ready queue' flag for the specified process. The process is inserted in the normal ready queue if the process state indicates that the process is ready to run. Error occurs if the process had the 'use alternate ready queue' flag already reset.

- Parameters for SET_NORMAL_READY_QUEUE:

  PROCESS (no default) is the process to be transferred to the normal ready queue. This cannot be the running process.

- Results returned by SET_NORMAL_READY_QUEUE:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.13.3 MARK_DIRTY

MARK_DIRTY marks a process 'dirty'. Error occurs if the process was already marked 'dirty'.

- Parameters for MARK_DIRTY:

  PROCESS (no default) is the process to mark. This cannot be the running process.

- Results returned by MARK_DIRTY:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.13.4 UNMARK_DIRTY

UNMARK_DIRTY clears the 'dirty' flag for the specified process. Error occurs if the process was not marked 'dirty'.

- Parameters for UNMARK_DIRTY:

PROCESS (no default) is the process to un-mark. This cannot be the running process.

- Results returned by UNMARK_DIRTY:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

### 2.13.5 GET_PROCLIST_HEAD

GET_PROCLIST_HEAD searches the process list, starting from the head (with the highest priority) towards the end, controlled by the 'STEERING_FLAGS' described above. If a process is found, MARK_DIRTY is called automatically for this process (indivisible action). If no process is found during the first search, and RETURN_IF_NOT_FOUND is FALSE, then all processes changing state are checked while waiting.

- Parameters for GET_PROCLIST_HEAD:

STEERING_FLAGS, as described above.

- Results returned by GET_PROCLIST_HEAD:

ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

PROCESS is the process found, if any.

TIME_ELAPSED gives the number of clock ticks the scheduler process was placed in the waiting state waiting for a process fitting the STEERING_FLAGS. This is a 16 bit value, which blocks before overflow.

### 2.13.6 GET_PROCLIST_TAIL

GET_PROCLIST_TAIL searches the process list, starting from the tail (with the lowest priority) towards the start, controlled by the 'STEERING_FLAGS' described above. If a process is found, MARK_DIRTY is called automatically for this process (indivisible action). If no process is found during the search, and RETURN_IF_NOT_FOUND is FALSE, then all processes changing state are checked while waiting.

- Parameters for GET_PROCLIST_TAIL:

STEERING_FLAGS, as described above.

- Results returned by GET_PROCLIST_TAIL:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  PROCESS is the process found, if any.

  TIME_ELAPSED gives the number of clock ticks the scheduler process was placed in the waiting state waiting for a process fitting the STEERING_FLAGS. This is a 16 bit value, which blocks before overflow.

### 2.13.7 GET_PROCLIST_NEXT

GET_PROCLIST_NEXT searches the process list, starting from the given process towards the end, controlled by the 'STEERING_FLAGS' described above. If UNMARK_IT is set TRUE, then UNMARK_DIRTY is called for the given process. If a process is found, MARK_DIRTY is called automatically for this process (indivisible action). If no process is found during the first search, and RETURN_IF_NOT_FOUND is FALSE, then all processes changing state are checked while waiting - including the processes stored in front of the given process.

- Parameters for GET_PROCLIST_NEXT:

  PROCESS (no default) is the process to start the search from (this process itself is not included in the first search).

  UNMARK_IT (default: TRUE) controls whether or not the given PROCESS is to be UNMARK-ed.

  STEERING_FLAGS, as described above.

- Results returned by GET_PROCLIST_NEXT:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  PROCESS is the process found, if any.

  TIME_ELAPSED gives the number of clock ticks the scheduler process was placed in the waiting state waiting for a process fitting the STEERING_FLAGS. This is a 16 bit value, which blocks before overflow.

### 2.13.8 GET_PROCLIST_PREV

GET_PROCLIST_PREV searches the process list, starting from the given process towards the start, controlled by the 'STEERING_FLAGS' described above. If UNMARK_IT is set TRUE, then UNMARK_DIRTY is called for the given process. If a process is found, MARK_DIRTY is called automatically for this process. If no process is found during the first search, and RETURN_IF_NOT_FOUND is FALSE, then all processes changing

state are checked while waiting - including the processes stored following the given process.

- Parameters for GET_PROCLIST_PREV:

  PROCESS (no default) is the process to start the search from (this process itself is not included in the first search).

  UNMARK_IT (default: TRUE) controls whether or not the given PROCESS is to be UNMARK-ed.

  STEERING_FLAGS, as described above.

- Results returned by GET_PROCLIST_PREV:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  PROCESS is the process found, if any.

  TIME_ELAPSED gives the number of clock ticks the scheduler process was placed in the waiting state waiting for a process fitting the STEERING_FLAGS. This is a 16 bit value, which blocks before overflow.

### 2.13.9 BROADCAST_PROCESS

BROADCAST_PROCESS is called by a scheduler process if it wants to 'get rid of' (to say it bluntly) a process. The process to broadcast should be marked dirty. The coprocessor will try to send the task descriptor to the other coprocessors connected to the local token ring. If this succeeds, then the process is marked 'sent-broadcasted' for the local coprocessor, and 'received-broadcasted' for the other coprocessors. Because inter-coprocessor bus usage may delay the broadcasting, a timeout may be specified.

- Parameters for BROADCAST_PROCESS:

  PROCESS (no default) is the process to broadcast.

  TIMEOUT (default: 0 = infinite) is the number of clock ticks allowed before the actual broadcast starts.

- Results returned by BROADCAST_PROCESS:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  TIME_ELAPSED gives the number of clock ticks the scheduler process was placed in the waiting state waiting for the sending of the broadcast message. This is a 16 bit value, which blocks before overflow.

### 2.13.10 CLAIM_PROCESS

CLAIM_PROCESS is called by a scheduler process if it wants to 'get hold of' a process. The process to claim should be marked 'received-broadcasted' or 'sent-broadcasted' (the latter case is a 're-claim' !). The coprocessor will try to send a 'claiming process' message to the other coprocessors connected to the local token ring. If this succeeds, then the process is removed from the process queues in the other coprocessors, and the 'xx-broadcasted' flags are reset in the local process queue. Because inter-coprocessor bus usage may delay the claiming, a timeout may be specified. If more than one coprocessor tries to claim the same process, then the inter-coprocessor bus hardware will resolve the request on a (preferably) round-robin priority basis.

- Parameters for CLAIM_PROCESS:

  PROCESS (no default) is the process to claim.

  TIMEOUT (default: 0 = infinite) is the number of clock ticks allowed before the actual 'claiming process' message sending starts.

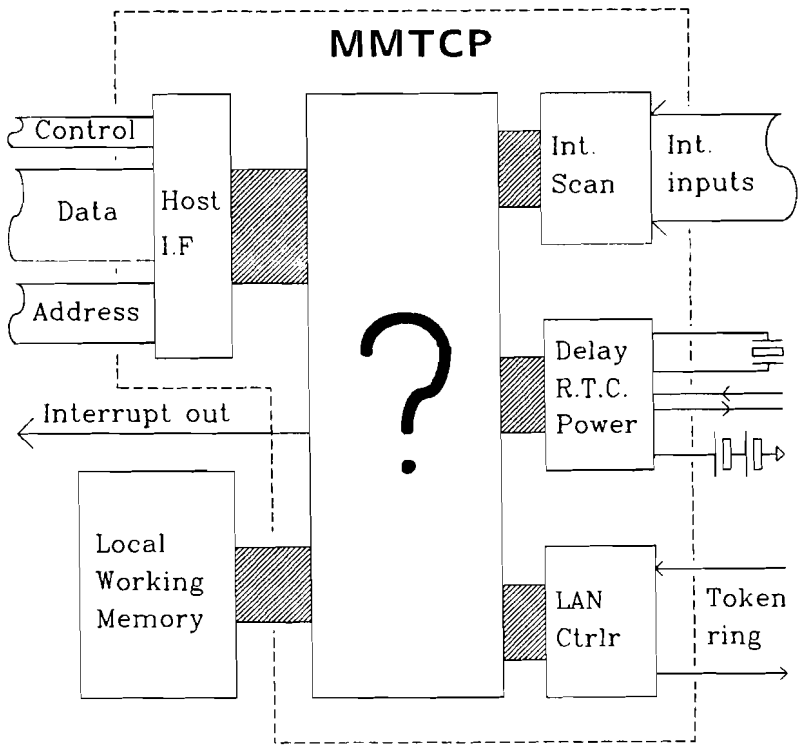
- Results returned by CLAIM_PROCESS:

  ERROR_CODE is a (16 bit) word indicating whether an error occurred, and, if so, which error it is.

  TIME_ELAPSED gives the number of clock ticks the scheduler process was placed in the waiting state waiting for the sending of the 'claiming process' message. This is a 16 bit value, which blocks before overflow.

## 3. Functional blocks

If we take a look at the MMTCP from the outside, we can see the following hardware interfaces:




In this chapter, I will try to identify the functions to be performed by the MMTCP and describe ways to implement them in hardware, filling in the big question mark in the middle as far as possible.

Processing time can be saved by executing functions in parallel. To do this, they should be build into separate hardware blocks, which are therefore appropriately called 'functional blocks'.

The functional blocks will be tied together with high speed on-chip data buses. Although these buses are not real functional blocks themselves, they are important enough to be described in the next subchapter.

## 3.1 On-Chip Buses

In this subchapter I will try to give some ideas for the on-chip buses present on the MMTCP chip. The chip will contain three buses:

1) An internal messaging bus for sending messages between the functional blocks.

2) An internal working memory access bus to enable the functional blocks to read and write in the (off-chip) working memory.

3) An internal host interface bus to connect the host interface logic to the host-addressable registers present in several functional blocks.

The first two buses are completely controlled by the functional blocks themselves. The internal host interface bus is nothing more than an on-chip version of the external host interface bus, and is controlled by the host processor. This last bus is therefore not very interesting, and will not be described here.

In the following discussion, the functional block initiating a transfer will be referred to as the bus master, the functional block which is addressed will be referred to as the bus slave.

The messaging and working memory interface buses serve two different purposes, and are therefore equipped with specialised hardware to optimise their use. Both buses, however, are to be used by different functional blocks, and therefore need some form of bus arbitration. For the internal messaging bus, there can be several bus masters connecting to different slaves. For the memory access bus, there is only a single slave - the working memory interface functional block. These two buses will be described separately in the two subchapters below.

### 3.1.1 Internal messaging bus

The internal messaging bus is used to exchange messages between the functional blocks present on the MMTCP chip. Almost all blocks are connected to this bus, most of them have the capability to take control of the bus to initiate a message transfer. The addressed slave registers can be viewed as simple input/output ports.

Most message transfers will be in the direction master to slave (the bus master does a bus 'write'). This makes it possible to use these write actions as handshake signals with the data being written as specification for the cause of the handshake. In some cases, the actual data is of no concern - the write action itself is sufficient as a signal. To do this, the slave register should be equipped with a 'written' flag, which is set when the register has been written by a master. This flag can be used inside the slave functional block to change the flow of a state machine or microprogrammed controller (or even to cause a microprogram 'interrupt'). In principle, read actions can also be used for handshake signals, but I don't think this will happen often.

This bus will be used for internal messaging only (there is no direct connection to the outside world). The speed of this bus is therefore not dependent on external hardware, and can be optimised for internal use (preferably top speed, of course - one transfer per clock cycle). The width of this bus is open for discussion, but

because most of the MMTCP hardware is centered around handling 16 bit quantities, the most logical choice seems to be a full 16 bits bus width. The number of slave registers connected to the bus determines the number of address bits needed, and will probably lie somewhere in the range of 32..64 registers (this is only a guess !). We should reckon with 5 to 7 address bits for the internal messaging bus. Extra bits can be used for easier block decoding of the addresses - 4 bits for the functional block and 3 bits for the specific register in this functional block, for instance.

The messaging bus should be equipped with some kind of 'bus locking' mechanism to be able to do protected 'test and set' operations on hardware semaphore registers. A straightforward implementation uses a flag emanating from the bus master during a bus access cycle indicating that it wishes to keep the bus until the next transfer from the same bus master.

The hardware for this internal bus is relatively simple, both on the master side and on the slave side of the bus. The master initiates a transfer by setting the wanted slave register address, the transfer direction and the bus lock request flag in a special register, and sets a 'bus request' flip-flop. The outputs of these bus request flip-flops are connected to some form of arbitration logic, to make sure that only a single master can access the bus at the same time. During the next clock cycle, the arbitration logic generates an 'acknowledge' pulse for the bus master that is granted access to the bus. This acknowledge pulse resets the request flip-flop, gates the address and flag register from the corresponding bus master on an internal control bus, and initiates the bus transfer to take place at the next clock pulse.

The arbitration logic should contain the hardware to lock the bus following a request from a bus master. This can be implemented as a simple flip-flop, clocked at the same edge as the actual data transfer. If one of these flip-flops is set, it indicates that the corresponding bus master has locked the bus for it's own use, and only the requests from this bus master will be honoured until after this master has executed a bus cycle with the 'lock request' flag reset.

### 3.1.2 Internal working memory access bus

The working memory access bus serves to connect the internal functional blocks with the external working memory via the working memory interface functional block to be described below.

Because the working memory is slower than the internal functional blocks, several MMTCP clock cycles are needed to complete an access cycle. This, together with the fact that the external memory interface has a multiplexed address/data bus makes it possible to multiplex the internal address and data buses (which saves a lot of interconnections with a data width of 16 bits and 21 address bits).

The interface from the functional blocks can be almost the same as for the internal messaging bus. The functional block requesting access places the wanted address with some control bits in a register, sets up the data to write for a write action, and requests the transfer by setting a request flip-flop.

An arbiter in the memory interface functional block decides which functional block may execute the next bus cycle, then reads the address and control register via the internal bus. This address is transferred to the external bus and latched

there (possibly in two halves), after which the data register is read in case of a write cycle (again, using the same multiplexed internal bus). In case this was a write cycle, the access is completed as far as the requesting functional block is concerned, and this block can immediately request another bus cycle. If the functional block requested a read access, the memory interface functional block reads the data from the memory, and writes the data read to a read data holding register in the requesting functional block, completing the transfer.

The actual interface to the external memory chips is described in the subchapter on the working memory interface functional block. Several additional functions can be build into the memory interface functional block, to be controlled by the internal memory access bus. These serve to make it easier for the other functional blocks to interface with the external memory:

1) **Table indexing**. Most of the working memory is addressed as a memory block number (16 bits) and an offset within this block (5 bits). For more information regrading this addressing method (and why it is used), see subchapter 3.3: 'Functional Blocks: Memory allocation and de-allocation'. This method works fine for that part of the memory that is used for the storage of dynamic variables, but not for the variable length tables.

   To read from such a table, one would like to simply add an index offset to the start of the table (the start of the tables are fixed after the INIT_SYSTEM call is executed by the host processor). Because many functional blocks use these tables, it would be wasteful (both in chip space and time needed) to have them all calculate the precise address for table access themselves, as this can be integrated into the memory access functional block.

   The idea is to store the start locations of the tables in a register array inside the memory interface functional block, together with the index adder needed to form the actual address. Now a functional block need only supply the wanted table number together with the wanted indexing offset. The longest table will be well under 64 kilowords, so the actual adder need not be 16 bits wide. The remaining address bits can simply be incremented for an overflow from this adder.

2) **Byte access**. The working memory is organised in 16 bit words, which poses no problems in case the accesses are done a word at a time. In some instances, bytes must be read or written because using words for some of the values simply uses to much memory. This is no problem for reading, as simply the whole word can be read, and the half that is not needed can be ignored.

   Writing poses some problems, because we only want to change that part of the word that contains the target byte, while the other half should remain intact. This necessitates using a read/modify/write ('RMW') cycle mechanism, which can be build into the memory interface functional block. Using such a mechanism makes it possible to optimise memory timing for a RMW cycle, instead of using a normal read cycle back to back with a normal write cycle.

   The byte shifting for reading and writing can also be done in the memory interface functional block, so that bytes are always written and read over

the lower half of the internal memory access bus, irrespective of their position in the memory word.

3) **Address incrementing.** A special control bit can be used to request addressing the logically 'next' word in the working memory. The incremented address should be written back to the address registers in the requesting functional block, using an extra data transfer cycle on the internal buses. The way an address is incremented depends upon the access cycle type:

   * For table access cycles, the table index offset should be incremented, which is a very simple operation.

   * For non-table access cycles, we run into some trouble. As long as the end of a memory block is not reached, we can simply increment the five bits block offset. But when the end of a memory block is reached, we will have to do an extra read cycle to obtain the block number of the next block in the chain, and reset the block offset to the first usable word in the memory block. This extra read cycle can be generated by the memory interface block itself, without using transfer cycles on the internal buses. The end of a chain will be indicated by an invalid block number being returned, the functional block requesting the access cycle should take care of this.

I propose the following control bits to be used for the working memory access bus:

**Table_Access** (T_A): this control line is TRUE when a functional block wants to use the table access mechanism described above.

**Lock_Bus** (L_B): this control line is TRUE when a functional block wants to 'lock' the working memory access bus for it's private use (preventing other functional blocks to access the working memory). The bus is un-locked when the same functional block has completed a memory access cycle with this bit reset.

**Inc_Address** (I_A): this control line is TRUE when a functional block requests the memory interface to increment the address that is used in the present cycle. The incremented address will be written back to the address registers within the functional block requesting the memory access cycle.

**Write_Cycle** (W_C): this control line is TRUE when a functional block wants to write into the working memory, FALSE when it wants to read from the working memory.

**Byte_Cycle** (B_C): this control line is TRUE when a functional block wants to read or write a single byte instead of a whole word.

**High_Byte** (H_B): this control line is only checked if Byte_Cycle is TRUE, and it indicates that the byte to read or write is located in the upper half of the addressed word. It is possible to share this control line with the Table_Access control line, as will be illustrated below.

Because the number of tables is lower than 9, two bits in the 'offset' bus can be used to encode the access type for table accessing. If we number the offset bits

from 0 to 4 (0 being the least significant bit), then the table number can be encoded in bits 0..2, bit 3 will become the 'Table_Byte_Cycle' bit and bit 4 (the most significant bit) will become the 'Table_High_Byte' bit.

Doing so makes it possible to encode a table access with 'High_byte' TRUE and 'Byte_Cycle' FALSE, saving the expense of a separate 'Table_Access' line.

The following table gives a list of the possible access cycles and how they are specified (the 'Lock_Bus' and 'Inc_Address' control bits have no influence on the type of the access cycle):

| W_C | B_C | H_B | <-offset-> | <--address--> | Access Cycle: |
|---|---|---|---|---|---|
| | | T_A | 4 3 2..0 | 15..? ?..0 | |
| 0 | 0 | 0 | [offset] | [block nr.] | normal word read |
| 1 | 0 | 0 | [offset] | [block nr.] | normal word write |
| 0 | 1 | 0 | [offset] | [block nr.] | normal low byte read |
| 1 | 1 | 0 | [offset] | [block nr.] | normal low byte write |
| 0 | 1 | 1 | [offset] | [block nr.] | normal high byte read |
| 1 | 1 | 1 | [offset] | [block nr.] | normal high byte write |
| 0 | 0 | 1 | ? 0 [TN] | ????? [TI] | table word read |
| 1 | 0 | 1 | ? 0 [TN] | ????? [TI] | table word write |
| 0 | 0 | 1 | 0 1 [TN] | ????? [TI] | table low byte read |
| 1 | 0 | 1 | 0 1 [TN] | ????? [TI] | table low byte write |
| 0 | 0 | 1 | 1 1 [TN] | ????? [TI] | table high byte read |
| 1 | 0 | 1 | 1 1 [TN] | ????? [TI] | table high byte write |

[TN] is the Table Number, [TI] is the table indexing offset.

Note that the data bus is time multiplexed with the 16 bits address bus, and that byte data is always transferred over bits 0..7 of this bus (the least significant bits).

### 3.1.3 Multiplexing the internal buses

In the previous paragraphs, both internal buses (the messaging bus and the memory access bus) have been depicted as being separate entities. This may, however, use up a lot of chip space, because both buses use approximately 25 bus lines. To save chip space in both the bus transceivers and the chip floorplan space needed for the bus traces, it might be an idea to use a single set of bus traces for both buses.

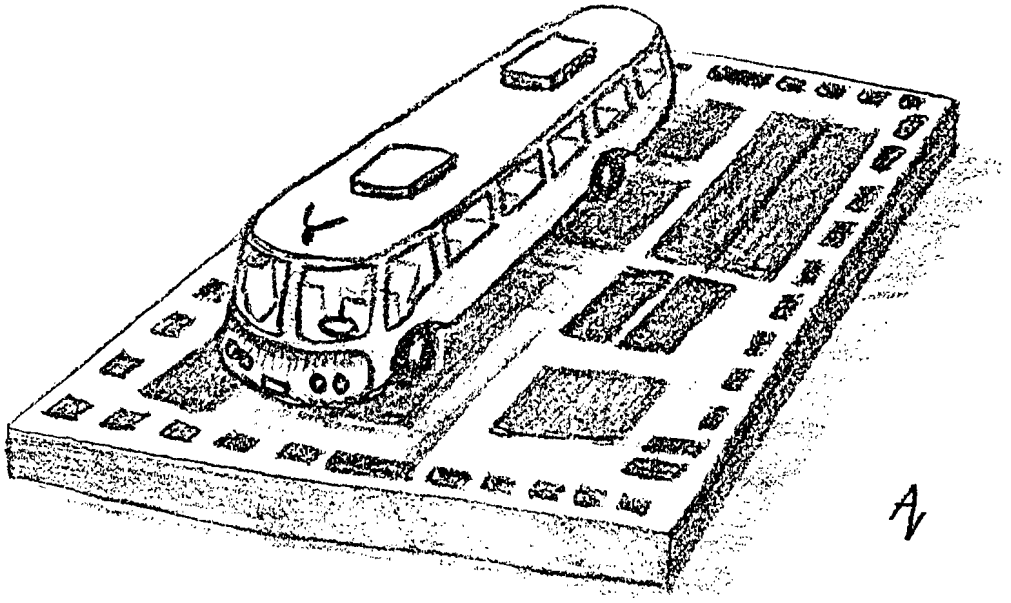
In principle, this should not pose too much problems, because the buses can be mapped easily onto eachother. Both buses have a 16 bit datapath (multiplexed with a 16 bit address for the memory access bus) and both buses have read/write and lock control bits. The memory interface bus has a 5 bits offset bus and three bits to control byte access, table access and address incrementing - these bits taken together may constitute an eight bits message register address for the messaging bus.

With some simple extensions to the bus arbiter for the messaging bus, it is possible to suppress message transfers while a transfer for the memory access bus is done (either an address or a data transfer). Even when the memory access bus is used at it's full data bandwidth, there will be clock cycles left to transfer messages over

the same hardware bus lines (a memory cycle may take well over 4 clock cycles, while only 2 or 3 are used for the internal transfers).

Note that the bus lock mechanisms should be separate for the messaging bus and the memory access bus. This should be done because the buses must be regarded as logically separate entities, in spite the fact that they share the same data path in hardware.

Also note that multiplexing the memory access bus with the messaging bus should be regarded as the stealing of message transfer cycles from the messaging bus. At this moment, it is impossible to decide whether this can be done without degrading the system performance too much - we will simply have to wait until we have enough data available to calculate a rough estimate of the bus bandwidth needed by the messaging bus. If the performance degrades too much, then we are forced to go back to separate data paths for both buses.



**3.1 - Functional Blocks: On-Chip Buses**

## 3.2 External Working Memory

The external working memory is implemented off-chip, as it's name suggests. Therefore we will need a memory interface to control this functional block. This interface will be the direct connection between the on-board busses and the off-board standard memory chip(s). Besides the basic memory interface functions, additional functions like table addressing and byte access cycles can be built into this functional block. These functions have been described in subchapter 3.1.2: 'Internal working memory access bus'.

To comply with the internal bus and register widths, the external bus width should be 16 bits. This may also be an absolute necessity to get a high enough bus bandwidth.

To be able to build a minimum component count system, it might be an idea to have some memory implemented on-board. A memory size of 2 KWords should be enough to build (very) small multitasking systems, like a simple terminal or a sensor/actuator controller. A memory like this will require extra chip space, so we have to weigh the advantages against the disadvantages before implementing it.

A second idea might be the implementation of a cache memory in the memory interface. The speed gain normally achieved with a cache memory will not be gained when used with the multitasking coprocessor. The reason for this is the following:

The coprocessor will try to optimise it's use of the external memory because several internal blocks have to make use of it. To prevent the external memory from becoming the performance 'bottleneck', all operational blocks will try to keep data read from the memory in internal registers. All data writes will be postponed until the time when it is known in advance that no additional changes have to be made in the data word. Doing so, an 'intelligent' cache memory is formed, which will work better than any other cache memory that can be implemented.

To be able to use low-cost dynamic memory chips, the memory interface should have the possibility to multiplex the desired address, and generate correctly timed RAS (Row Access Strobe) and CAS (Column Access Strobe) signals. Although this address multiplexing already cuts the necessary number of address lines in half, it will be necessary to multiplex these address lines with the data bus to save package pins. This can be done if the column address is latched externally at the right moment (the latch signal will have to be provided by the memory interface). The number of address lines to be multiplexed should be adjustable, because the different sized dynamic memory chips have different numbers of address pins:

| | | | |
|---|---|---|---|
| 16K x N: | RAS = A0..A6, | CAS = A7..A13 | (7 address pins) |
| 64K x N: | RAS = A0..A7, | CAS = A9..A15 | (8 address pins) |
| 256K x N: | RAS = A0..A8, | CAS = A9..A17 | (9 address pins) |
| 1M x N: | RAS = A0..A9, | CAS = A10..A19 | (10 address pins) |

If dynamic memory chips are used, then the remaining address lines can be brought out to be latched and decoded for bank selection of multiple memory banks. If the maximum memory size is set to 2048 KWords (65536 pages of 32 words), then the following number of address lines has to brought out to be latched:

| | | |
|---|---|---|
| 16K x N: | A14..A20 | (7 lines, 128 banks) |
| 64K x N: | A16..A20 | (5 lines, 32 banks) |
| 256K x N: | A18..A20 | (3 lines, 8 banks) |
| 1M x N: | A20 | (1 line, 2 banks) |

Dynamic memory chips have to be 'refreshed' to retain their data values. This is also a function to be handled by the memory interface. The time distance between refresh cycles and the number of bits in the refresh counter should be adjustable.

The interface to the dynamic memory chips will consist of the following signals:

16 address/data pins, RAS, CAS, write enable, latch enable.

It should be possible to connect static memory chips too. In it's simplest form, the whole address will be latched off the data input/output pins in a single cycle, providing a 64 KWord address space with 16 address bits.

The interface to the static memory chips will consist of the following signals:

16 address/data pins, read, write, latch enable.

This interface uses only 19 pins (the dynamic memory interface used 20 pins), which means that one pin can be used as an extra address line, providing 128 KWords of memory.

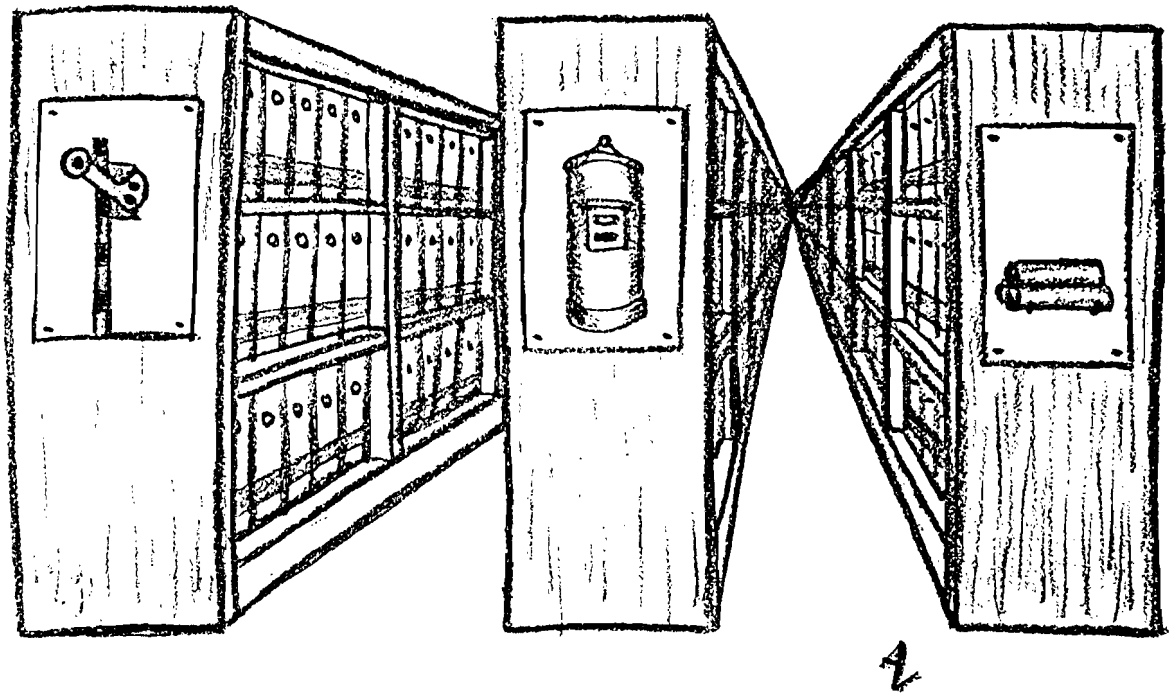
It is possible, of course, to latch the addresses for the static chips in two halves too (which must be done to be able to address to complete memory space). In this case, the RAS line can be inverted to act as the latch signal for the first address word, and the CAS line becomes the read strobe (memory refreshing can be turned off).

Note that some manufacturers produce 'byte-wide' dynamic RAM chips. These chips have no multiplexed address bus, but have to be refreshed to retain their memory contents. Together with the possibility to connect static RAM chips with address multiplexing, we can only reach one conclusion: memory refreshing and address multiplexing should be regarded completely separate options, and should not be coupled.

To provide some error protection, on board parity checking can be provided. Because reading and writing is always done on a word basis, only one parity bit has to be provided, using one bidirectional pin. If external error detection and/or correction circuits are used, then this pin can be used as an input to signal (uncorrectable) errors. In case an external error correction circuit is used, memory 'scrubbing' will be one of the functions to be provided by the memory interface (memory scrubbing is the periodic readout of all the available memory to correct all correctable errors before they turn into uncorrectable errors).

Upon detecting an error, then the memory interface will send a message containing the error location to one of the other functional blocks (probably the memory allocation and de-allocation functional block). This functional block will then decide what actions are to be taken. The functional block requesting the memory transfer cycle has to be signalled too (maybe put into some kind of hardware 'sleep' state).

All these options can be set when the system is initialised, before the memory is actually used (TYPE_OF_LOCAL_MEMORY parameter in the INIT_SYSTEM function call). This means that all options can be made software selectable, and no strapping pins are necessary. Following startup, the memory has to be initialised and checked by one of the functional blocks (again, the most logical choice would be the memory allocation and de-allocation block, but this is not an absolute necessity).



**3.2 - Functional Blocks: External Working Memory**

## 3.3 Memory allocation and de-allocation

The memory use is by no means a static one. Semaphores are created and deleted at will by the running processes. Blocks to store overflow messages for an 'infinite' mailbox have to be created when needed, and deleted after their contents have been copied into the main mailbox buffer block.

This allocation and de-allocation of memory blocks can be done by a central memory management functional block. Having only one functional block to handle this will make a more compact system possible (no duplication of functions), and also makes it possible to arbitrate between the requests from the other functional blocks. The memory management block will also be the place where the START_OF_LOCAL_MEMORY and END_OF_LOCAL_MEMORY parameters from the INIT_SYSTEM function call are written to.

The memory management functional block can also be used during the initialisation of the external memory, following the specification of the memory interface options. The block will then have to initialise the memory to a known state, and can also check for errors. During initialisation, the data structure needed to do the allocation and de-allocation should be build, and blocks containing errors can be excluded from this structure. Space should be reserved for the fixed memory structures:

* the interrupt semaphores table
* the hashing table used by the process address search functional block
* the hashing table for the channel semaphores
* buffers for the stream input/output handlers (if special buffers are used for this purpose)
* the accessible ring numbers table for a 'bridge' multitasking coprocessor

If memory errors are detected by the memory interface functional block, then the memory management functional block will receive a signal and the address of the error. If the error was detected during memory 'scrubbing', then the memory manager may consider removing the block from the available blocks list, so that the block will not be used again (if it was not already in use). In any case, a message will have to be written to the system errors mailbox (described in subchapter 2.6: 'Functional description: Mailboxes'). A special user-defined process running on the host processor can read these error messages and decide what to do with them.

The algorithm used for the dynamic memory management can be fairly simple (and it should be, because it has to be done in hardware). The problem is, that the block lengths that are needed by the functional blocks vary a lot (several words for a semaphore to kilobytes for a long pipe), so that some kind of segmented memory would be helpful.

The problem with segmenting is that it normally leads to memory 'trashing' (leaving small unused 'holes'), which have to be closed by a process which is called 'garbage collecting'. When this is done, all the segments are moved upward in memory until all the small holes are removed, and collected together in a big empty space at the end of the memory. Unfortunately, this moving of segments will pose big problems in our system, because an entity is identified by its location in the memory of the multitasking coprocessor where it resides (this will make it much easier to address an entity, and automatically leads to protection from multiple

used addresses). This drawback could be overcome by using translation tables, but this will use up extra memory and (precious) processing time.

A paged memory system will do much better, especially if we organise it in such a way that it does not use (space consuming) tables. I propose a system of linked chains of memory blocks, which uses relatively little space, provides certain protections, is easy to handle in software and can be supported by dedicated hardware.

As already indicated in the subchapter describing the internal memory access bus, the memory is divided into blocks of 32 words (64 bytes) each. The first word in each block can be used as a pointer to another block (to chain blocks together). If the first word is 0, this automatically means 'end of chain' (Pascal 'NIL' pointer). Use one byte in the block (preferably the third byte) to indicate the current 'use' of that block. If a block is free or contains a memory error, this should also be indicated here.

Each of the entity types has it's own 'use' code. If an entity needs more than one memory block, then the first block in the chain should have a slightly different 'use' code from the rest of the blocks (as already stated, the last block will have the link pointer set 0). Having special 'use' codes for the first block of an entity's chain helps in the address checking, as each entity is addressed by it's first block.

When the system is initialised, all the memory blocks are checked, and the blocks that have no errors in them are linked together in a huge chain of free blocks. When an entity is created, the number of blocks needed are simply taken from the head of the chain, the 'use' bytes are updated and the 'link' word of the last block is set 0. When an entity is deleted, the blocks in it's chain are given the 'use' code 'free' and the chain is linked to the end of the free blocks chain, making these blocks available for re-use.

Using such a memory structure seems elaborate and time consuming, but it can easily be prevented from becoming so. The first block of each entity should contain the most important data for that entity, so that this data can be addressed directly without searching in the chain. All the forward and backward pointers for the chains an entity is placed in should be in the first block, so that searching these chains can be done as fast as possible. Also, the main status word(s) for an entity should be located here. Less important data can be stored in the second block of an entity's chain (needs only one intermediate word fetch to address this block). The task descriptor cache described in the next subchapter provides an alternative way to prevent most of the searching in the linked lists for a task descriptor.

Entities which have buffers (mailboxes and pipes) do not suffer too much performance degradation from this memory system. These cyclic buffers are always written and read in the forward direction (following the chain of blocks), which can easily be done by having a 'current block' pointer together with an 'offset pointer' within the current block. The current block pointer only needs to be updated when the offset pointer has reached the end of it's block. When the end of the chain is reached, the current block pointer can be reset to the first block in the entity's chain that is used as a buffer. Following a chain of blocks while reading or writing in these buffers can be implemented in the local working memory interface functional block, as described in the subchapter on the internal memory interface access bus.

### 3.3 - Functional Blocks: Memory (De-) Allocation.

Hardware can be used to speed the allocation of memory (de-allocation can always be done in the 'background', as this is not very time critical). An on-chip FIFO-type memory can be used to store pointers to the first few blocks in the free blocks chain (at least enough empty blocks to satisfy most requests at once - all requests if possible).

If a functional block requests the allocation of memory, it can send the number of blocks requested to the memory (de-) allocation handler, together with the intended 'use' code for the memory. The allocation handler then indexes in the FIFO to retrieve the pointer to the last block in the chain to be allocated, and writes the 'NIL' link pointer in that block. The first block is written the 'use' code (with the 'first block' indication set), and the pointer to the first block is written back to the requesting functional block. After this is done, the newly allocated blocks are removed from the FIFO and the FIFO is filled up again by reading forward in the free blocks chain.

This last action can be done in the background, as this is not a very high priority task. It has a higher priority, though, than the updating of the 'use' bytes in the newly allocated blocks (which was not done yet).

De-allocation of memory can be done by writing the pointer to the first block of a chain to be deallocated to the memory management functional block. This pointer is then written to the chain pointer in the originally last block of the free blocks chain. Changing the 'use' bytes to 'free' is a low priority task that can be done in the background.

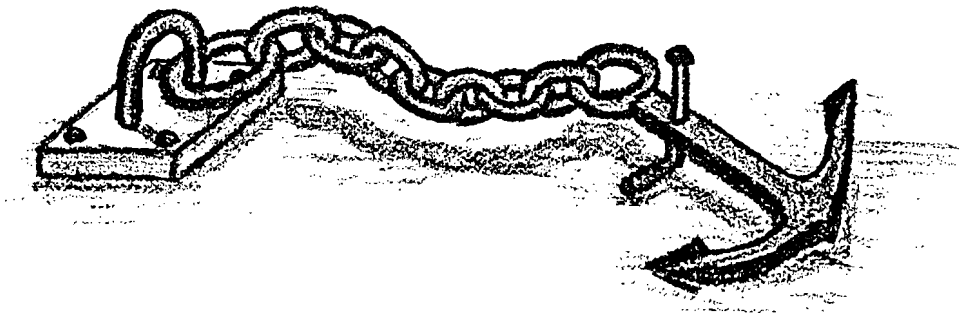
A problem with multitasking operating systems is that tasks are sometimes not informed about the deletion of an entity. Such a task may try to use the entity identification later, while this identification is in use by another entity, which may be used for a totally different purpose. This is not a problem if the entity type differs from the original entity type, as this error will be detected by most of the operating systems (in our case, the type must match the 'use' byte). But if the type of the old entity is the same as that of the new one, serious trouble may result, even system crashes.

There is no way to make certain that this will never happen. We can only try to hold off this problem as long as possible by re-using pointers only when there are no 'fresh' pointers available any more (hoping that tasks that want to make use of outdated pointers will have been deleted by that time). The first step towards this goal is by using the chain of free blocks as a time delay 'device'. Linking deleted chains to the end of the free blocks chain will make certain that blocks fetched from the head of this chain are deleted as far back in time as possible.

More elaborate techniques are possible too, like trying to allocate new entities with as first block a memory block that has never been the first block of an entity before. But this will fail after a while too, because somewhere in time, all blocks will have been the first block of an entity once, and then even this system breaks down.

Eventually, any system will break down when the memory is constantly used to a high percentage of the available memory (the time delay in the free blocks chain will then be short). We will simply have to live with the possibility that this might happen.

The only way to avoid it is to add extra software to the shell surrounding the kernel, so that user processes can only use entities from which the operating system knows that they are valid. The operating system shell will need some form of entity administration to do this, which may make the system a bit slower.



**3.3 - Functional Blocks: Memory (De-) Allocation.**

## 3.4 Process Searching and Task Cache

This functional block serves two purposes:

1) Provide hardware to speed up searching for processes by logical process number (to be able to decide whether or not a process descriptor is located in the local working memory, and, if so, where it is located).

2) Provide a cache memory for the task descriptors of the most 'active' tasks. This can save a large number of memory accesses and also provides hardware 'assistance' for the scheduler list search functions.

Both functions benefit from this combination, and will be described separately in the next two subchapters.

### 3.4.1 Process address searching

In a multiprocessor multitasking system, the possibility exists that processes are moved between host processors (to get a more even distribution of the workload in the system).

The problem here is, that a process gets an identification number when it is created, and that this number has to remain the same throughout the lifespan of this process. For 'static' entities like semaphores, the identification number assignment is easy - the number of the coprocessor where the entity is stored and the address in the working memory are concatenated into the entity's identification number.

For a 'dynamic' entity like a process block, this assignment poses large problems. A process might be created in coprocessor 'A', moved to coprocessor 'B', and later moved back to 'A'. If the memory space originally occupied in the memory of coprocessor 'A' is in use when the process has to be moved back, then the process will have to be stored at another location, breaking the direct 'link' between the identification number and the address.

Another problem is that there may never be several processes with exactly the same identification number in a single multiprocessor system.

To cope with these problems, I propose the following algorithm to be used when assigning an identification number to a new process:

1: Generate a number by concatenating the identification number of the creating coprocessor with a value stored in a counter in one of the on-board registers, then increment the counter (wrap around).

2: Check if there is a process in the local memory with the same identification number. If there is such a process, then repeat again from step 1.

3: Send a data package to all the coprocessors in the local system, asking them to search for a process with the newly generated identification number, then wait a short while to give them time to search for the process and respond. If one of the other coprocessors has found a process with the same identification number, then repeat again starting from step 1.

4: If we end up here, then we can be sure that the generated identification number is not used for another process in the system. The process administration block can be stored in the local working memory, and the process can be assigned the new identification number.

NOTE: in systems where there is a single MMTCP creating (most of-) the tasks, the possibility exists that all the available task identification numbers will be in use at once. Although the chances are slim that this actually happens, the algorithm should be able to detect this problem and return the error code 'no process ID available'.

Using this algorithm, there is no correspondence between the process identification number and the address in memory where it is stored. So, if a message is addressed to a process (or one of the pipes the process is reading from - the pipes are dynamic entities too !), then the coprocessors have to search for the process. This searching has to be done very fast, to be able to respond to the message in time.

To prevent this searching from becoming a performance bottleneck, it has to be avoided as much as possible. A trick which can be used with the pipes, for instance, can be to search only if a process tries to claim the pipe. A message can be sent back containing the address of the coprocessor where the pipe actually resides - all data sent to the pipe can then be sent directly to the correct coprocessor.

This block contains two hardware functions, and a combination of hardware and (microcoded) 'software' to execute the search in the working memory if the internal hardware has not found the process.

The first hardware function is a cache memory which directly translates the process identification number into a memory address. In this case a cache memory will be a very powerful tool to speed the searching, because it will always hold the most 'active' processes. If the cache memory 'hits', then no searching has to be done at all.

The second hardware block is a hashing 'machine'. Each process identification number is hashed into a shorter number (with a limited range - 0..63 for instance), and each of these shorter numbers correspond to a bit in an internal 'hash table'. If there is at least one process stored in the local memory where the identification number hashes into value 'x', then bit 'x' in the internal hashing table will be set. So, if bit 'x' in the hashing table is not set, then there is no need to search for a process with an identification number that hashes into value 'x'. This will prevent a lot of unnecessary searching.

The searching itself can be speeded up by using the number produced by the hashing machine as an index into a table in the local working memory. Each table entry points to the first process in the local memory that has an identification number that hashed into the corresponding hash code. Processes with the same hash 'value' are stored in memory as a linked list. The end of such a linked list is identified by storing an invalid pointer in the task descriptor block at the tail of the list. This same invalid pointer can be used in the table to identify those table entries that have no corresponding processes (these entries have the bit in the internal table reset).

The communication between this functional block and the other functional blocks can be performed using the internal messaging bus, as introduced before. An address search is initiated by writing the full identification number to an internal register in the address search block, the result is reported back as an address directly pointing in the internal memory (an invalid address can be used to denote 'process not found').

### 3.4.2 Task descriptor cache

In principle, the cache memory described above need only contain the logical process number and the memory location where the corresponding task descriptor is stored (a 16 bit binary number).

Unfortunately, a task descriptor does not fit into a single memory block (of 64 bytes), and therefore has to be stored in memory as a linked list of memory blocks (probably 3 to 5 blocks will have to be used). When accessing the last memory block of a task descriptor, a functional block will always have to traverse this linked list, which takes memory access cycles and time. This can be avoided by expanding the task descriptor cache entries with direct pointers to the memory blocks allocated to the task descriptors.

The task descriptor cache should now be equipped with 'match' circuitry for the memory pointers for the task descriptors. This will make it possible to search the cache in two ways, by logical process number and by memory location. The first search function will be used if the address is unknown (process address searching - see previous subchapter), the second search function can be used if the memory address is known (all task pointers in the working memory will point to the actual memory location of a task descriptor's first block).

A functional block can now obtain the location of a memory block in the task descriptor's chain of memory blocks by writing the address of the first block to the match circuitry in the cache memory. If the cache memory 'hits', then no memory accesses need be done at all, and the requested memory pointer can be read immediately. If the cache memory fails, then one 'line' in the cache memory will have to be loaded with the data of the requested task descriptor (by traversing the chain of blocks once), during which the requested pointer can be given to the functional block that asked for it.

The functions used by the host for virtual memory handling and process transfer can benefit from this cache memory too. The functions that search the 'scheduler list' actually work in two phases:

1) Following the function call, the linked list of all processes is searched in the requested direction for a process that meets the search criteria at that time.

2) If this search fails (and RETURN_IF_NOT_FOUND is set FALSE), then all process status changes are monitored until there is a process that does meet the search criteria.

This last part of the process 'search' may pose some problems. Without hardware assistance, all the important parameters of a task descriptor should be read and checked against the search criteria when one of them has been changed. This

would mean a great number of memory accesses with only a small possibility of success.

An 'active' process is a process that is undergoing state changes at a relatively rapid pace (at least, that is one way to describe it). Such an 'active' process will be present in the task descriptor cache memory most of the time (an 'inactive' process will be loaded there upon becoming active), so it might be an idea to keep the parameters that need checking in the task descriptor cache memory, and do the search criteria checking on-chip.

Keeping these parameters on-chip in the task descriptor cache also makes it possible to do the actual parameter changes in the cache memory. This will prevent a lot of unnecessary memory access cycles, because a task descriptor is loaded into the cache memory when a task has become active (reading the task descriptor once).

A task is removed from the cache when it has become so 'quiet' that other tasks can be considered 'more active' and need to be loaded in the cache memory (replacing the 'inactive' tasks). This way, we get a kind of 'working set' of active tasks in the cache memory. When a task is removed from the task descriptor cache, parameters that have been changed must be written back to the corresponding memory locations !

When a parameter of a task descriptor must be changed in the cache, the complete task descriptor is read (a 'line' in the cache memory), the new parameter value is inserted, and the result is written back. Just before writing the values back, they can be checked (in parallel !) against the search criteria set by the scheduler list search functions.

To do this, these search criteria should be kept in on-chip registers located near the task descriptor cache. This poses the problem that there cannot be an unlimited number of searches going on at the same time, so we will have to place a restriction here. In my opinion, two complete sets of search criteria should be enough. One set can be used by memory management tasks searching for processes that can be swapped out of the host memory, while the other set can be used to search for processes that can be swapped in.

There are two ways to handle the error that a new search is requested, while the maximum allowed number is already being done:

1) Disallow the new request (regard the request as a serious system error).

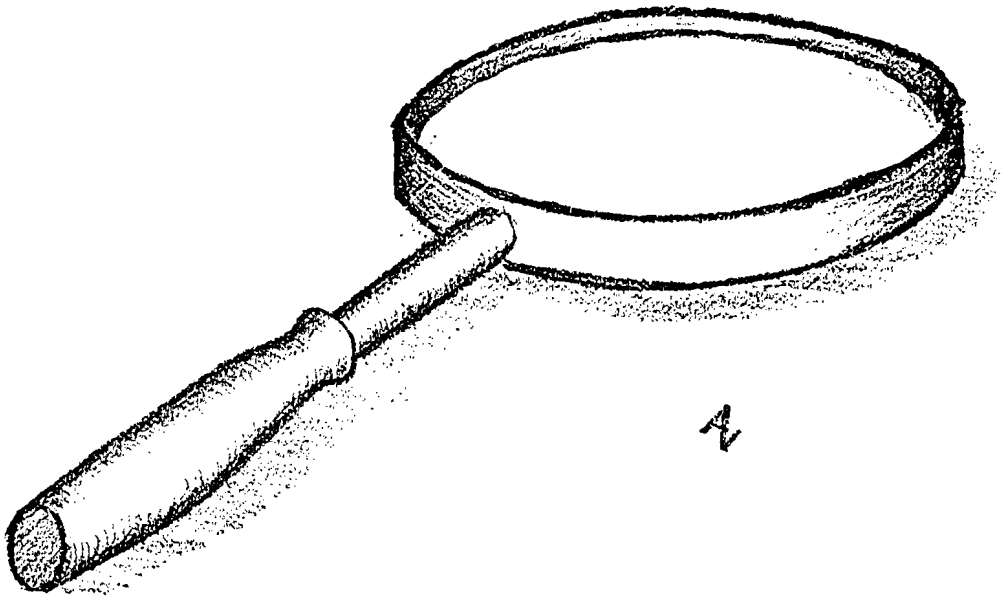
2) Stop one of the old searches, using the principle that a new search should take precedence over one that has not produced results in the past. In this case, a search initially specified in the same direction as the new one should be broken off first. If the searches currently done have the same initial direction, then the oldest one should be stopped.

Because of the large amount of parameters that must be checked for the scheduler list search functions, some tradeoffs may be necessary in the actual implementation.

Parameters like the user-defined size and state can only be changed by special function calls, so it is possible to have these calls check these parameters against

the given limits and set some TRUE/FALSE flags in the cache memory (saving a lot of space). Unfortunately, this makes it necessary to store a set of flags for each of the possible search criteria sets that can be used. It will also be necessary to recalculate these flags for all the entries in the cache when a new set of search criteria is loaded in the on-chip registers. All in all, using flags instead of the actual values saves space in the cache RAM lines, but takes more processing time and memory access cycles, and also makes the associated algorithms more difficult to implement.

A problem might be that the memory manager functions searching for tasks that can be swapped out generally are specifying search criteria that match 'quiet' tasks. This means that these tasks have a high probability of not being loaded in the task descriptor cache memory. A similar problem exists with the MIN_WAITING_TIME and MAX_WAITING_TIME parameters. How these can be solved is unknown at this time.



## 3.5 Host processor interface

The host processor interface is designed around the interface register set. These registers can be addressed directly by the host processor, and some of them will have hardware added to perform special functions.

The interface hardware should be able to connect to different types of host processors. The bus width should be selectable between 8 or 16 bits (strapping option). The Intel family processors have separate read and write strobes (active low). The Motorola family processors have a read/write select line and a single data strobe (active high). A strapping option will have to be used to select between both processor families. To be able to work with a multiplexed address/data bus, address latches should be provided for the host interface register select address lines. The chip select and byte enable lines should be latched too. The following tables give the different options:

| Bus width: | 8 bits | 16 bits |
|---|---|---|
| b.w.pin 1: | address A0 | Lo_Byte_Enable/ (D0..D7) |
| b.w.pin 2: | (?) | Hi_Byte_Enable/ (D8..D15) |

| Processor: | Intel | Motorola |
|---|---|---|
| p.pin 1: | RD/ | Data_Strobe |
| p.pin 2: | WR/ | RD-WR/ |

The total host interface consists of these four pins, 16 data lines, a number of address lines (5 or 6, starting with address A1 !), a chip select line (CS/), an address latch enable line (ALE) and an interrupt output line (DMA control lines are described in subchapter 3.13: 'Stream/Bridge data handling'). The active polarity of the interrupt output line can be programmed during initialisation of the chip. If address latching is not needed, then the ALE line can be tied high. For an 8 bit bus, the byte-to-word conversion can be done off-chip (connect the two bus halves together) or on-chip with a bus multiplexer. The first solution increases the host bus capacitance, the second uses some extra chip space (but leaves 8 pins free ! - these can be used as extra interrupt inputs).

Several of the registers generate 'attention' signals to other functional blocks when written or read. These attention signals can be seen as internal interrupts.

The parameter registers will have 'defaulting' hardware added to them. This defaulting hardware consists of a default - shadow - register for each of the parameter registers and a flipflop to remember whether the register has been written into or not. When a command is written into the command register, all these flipflops are set into the state 'not written'. Each write into a parameter register will set the corresponding flipflop into the state 'written'. If a functional block reads a parameter register, then the state of the corresponding flipflop determines whether the real parameter register or the default register will be placed on the internal buses.

Most of the default values are fixed, so that the corresponding default registers can actually be ROM. Only few of the default values can be specified during system initialisation, so these registers have to be RAM.

By making the 'connections' between the parameter registers and the default registers depending on the contents of the command register, the parameter

registers can be used as a linear array of registers (which is easier for the host processor as well as the internal functional blocks - it also eases the layout). This can be done by using a PLA to select the default register, with as inputs the contents of the command register and the (internally generated) parameter register address. With a PLA, it becomes possible to use a single ROM default constant for several of the interface registers.

To speed processing, 'double buffering' of the result registers may be needed. This makes it possible to have the results at hand for the process at the head of the ready to run list, in case the running process calls a function that will remove it from the running state. A task switch can then be done very quickly. Of course this can only be done if the coprocessor has enough time to 'prefetch' these result registers.

Most of the host interface registers are connected to a functional block that interprets and executes the host processor commands. If a functional block is used that does the actual host task switching, then the environment pointer registers, the special status word register and possibly the result registers can be connected to this block. The register used for the TRIG_INT_SEMAPHORE command can be connected to the interrupt handling block to be described later, because giving this command can be seen as the occurrence of an external interrupt.

### 3.5.1 Host interface protocol handlers

The host interface protocol can be split into three phases:

1: Receiving and interpreting the given command.
2: Executing the task switch.
3: Reporting the results of the command execution.

It might be an idea to have three separate functional blocks executing these phases. This is possible because reporting the results of a command does not necessarily follow the first two phases directly (the calling task may be put asleep by the given command). Task switching does not have to be initiated by giving a command, so it might be easier to have a separate block doing this (like a hardware 'subroutine'). Reporting the results can be done concurrently with the other tasks, for instance when another task is placed on the head of the ready to run list.

Restarting a task can be initiated by several sources. In all these cases, the same 'program' flow is followed, so it would be an idea to have a separate functional block to do the restarting of tasks (this block will be described later as 'task restarter'). The 'trigger' to restart a task can come from the following functional blocks:

1: Host command interpreter.
2: Interrupt handler.
3: Real time clock.
4: Delays generator.
5: LAN input controller.
6: LAN output controller.

I will now describe in more detail the host interface protocol functional blocks.

### 3.5.2 Host command interpreter

This functional block waits for the host to write a command (is the receiver of the 'command register written' attention signal), interprets the command and makes the decisions regarding 'what to do with it'.

If a command is given that immediately returns a result, then the result is given directly to the result register handler. Otherwise the results will have to be written to a result holding area in the task descriptor block in the working memory (to be fetched later by the result register handler). In this case the results can be changed by the task restarter, before they are fetched by the result register handler.

If a command triggers the restarting of another process, then the command interpreter signals this to the task restarter. All commands that send units, messages etcetera to other processes fall in this group. If they do no restart a process in the local coprocessor with a higher priority, then the calling task can immediately continue.

If a command makes it necessary to send a message to another coprocessor, then the command interpreter should signal this to the LAN output controller. To make concurrent processing and sending of messages possible, the message to send is placed in the working memory, and the address of this message block is written to the LAN output controller. Within the LAN output controller, a register structure can be used to store addresses of multiple messages to be sent (so that the LAN output controller can choose which message to send - based upon some kind of priority algorithm). If an action is to be taken upon the successful transmission of the message block, then this action will have to be coded inside this block (the LAN output controller will have to undertake this action).

The command interpreter will be one of the more difficult functional blocks to build, because it has so much tasks to perform (including range checking for parameters). It may be necessary to implement it as a microcoded machine.

### 3.5.3 Host task switcher

The host task switcher is one of the more 'simpler' functional blocks. In principle it has to do not much more than:

1: Wait for the host processor to write into the 'store environment pointer' interface register. This waiting can be done by using an internal trigger signal generated by the host interface registers functional block. The data will then be available in the mentioned interface register.
2: Store the given pointer in the process administration block for the running process.
3: Wait until a new environment pointer is available to be given to the host processor.
4: Store this pointer in the 'new environment pointer' interface register.
5: Signal the host processor to read the new environment pointer.
6: Wait until the host has read the new pointer.

But this block can do more. It can, for instance, monitor the internal register that holds the address of the administration block for the running task, and request a forced task switch if this register is changed (maybe after inserting a short time delay, to prevent too fast task switching). This means that the 'force task switching' interrupt output pin will be connected to this functional block. This block can work in close cooperation with the host result register handler functional block, to be described next.

### 3.5.4 Host result register handler

The host result register handler is responsible for the loading of the result registers before a task is restarted. To make quicker task switches possible, this block may try to predict the next task to be restarted and load the results belonging to this task in a set of 'shadow' result registers. In most cases, the best 'bet' will be the task that is waiting at the head of the ready to run list.

This block can also be used to monitor the head of the ready to run list, and automatically request a task switch (using the task switcher), when the process placed there has a higher priority than the currently running process. This will make it unnecessary to have separate 'attention' lines running into this functional block to request the starting of a new process.

For some of the simpler host commands (those which directly return a result), the result register handler immediately receives the results from the host command interpreter (using the internal messaging bus), and places them in the correct result registers.

The complexity of this block is unclear at the moment. It may be possible to build it as a relatively complex state machine, but it can also be implemented as a rather simple microprogrammed system. Most of the work it has to do involves the transferring of data, and the only operation it has to do is a simple compare for equality between two numbers (this last operation can even be done completely hardwired).

### 3.6 Task restarter

The task restarter is responsible for all actions that may release a task from a waiting list. Amongst others, these actions are:

- Sending of units to a semaphore.
- Signalling a channel.
- Sending messages to a mailbox.
- Reading messages from a mailbox (fixed size mailboxes).
- Making an exit from a region.
- Sending data to a pipe.
- Releasing a pipe.
- Delay or timeout ending.
- Real Time Clock match.

The task restarter will have to differentiate between two possibilities. The first is when a local task has to be restarted (no problem). The second is when a task must be restarted which resides in another coprocessor (but was waiting here for an event). In the latter case, a message will have to be sent to the other coprocessor (either via the LAN or via a port back to the host processor if it is a packet intended for another multitasking coprocessor ring network and the local host has 'bridge' capabilities).

If the packet is to be sent using the local coprocessor LAN ring, then the on-chip LAN controllers must be given instructions to send the message, wait for an acknowledgement, and destroy the memory block that was used to hold information regarding the non-local waiting process - the memory block itself can be the message sent back.

If the packet is to be sent via the host's main LAN, then the package must be written to the host memory using a DMA channel, and the host processor must be informed that this has been done (which also acts as a command to send the packet).

The task restarter will have some really complicated algorithms to perform, and therefore may have to be build as a microprogrammed machine. To save chip space, it might be an idea to build this microprogrammed machine as a pipelined machine with interleaved program execution. This will make lots of duplications unnecessary (only one ALU, only one PC incrementer, only...). The other task in the pipeline can then be the host command interpreter, for instance.

**3.6 - Functional Blocks: Task Restarter.**

## 3.7 Interrupt scanner

The interrupt scanner is the hardware interface for the real world to trigger external events. It consists of hardware to scan several interrupt lines, convert these lines into an internal line number, and trigger the sending of a unit to a semaphore. The actual sending of the unit to the semaphore (and the decision making as to which semaphore should be used) is all done in the interrupt handler functional block, to be described next.

To expand the number of input lines to be scanned, external multiplexers can be used, to be controlled by several output lines. To control these multiplexers, some of the direct interrupt input lines must be sacrificed. The following combinations can give an idea of what I mean:

Fully parallel:
12 interrupt inputs, no scanning done (default after chip reset).
Quadruple multiplexing:
4 groups of 10 interrupt inputs, scanned by 2 scan select lines (40 interrupt inputs in total).
Hextuple multiplexing:
16 groups of 8 interrupt inputs, scanned by 4 scan select lines (128 interrupt inputs in total).

Interrupt lines which are not used can be left unconnected. If none of the lines in a scanned group is connected, then this group does not have to be scanned (this shortens the cycle time for the scanner). It is the responsibility of the running processes not to enable an unconnected interrupt input.

If an 8 bit host processor interface bus is to be used, then the remaining 8 data lines can be used as extra interrupt inputs (assuming the 16 bits to 8 bits bus multiplexer is build on-chip). If these lines are used as extra scan input lines, then the maximum number of interrupt inputs can be increased to 16 * 16 = 256 interrupt inputs.

The configuration for the input lines and the external multiplexers can be given when the system is initialised. The INTERRUPT_CONFIGURATION parameter for INIT_SYSTEM is dedicated to this purpose (this parameter will be written to a register in this functional block).

To prevent unnecessary work to be done by the interrupt handler functional block, the masking of interrupt lines done with the ENABLE_INTERRUPT and DISABLE_INTERRUPT function calls can be done in hardware by this block. This block will also be used to select the active polarity of the interrupt lines and select between edge or level triggering for an interrupt. Both options are given as parameters for the ENABLE_INTERRUPT function call. This can be done because INIT_SYSTEM disables all interrupt inputs.

### 3.8 Interrupt handler

This functional block receives commands to send units to semaphores from the interrupt scanner or from the host processor using the TRIG_INT_SEMAPHORE function call.

Both commands are given in the same format - an index (scan line) number into a table of interrupt semaphores.

The translation of a scan line number to the corresponding interrupt semaphores will have to be done using a table in the working memory. Also, a table will be needed to hold the maximum number of units allowed in the normal interrupt semaphore before sending the unit to the alternate interrupt semaphore (the 'units ceiling' for this normal semaphore). The table length will be adjusted to the actual number of semaphores specified with INTERRUPT_CONFIGURATION.

The actions taken by the interrupt handler on receiving an interrupt line number can roughly be sketched as follows:

1: Read from the translation table the identification numbers for the normal and alternate semaphores together with the 'units ceiling' setting for the normal interrupt semaphore.

2: If none of the semaphores is specified, then send a message to the system errors mailbox (using the task restarter and the internal messaging bus).

3: If no alternate semaphore is specified, then send a single unit to the specified semaphore, using the task restarter. If this leads to an error, then send an error message to the system errors mailbox.

4: If only an alternate interrupt semaphore is specified, then send a single unit to this semaphore, using the task restarter. If this leads to an error, then send a message to the system errors mailbox.

5: If both semaphores are specified, then first try to send a single unit to the standard interrupt semaphore, with the 'units ceiling' value as the MAX_UNITS parameter, using the task restarter. If this leads to error, then try sending a single unit to the specified alternate interrupt semaphore (also by using the task restarter). If this last action leads to an error, then send an error message to the system errors mailbox.

This functional block is of intermediate complexity, probably too complex to be realised as a state machine. A small microprogrammed system may be needed.

## 3.9 Real time clock and Power Switch

The real time clock functional block consists of a crystal controlled oscillator (running at a lower speed than the system clock to preserve power in the power down mode), real time clock counters and a comparator to compare the counters to a preset time and date. Some simple hardware should be added to enable the automatic power-on of the system on a real time clock match. An attention message is sent to the task restarter, to signal that a scheduled task can be restarted.

The real time clock counters can be set and read from the host processor function handler functional block (only the local real time clock can be read and/or set). The time/date compare registers can be set from different sources:

* The host command interpreter, when a task has has been placed in the real time clock waiting queue with a match time more close to the current time than a task that was already waiting, or when the currently waiting task is removed from the real time clock waiting list (with UNLINK).

* The task restarter, when a real time clock match has occurred and the match time for the next task in the real time clock waiting list has to be placed in the compare registers.

Because only local tasks can be waiting for the local real time clock, some special work has to be done if a task is moved between coprocessors while waiting for a real time clock match. Which of the functional blocks has to perform these functions is as yet unclear. Because the inserting of a task in the real time clock waiting list as a far from simple operation (the list should be kept in time order), it has to be done by one of the more 'powerful' functional blocks. For this to be possible, the LAN input controller must be able to give commands to this functional block in a 'preformatted' way (by using some special message registers).

**3.9 - Functional Blocks: Real Time Clock and Power Switch.**

## 3.10 Delays generator

The delays generator works almost exactly like the real time clock controller. The number of clock ticks per second is set by the CLOCK_TICK_SPEED parameter for the INIT_SYSTEM function call.

The other functional blocks maintain a linked list of processes waiting for some kind of timeout. This list is kept sorted on increasing time distance between the current time and the time when the process has to be restarted again. Each of the waiting process control blocks has a field containing the number of clock ticks between the time this process is restarted and the next process in the list is restarted (this number can be zero if the next process has to be restarted at the same time).

Inserting and deleting of processes in this list can be done by the host command interpreter (either following a command from the local host, or after reception of a command block from a remote host, as described for the real time clock functional block).

To restart a task, the delays generator writes a message to the task restarter. The task restarter has to remove the waiting processes from the delay list that have to be restarted at this time, and write the number of clock ticks for the next process to be restarted to the delays generator. If, while processing the current restarts, there have passed so much clock ticks that the next process has to be restarted immediately, then the task restarter should immediately continue with this process (a similar algorithm will have to be build in the real time clock functional block too).

The delays generator can be used to generate the timeouts for the LAN input/output controllers. This will simplify the LAN controllers hardware, and also makes it more easier to generate precise delays (if necessary).

The delays generator also contains a free-running 32 bit counter, incremented once for each delay tick. This counter can be read by the other functional blocks to update the three state timers for the tasks (one each for the running, waiting and suspended states). Each time a task control block is inspected or changed, the timers are updated using a 'last update time' field in the control block. This is done as follows:

1: Subtract the 'last update time' field from the current value of the free-running time counter.

2: Add the difference to the state timer that was 'running' since the last update.

3: Store the current value of the free-running time counter in the 'last update time' field.

**3.10 - Functional Blocks: Delays Generator.**

## 3.11 LAN input controller

The Local Area Network input controller handles the data stream coming from the other coprocessors in the local interconnection network. For this functional block and the LAN output controller, we currently assume that the interconnection link is in the form of a token ring network.

The LAN input controller is build in several 'layers', each layer handling progressively more complicated data structures.

### 3.11.1 LAN input lowest layer

The lowest layer handles the incoming bit stream, extracts clock and synchronisation information from it, deserialises the data into bytes, and presents a data stream consisting of bytes to the next layer (together with some signal lines, like 'sync detected'). Also, checksum calculation will have to be done here, with error reporting to the next layer. This layer is relatively simple, and has to be able to cope with high bit rates (on the order of 4 megabits/second). It will therefore be necessary to build this part as a (very simple) state machine or by using random logic.

For the IBM token ring, the lowest layer has to cooperate with the lowest layer of the output controller to change the data stream 'on the fly' (there is only a one bit time delay between the reception of a bit and the time it has to be transmitted again). This is necessary to change a free token into a busy token before transmission can start, and also to request a so-called priority token to be sent. Also, if an error is detected, then the 'error detected' bit in the closing flag will have to be detected and (if not yet set) set to the TRUE state. Another 'on the fly' operation is the setting of the 'message copied' bit as an acknowledge to the sending party (when the message was addressed to this station and there were no errors detected).

The 'message copied' bit is a special case when the message contained a process address resolution request. In this case, the higher levels of the LAN input controller have to use the process address search functional block to see if the searched process is currently in the local coprocessor. If this can be done fast enough, then the 'message copied' bit can be set immediately to indicate that the process is found (acting as an early acknowledge). If the searched process is found, then a message has to be sent to the requesting coprocessor containing the complete address of the process block - this message will act as an acknowledge to the address resolution request too.

### 3.11.2 LAN input intermediate layer

The next higher layer of the LAN input controller has to receive the messages and store them in a temporary buffer in the local working memory. This layer can ignore all messages not addressed to the local coprocessor, with three exceptions:

1: Messages with an 'all stations' address, like the address resolution request messages.

**3.11 - Functional Blocks: LAN Input Controller.**

2: Messages sent by the local LAN output controller. These messages have to be 'purged', while the LAN output controller has to be notified that this has been done (so that a new free token can be sent, possibly with a priority token as requested by another transmitter on the ring). Important data from such a message will have to be extracted, while the rest of the message can be ignored. All this important data is located in the message header and trailer bytes, and consists of the priority token request field and the 'error detected', 'message received' and 'message copied' flag bits. Handling this data is done by the LAN output controller.

3: Messages to be taken from the ring by a bridge, if the local MMTCP has bridge capabilities. In this case, the ring number should be checked and the packet should be taken off the ring if the host connected to this MMTCP can take care of the transport to the indicated remote network.

If a message is ignored or received with an error, then the temporary message buffer can be used again for the next message. Otherwise the LAN input controller will have to request extra memory space from the memory management functional block. If a message is received with an error while the 'error detected' bit was not set in the closing flag, then an internal error counter can be incremented (and an error message sent to the system error messages mailbox - maybe some 'filtering' will be necessary to prevent line noise from overflowing the system with error messages).

When an error free message has been received and stored into memory, the address of this message is sent to the highest layer in the LAN input controller functional block. These addresses can be stored in a small hardware FIFO, so that temporary delays in the handling of the messages can be bridged by queueing the new messages. If the FIFO is full, then the 'message copied' bit cannot be set, to indicate the sending coprocessor that the local coprocessor's input handlers are overflowing. If the message had an 'all stations' address, then it is necessary to set the 'error detected' flag, to request sending the message again. The FIFO buffer can be equipped with extra hardware to sort the packets on priority, so that packets with a high priority are handled before lower priority packets, even if they were received later.

### 3.11.3 LAN input highest layer

The highest layer in the LAN input controller functional block examines the received messages (removes their addresses from the 'FIFO' queue), and determines what to do with them. There are several possibilities:

1: The message contains units for a semaphore, data bytes for a pipe or mailbox, requested status data for a remote entity or things like that. In this case, the message is reformatted and sent to the task restarter. The task restarter is instructed to send the results (or resulting errors) from the restart action to a remote process, and the received data block can be destroyed (given back to the memory management functional block).

2: The message contains a status request for one of the local entities. In this case, the reply is placed in a memory block and sent back to the requesting coprocessor. The received data block can be destroyed.

3: The message contains a request from a remote process to wait at a local semaphore, mailbox, region etcetera (it is not possible for a remote process to wait for data coming from a pipe !). In this case, the request is placed in memory as a 'placeholder' block for the remote process, and the placeholder block is linked into the chain of waiting processes for the requested entity. The placeholder block can be recognised by the command processor and the task restarter, so that they automatically send their results to the remote coprocessor. If the command processor finds no errors while placing the placeholder block in the waiting list, then an acknowledge message is sent back to the remote hostprocessor indicating that the process is placed in the waiting list.

4: The message is a special message used while transferring a process between two coprocessors. In this case, the message may contain a part of a process control block, a pipe control block (including the pipe buffer), a waiting pipe data block, or a part of the pipe requester's waiting list. In all these cases, the received data packets require special handling like relinking of linked lists. The most logical place to do this processing is the highest level of the LAN input controller, which will therefore have to be build as a microprogrammed machine.

5: An acknowledging message is received for a message sent by the local LAN output controller. The contents of this message have to be analysed and acted upon. In some cases this means that the output buffer for the message originally sent can be released back into the memory pool. In other cases, a task can be restarted too.

6: The message contains data for a data stream. These messages are described in subchapter 2.12: 'Functional description: Stream data transfer'.

The highest layer of the LAN input controller will have to be build as a microprogrammed machine.

## 3.12 LAN output controller

The Local Area Network output controller generates the data stream going the other coprocessors in the local interconnection network. As already stated for the LAN input controller, we currently assume that the interconnection link is in the form of a token ring network (as proposed by IBM, Literature 1 and 2).

The LAN output controller will be build in layers, just like the LAN input controller.

### 3.12.1 LAN output lowest layer

The lowest layer works in close cooperation with the lowest layer of the LAN input controller. This is necessary to change bits in the message leader and trailer bytes, and to start sending messages at the correct time. Also, the clock needed to send the data may be generated in the lowest layer of the LAN input controller (with a digital or analog VCO). While data is being transmitted, this layer has to handle the serialising of the data bytes obtained from the higher levels, and also has to calculate and append the checksum at the end of the transmission. This layer should be able to generate leader and trailer bytes, and format empty tokens from them (to be sent around the token ring).

### 3.12.2 LAN output intermediate layer

The next higher layer of the LAN output controller is working just like the same level layer in the LAN input controller, but now in the opposite direction. This layer gets the addresses of blocks to be sent, and tries to send them out on the token ring. The addresses of the blocks can be stored in a hardware FIFO, so that some flexibility and buffering is possible. This FIFO buffer can sort messages on priority like the buffer used between the high and intermediate levels of the LAN input controller.

### 3.12.3 LAN output highest layer

This highest layer finds most of it's work in protocol and error control. The data to be sent is already stored in a memory block, together with instructions on what to do when the block has been sent. This layer uses timeouts to determine whether the interconnection link is used to heavily or not, and uses these timeouts together with the data extracted from a returned data block to determine what to do (a task may have to be restarted and/or a memory block may have to be returned to the free memory pool).

Another task that can be done by the highest layer might be the transferring of all the process data when a process is transferred to another coprocessor. This task consists of analising the data structure, doing some reformatting, sending all the related data blocks to the receiving coprocessor and finally removing the data structure from the local memory. This releases the host command interpreter from a lot of work, so that this important functional block is available much faster after a 'transfer process' message is received (following a successful CLAIM_PROCESS call in another coprocessor).

**3.12 - Functional Blocks: LAN Output Controller.**

The highest layer in the LAN output controller also plays an important role in the transfer of stream data. This will be explained in more detail in the section on stream data handling.

Timeout errors, if they occur frequently, will have to be reported in the system errors mailbox.

The highest layer of the LAN output controller may have to be build as a microprogrammed machine. To save chip space, it might be an idea to build this machine as an pipelined system, where a second process is interleaved with the LAN output controller. The most logical choice would be to make this the highest layer of the LAN input controller.

The LAN controllers, both input and output, should be able to act as (secondary) ring monitors, as described for the IBM token ring protocol. This means, amongst other things, that they should be able to detect the loss of a token, or the presence of a circulating free token. They should have the possibility to force a new free token on the ring, generate and receive 'beacon' frames, and to enter and exit the ring monitor state. Note that the ring monitor must insert an elastic buffer store in the bit stream, and the hardware to do so should be available on every coprocessor chip. If possible, hardware should be available on-chip to direct the data stream via several inputs and outputs (so that 'backup' rings can be formed).

**3.12 - Functional Blocks: LAN Output Controller.**

### 3.13 Stream and Bridge data handling

Stream data is used to transfer large blocks of data using the LAN that interconnects the multitasking coprocessors.

Bridge data consists of data packets which are to be sent to- or are received from an external LAN controller, connected to the host of a MMTCP.

Both these forms of data transfer involve a larger than normal number of data bytes (with 'normal' I mean the number of parameter or result bytes transferred for a standard function call). To let the host processor handle these data transfers may therefore be too slow, especially in the case of stream data, where the total number of data bytes may run in the millions.

In both the stream and bridge data streams, the data should be transferred into or out of the host memory as a linear array of data bytes, which makes it possible to use a simple Direct Memory Access (DMA) protocol.

DMA can be done by letting the multitasking coprocessor take over the system bus, but this necessitates making the multitasking coprocessor's host interface compatible with the various forms of bus acquisition and control protocols used by different processor families. This is obviously not the way to do it, because it may mean that we have to build several different multitasking coprocessor chips, each with a different host bus interface.

The other road to DMA is by using simple DMA 'request' and 'acknowledge' lines, to be connected to the host bus. The coprocessor simply requests data transfers with the 'request' lines, and the host system replies by returning (one of) the 'acknowledge' lines active while reading from- or writing to the coprocessor. In case multiple channels have to be supported, the interface should contain lines to indicate which DMA channel should be used for the current DMA cycle. Because this interface is inactive after reset, the exact hardware configuration of the interface can be made software selectable (no strapping pins needed).

This involves extra hardware on the side of the host interface (in the form of standard DMA controller chips), and therefore falls a bit outside the normal operation of the multitasking system hardware. This is because these DMA controllers have to be controlled by the host processor, and in most cases support only a limited number of data streams at the same time (simply the number of DMA 'channels' available).

For the stream data handling, there have to be processes running on the host processor that assign DMA channels to the requested data streams, initialise these DMA channels, and provide overall data transfer progress monitoring. In most operating systems, these 'stream control' processes will be part of - or at least very strongly depend on - the memory management processes that control the use of background memory and the process transfers.

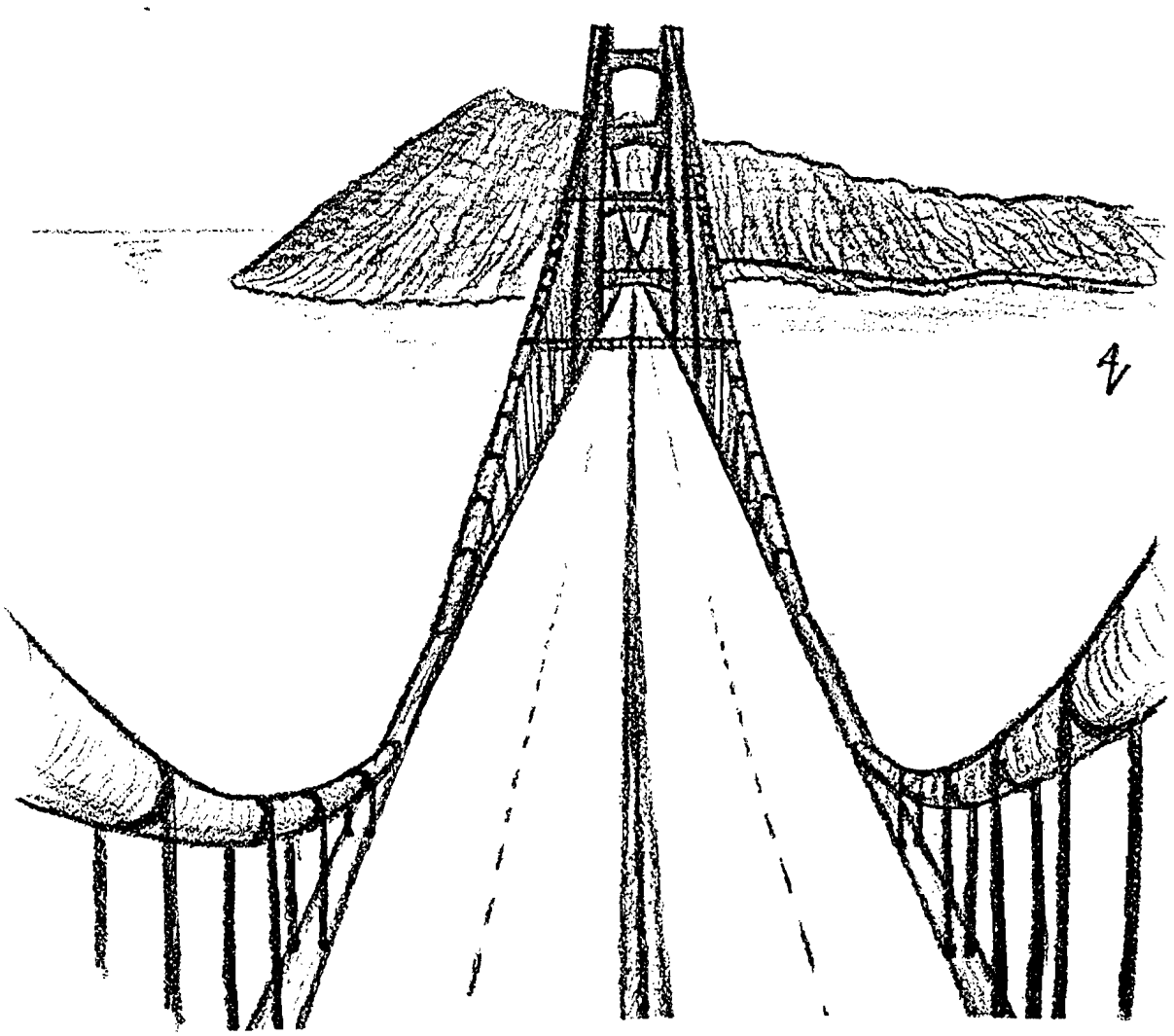
For the bridge data handling, host processes should be introduced to set up and control two DMA channels to transfer data between the MMTCP and the host memory. One channel is permanently assigned for reading from the coprocessor the data packets intended to be sent using the external LAN. The other channel is permanently assigned for writing packets received from the external LAN to the coprocessor. These bridge data DMA channels share the DMA hardware on the

**3.13 - Functional Blocks: Stream/Bridge data handling.**

coprocessor chip with the DMA channels for stream data. If a MMTCP is set up for connection to a LAN-equipped host processor, then two DMA channels are 'stolen' from the stream data handling (meaning that that particular MMTCP has two stream data DMA channels less than other MMTCP's).

The commands to establish a data stream are interpreted by the host command interpreter. The data streams themselves are sent and received by the LAN input and output controllers. Because these blocks contain most of the intelligence needed for these data stream transfers, only relatively simple hardware is needed to transfer the data back and forth between the local working memory and the host memory. This hardware can be seen as the on-chip counterpart of the DMA controller chips that control the actual reading and writing in the host memory.

In it's simplest form, the on-chip hardware consists of only a few loadable counters, a register to select the channel to use, a mode control register and a simple state machine controller. One counter is used to point to the local working memory word to be read or written, a second counter is used to count the number of data bytes to be transferred (this counter should have enough capacity to transfer a single data packet's length of data). Hardware should be provided to control the word-to-byte and byte-to-word conversions, if this is needed by the external hardware (the host bus interface contains the needed multiplexers and demultiplexers - they only need to be controlled in the correct way).



**3.13 - Functional Blocks: Stream/Bridge data handling.**

## 3.14 Chip test hardware

Even a VLSI chip of this size should be completely testable. This is only possible if it can be partitioned into blocks of manageable size that can be exercised separately.

The internal buses play a mayor role in the testing of the MMTCP chip. Via these buses, all functional blocks are connected together, and, if we use registers to exchange handshake signals via these buses, then these buses are also the only connections between the functional blocks. This makes it possible to use the internal buses to isolate the functional blocks from eachother and test them one at a time. Excitation signals can be fed into functional blocks by reading and/or writing via the internal buses, the response can be checked by monitoring the message flow following an excitation.

In principle, only the internal messaging bus need be equipped with hardware to force messages on the bus and monitor the responses, because most of the activities on the memory interface bus can be observed by looking at the pins connected to the external working memory. However, to check whether the external memory interface is working correctly, it should be possible to exercise the memory interface functional block with all bus cycles possible. This can be done by requesting them directly from the host interface (using special registers in the chip test functional block).

Generating excitation messages on the messaging bus is simple enough, this can be done with some special host accessible test registers to hold the register address and control bits, together with some registers to hold the data written or read.

Monitoring the message flow may pose some more problems, as this message flow can be very fast. This problem can be overcome by building some simple 'locking' hardware in the bus arbiter, which can prevent the bus from carrying messages until the external test hardware has read the results from the previous test.

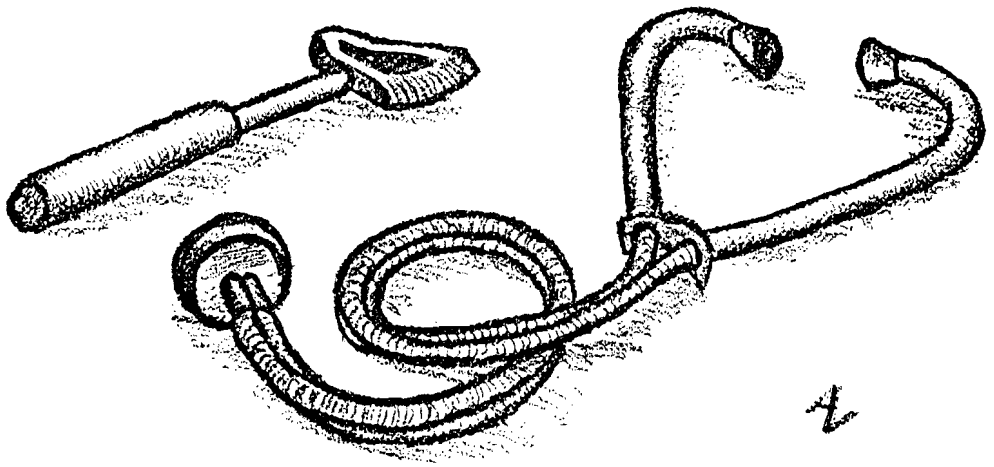
Another problem is that we must prevent functional blocks that are not the subject of the current test sequence from responding to messages generated by the functional block under test. This can be done with an extra 'address enable' line on the internal messaging bus. During normal operation, this line can be active, enabling the slave registers for reading or writing. During monitoring, this line can be set inactive, so that the slave registers do not respond at all to any messages (only special 'monitor' registers inside the test hardware functional block will be written and/or read).

Most functional blocks contain 'buried' registers to hold parameters during the execution of the MMTCP algorithms. These registers range from simple parameter registers (in the data path of a functional block) to state registers and microprogram counters and -stacks (in the control part of the functional blocks). The data path registers may be made accessible by adding some extra hardware to the interfaces between the main chip buses and the functional block's internal datapaths, but this is only possible if the functional block is not too complicated (having multiple internal buses, for instance). Especially for the registers in the control parts of the functional blocks, we will have to add extra hardware to be able to access them from the messaging bus (reading and writing, of course !). It might be wiser to connect these registers into a scan path, and connect the scan paths of the functional blocks to a specialised 'scan path bus' of some form, which

can be connected to some control and shift registers in the test hardware functional block.

Some of the functional blocks contain of specialised memory to speed up the algorithms. Connecting the memory cells together to form a scan path might take too much extra hardware (and, in some cases, might not be possible at all). When these memory structures are large enough, it might be worth wile to add special hardware at the periphery of these memories for a direct connection to the internal messaging bus. If the actual contents of these memories are not important for a functional test, it is also possible to include self test hardware (for a simple 'go - no go' test). Alternatively, we can try to force a functional block to fill it's internal memory by reading from one of the internal buses, after which we can monitor it's responses to excitations we send to the functional block via the messaging bus.

To prevent accidental or malicious use of the test hardware, I propose a special package pin to be used to enable the testing modes. If this pin is not in the active state, at least all the writes in the test registers should be disabled (in most cases, reading poses no problems).



**3.14 - Functional blocks: Chip test hardware.**

## 3.15 Thoughts for the future

The decomposition of the multitasking coprocessor described in this chapter is meant as a starting point for the actual realisation process, and is by no means definitive.

Some of the blocks may be split up into smaller blocks, other blocks may be combined into larger ones. It is even viable that completely new blocks will be introduced.

Combining functional blocks in such a way that they share a lot of hardware is possible for microprogrammed machines. As already indicated in this chapter, it is possible to run several microprograms in a 'pipelined' fashion. This will make them use common hardware while they remain logically independent (they will run slower, of course !).

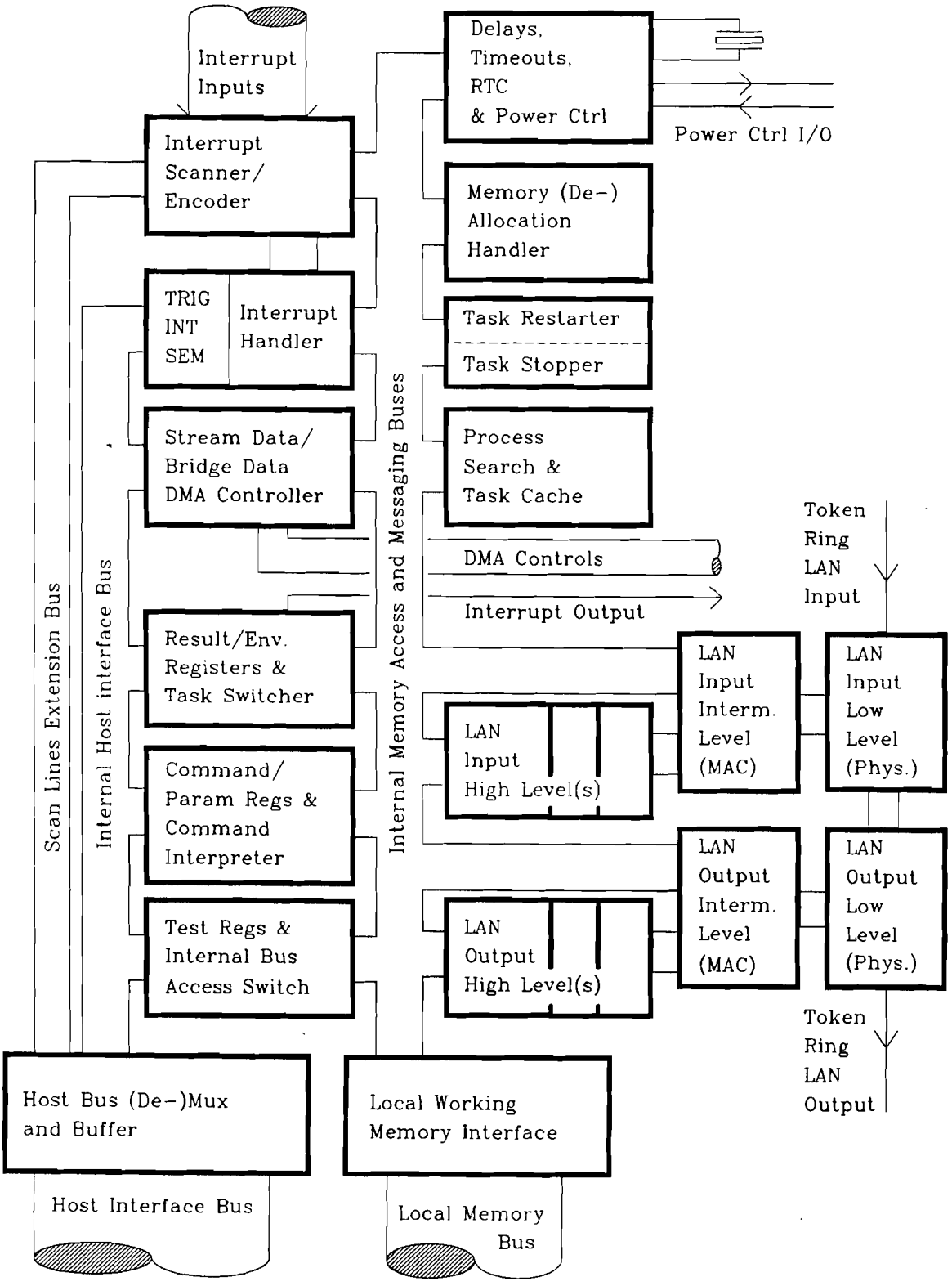
The actual implementation of the host task switcher and host result register handler is a bit fuzzy at the moment. It is very well possible that these blocks can be combined into a bigger functional block. To figure out what to do with them involves rigorously defining how they interact, and translating this interaction into hardware resources like registers and status flags.

The 'task restarter' takes over processing from the host command interpreter and other internal blocks if a function is to be performed that may restart a task. In the previous pages, the stopping of tasks is assumed to be done in the host command interpreter, but it is also possible to build a separate functional block to do this (or even incorporate these functions in the 'task restarter' functional block as low priority functions). In any case, removing this function from the host command interpreter will make it easier for the LAN input controller to handle packets received from remote coprocessors containing commands to stop a task in the local coprocessor.

Addressing of entities in a system of coprocessors interconnected by coprocessor LAN's, with bridge processors connecting these local LAN's together, may pose some problems. Addresses generated by host processes and addresses in packets received from the local interconnection ring always have to be checked (the 'accessible ring addresses' table may have to be consulted if the local host has bridge capabilities). It might be easier to build a specialised functional block to do this. This new block will then relieve several other blocks from this address checking, and might do it faster by using specialised hardware instead of (slow) microprograms.

## 3.16 Schematic overview

The following diagram gives a more detailed view of the functional blocks and their interconnections described in this chapter:




**3.16 - Functional Blocks: Schematic Overview.**

## 4. Conclusion

In the previous chapters of this report, I have presented my ideas for a hardware solution to the problems of multitasking and communication. Chapter one presented the background for the chosen solutions (the 'why'). Chapter two contained a set of functions that should be the starting point for the design of the hardware (the 'what'). Chapter three gave a breakdown of these functions into cooperating hardware blocks (the 'how').

Chapters two and three are by no means definitive. In the time to come, choices have to be made as to which functions will actually be incorporated in the MMTCP design.

Following this, the data structures and algorithms must be designed and tested, forming the design 'layer' just below the specification. These algorithms and data structures will be influenced heavily by the MMTCP interfaces to the outside world, especially the local area network connecting to other MMTCP's. Once the algorithms are known, we will have to take a second look at the functional blocks we are going to use to execute these algorithms.

The algorithms must be divided into steps that can be executed concurrently by the functional blocks. Once it is known what the functional blocks have to do, we have to specify the communication protocols to be used between them.

Once the internal communication protocols are known, we can start with the hardware design of the separate functional blocks. If we did our work right up to this point, we should not run into serious trouble with the actual hardware design. The hardware interface between the functional blocks consist of several buses, which have already been described in this report (on purpose in somewhat more detail than the other functional blocks).

In this report, I have not described the local area network protocols in much detail. I have simply assumed the network is there, and can be used to exchange messages between MMTCP's (without errors, keeping the messages in the correct order if necessary). My colleague Herman Vos is working his way through some very thick draft standards from ISO and ECMA, and has obtained the specifications for a group of integrated circuits Texas Instruments has designed to build a token ring LAN controller (the LAN type we want to use).

From his preliminary work, I have learned that the addressing and packet forwarding methods I had envisioned do not conform to these standards. This is possibly due to the fact that I look at these methods from the hardware point of view - the simpler the better. When packets have to be transferred by bridges and gateways, the problems even get worse, because the draft standards for doing this are very fuzzy (to say the least), and seem to be much more complicated than what I thought would be needed.

The chances that our MMTCP shares a token ring with other controllers may be very slim. Even if this situation occurs, then probably the only way the MMTCP's communicate with the other controllers will be by stream data transfer (if they communicate at all). It may be enough to make the communication between MMTCP's transparent to the other controllers, so that the MMTCP's have a logically separate network sharing the physical token ring hardware with other controllers. The problems get bigger again when packets must be transferred to other networks

across standard bridges and gateways - these problems must be solved if want to keep this possibility.

Another point for discussion is the placement of the LAN controller on the same chip as the rest of the multitasking coprocessor. Using a normal LAN controller connected to the host processor is possible (would work like a permanent gateway connection), but will place a very heavy workload on the host processor because the host will then have to handle a large stream of very small packets. Connecting a standard LAN controller directly to the coprocessor chip takes a lot of pins, is inflexible and maybe not even possible because most LAN controllers assume a host memory to be present and have a DMA bus master interface. Designing a special LAN controller to be connected to the multitasking coprocessor can be done, but again uses extra pins and necessitates the development of special interface protocols and hardware.

We keep getting back to the solution where the LAN controller is placed on the same chip as the multitasking coprocessor. In that case, the LAN controller can make use of the functions provided by the other functional blocks like working memory management and error handling. This integration also makes it possible to build a low component count system, a feature that system designers like a lot.

During this graduation period, I have worked on the construction of a 'solid' set of functions, always keeping in mind there had to be ways to implement them in hardware. This hardware directed thinking has resulted in the set of cooperating functional blocks described in chapter three. I think dividing the MMTCP into functional blocks is the only way to get working silicon in a reasonable amount of time (my estimate for the amount of devices needed runs in the hundreds of thousands).

This project needs a lot of work before a working version of an MMTCP can be shown to the public. I hope I can continue working on it because I do not like to leave work undone. Negotiations with Prof. Stevens have already started in that direction. I again wish to thank everybody who helped me during this period, I hope I am not too much in debt.

Eindhoven, july 8, 1987 Ad Verschueren.

## A. Some existing multitasking operating systems

To design a usable multitasking coprocessor, one must make sure that this chip will be easy to integrate in existing multitasking operating systems. The coprocessor should provide the functions of these operating systems where possible directly, if that is not possible then with simple software additions, or not at all if the functions are too much specialised. We should weigh the cost of implementing a function in hardware against the benefits we get from doing so.

This appendix compares several multitasking operating systems, which each have their own peculiarities (pointed out in each subchapter). The four multitasking operating systems described in this appendix are:

1) Intel's iRMX 86
2) Texas Instruments' Microprocessor Pascal System
3) LEX (a multitasking kernel developed in Eindhoven)
4) Bell Labs' UNIX

Because iRMX 86 is so powerful, I will use it to compare the other systems to. As we will see, the additions that have to be made are minor, and in most cases concentrate on semaphore usage and hardware interrupt interfacing (which was to be expected anyway). The set of functions provided by the MMTCP is based upon those provided by iRMX 86, because our original idea was to emulate iRMX with the MMTCP. Appendix B provides a direct comparison between the MMTCP and iRMX 86 (including higher operating system software layers within iRMX 86).

The MMTCP started out as the hardware form of iRMX 86, stripped from all memory management functions. The three other operating systems described in this appendix contained features that we have added to the original design to enhance it's functional capabilities. Pipes were added at that time to get a more general approach to communication between tasks.

Later, when the decision was made to include a local area network controller, new functions were included to control the controller as well as the network itself. Stream data handling was proposed to provide a means to transport large blocks of data between MMTCP's and make a more efficient use of the network (by doing stream transfers in the background).

## A.1 Intel's iRMX 86

This multitasking operating system is specifically written for the 8086 family microprocessors (there exist functionally compatible versions for other microprocessor families from Intel). It is a comprehensive system providing functions which are not commonly found in multitasking operating systems, but make life easier for the programmers. This system does not support virtual memory, but otherwise provides more functions than UNIX !

In iRMX, all parts of the system (henceforth called 'objects') are known under a 16 bit (unsigned) number, called a 'token'.

Priorities range 0..255 (unsigned byte), with priority 0 being the highest (all interrupts disabled). Processes with priorities in the range 0..128 all automatically run with certain interrupt levels disabled (the maximum configuration provides for 56 hardware interrupt sources, not counting the clock interrupt).

iRMX 86 can be bought in the form of a so called 'OSF' (Operating System Firmware) chip, to be attached to the host microprocessor. This OSF chip contains a 16 kilobyte read only memory, interrupt controller and several timer/counters (from which one is used for a real time clock). The iRMX code itself is placed in the read only memory, together with some standard peripheral chip drivers (serial i/o and floppy disk). According to Intel, it should be possible to get a computer system up and running using an OSF chip and only a few lines of user written software. This chip is not a multitasking coprocessor, as it uses the main processor to execute the iRMX operating system programs.

iRMX can be regarded a 'toolbox' for multitasking operating system writers. Functions not needed can be left out when the system is configured, new functions can be added when the need arises (this can even be done dynamically during execution !).

The following paragraphs each focus on one of the objects the iRMX operating system handles.

### A.1.1 iRMX jobs

A job is a working environment for all other iRMX objects, and basically consists of a pool of memory and an optional 'object directory' to store symbolic names for objects contained in the job. Each job contains at least one 'task' (executable program). A 'parameter object' can be given to a job during creation, which is simply a token for some kind of object in the system. Tasks running in the job can retrieve this parameter object and use it for a user defined purpose.

The jobs are organised like a tree, with a root job (initialised at power on) and child jobs. The root job initially gets all available memory (with a maximum of almost 1 megabyte), and divides this up amongst it's children. Children in need for more memory can 'borrow' memory from their parent.

The following operating system functions are provided for iRMX jobs:

**CREATE_JOB** creates a job with an initial task. Limits must be given for maximum size of the object directory, maximum number of objects and tasks, maximum priority for contained tasks and maximum and minimum memory pool sizes. An optional parameter object token can be specified.

**DELETE_JOB** deletes a job (may be postponed by DISABLE_DELETION, see A.1.10). The job must be childless and may not contain 'extension objects' (user specified data types).

**OFFSPRING** returns tokens for child jobs.

### A.1.2 iRMX tasks

Tasks are the active objects within iRMX, they contain executable object code, data and stack spaces. Tasks themselves are part of a job, a single job can have more than one task running in it. Each task has it's own priority (may be different from other tasks in the same job), and can be in one of several states:

| | |
|---|---|
| **running:** | Task has processor. |
| **ready:** | Task can be made running. |
| **asleep:** | Task waiting for event to happen, like triggering a semaphore, sending data to a mailbox, elapsing of a time interval, an external interrupt, etcetera. |
| **suspended:** | Task stopped, either by itself or another task. |
| **asleep/suspended:** | Task waiting for event and stopped (by another task). |

The following operating system functions are provided for iRMX tasks:

**CREATE_TASK** creates a task, returning a token for it. The following parameters must be given: initial priority, start address, data space address, stack space address, stack length and a flag indicating that the task will use a numerical coprocessor (iRMX will then save and restore the state of this coprocessor during task switching).

**DELETE_TASK** deletes a task (may be postponed by DISABLE_DELETION, see below).

**SUSPEND_TASK** increases the 'suspension depth' (a counter counting the SUSPEND_TASK and RESUME_TASK calls, initially 0), and suspends (halts) the task if it was not already suspended.

**RESUME_TASK** decreases the 'suspension depth' and changes the task state from suspended into ready or from asleep/suspended into asleep if the counter reached 0.

**SLEEP** waits for a specified period of time (puts the task in the sleep state).

**GET_TASK_TOKENS** returns tokens for: calling task, calling task's job, calling task's job's parameter object and the root job.

**GET_PRIORITY** returns the priority of the calling task.

**SET_PRIORITY** sets the priority of the calling task, priority cannot be set above the maximum priority specified for the job.

### A.1.3 iRMX semaphores

Semaphores are the means to exchange abstract 'units' between tasks in the system. iRMX semaphores are counting semaphores with one waiting queue, which can be set to operate strictly FIFO or in priority order. It is possible to send or request more than one unit at a time from the semaphore. The units counter is a 16 bits counter, allowing the accumulation of 65535 units (the actual maximum is software setable).

The following operating system functions are provided for iRMX semaphores:

**CREATE_SEMAPHORE** creates a semaphore, returning a token for it. The task queue mode (FIFO or priority based), initial and maximum number of units must be given.

**DELETE_SEMAPHORE** deletes a semaphore (may be postponed by DISABLE_DELETION, see A.1.10). All waiters are released (wake up out of their sleep state).

**SEND_UNITS** adds the specified number of units to a semaphore. If this would increase the number of units above the maximum, then an 'exception' (iRMX terminology for an error) occurs. Waiting tasks in the semaphore queue are released if their requests for units can now be satisfied. This is a 'V' operation.

**RECEIVE_UNITS** places the requesting task in the semaphore queue according to the queue mode. If the task is placed at the head of the queue, and there are enough units available, then the task is released from the head of the waiting queue and the units counter is decremented by the requested number of units (the 'P' operation). Otherwise, if the task does not want to wait, it is removed immediately from the waiting queue and a flag is set to indicate the failure. If the task is willing to wait, it is placed in the sleep state and is awakened again when the request is satisfied or until a specified time period is elapsed (this last function is optional and always removes the task from the waiting queue).

### A.1.4 iRMX mailboxes

Mailboxes are the means to exchange 'messages' (consisting of tokens) between the tasks in the system. Mailboxes have two queues: one for tasks waiting for messages (FIFO or priority based), one for the messages themselves (always FIFO). The message queue consists of two parts, a 'high performance' queue, for which there is always space reserved at creation time and an 'overflow' queue, for which memory segments are created when needed (a time consuming task). The size of the high performance queue is adjustable. Most of the mechanics of a semaphore apply to a

mailbox, like optionally putting tasks asleep if there are no messages available, putting an optional maximum to this sleeping period, etcetera.

Each message may contain one or two tokens: the first is the message token itself, which can be of any type. The optional second token is the so-called 'response' token, which should be a semaphore or mailbox. If it is given, it indicates that the sender will wait at the specified semaphore or mailbox until the receiver sends unit(s) or a message to the indicated place. This establishes a handshaking protocol between the sending and receiving processes.

The following operating system functions are provided for iRMX mailboxes:

**CREATE_MAILBOX** creates a mailbox, returning a token for it. The task queue mode (FIFO or priority based) and the size of the high performance queue must be given (4..60 messages, in blocks of 4 messages).

**DELETE_MAILBOX** deletes a mailbox (may be postponed by DISABLE_DELETION, see A.1.10), releasing all waiters and discarding all messages in the message queue.

**SEND_MESSAGE** places a message in the message queue if there are no waiters in the mailbox task queue, otherwise gives the message to the task at the head of the mailbox task queue, removes this task from the head of the queue, and releases this task from it's sleeping state.

**RECEIVE_MESSAGE** gets a message from the mailbox message queue if there is at least one stored there. Otherwise, if the task is not willing to wait for messages to arrive, returns with a flag indicating the failure. If the task is willing to wait, places the task in the mailbox task queue (entering the sleep state). Optionally, a maximum waiting period in the task queue can be specified, which, if elapsed, will be indicated with a flag.

### A.1.5 iRMX memory segments

The memory in the iRMX operating system is divided in 16 byte 'paragraphs', of which there are 65536, giving a total of 1 megabyte memory (this is the way the 8086 family of processors organises memory). iRMX segment tokens always indicate the starting paragraph number of the segment, so that it is easy for the processor to access the segment. For each segment, iRMX maintains the starting address, the length in bytes and the job owning the segment. Memory is allocated and deallocated dynamically, but no garbage collection is done ! If a job does not have enough memory available, iRMX tries to borrow memory from the parent job, enlarging this job's memory pool at the expense of the memory pool of the parent job (this borrowing of memory can 'chain' all the way back to the root job).

A job initially gets its 'minimum pool size' memory allocated (more if that is not enough to contain the initial task). If more memory is needed, memory is borrowed in blocks with a user defined size (optimisation problem !), so that the borrowed memory size is always round up to an integer multiple of this block size (which is set during system configuration). If memory is released and the job has more memory than it's minimum pool size, the process is repeated in the other direction (possibly returning memory to the root job). Trying to increase a job's pool size above the 'maximum pool size' parameter will always give an error. No memory

borrowing will occur if the minimum and maximum pool size attributes are the same.

The following operating system functions are provided for iRMX memory segments:

**CREATE_SEGMENT** creates a memory segment, returning a token for it. The size must be given in bytes and is always rounded up to a multiple of 16 bytes.

**DELETE_SEGMENT** deletes a memory segment (may be postponed by DISABLE_DELETION, see A.1.10).

**GET_SIZE** returns the size of a memory segment in bytes.

**SET_POOL_MIN** changes the 'minimum pool size' attribute of the containing job (optimisation problem !).

**GET_POOL_ATTRIB** returns a pool's minimum, maximum and initial pool sizes, the currently allocated and available number of paragraphs.

### A.1.6 iRMX regions

Regions are protected pieces of code. A task executing in a region cannot be suspended, deleted or preempted. Errors occurring in a region are handled at region exit. A region is in essence a semaphore with initially one unit. Regions should be used with great care, as tasks getting stuck in a region easily lead to a system crash.

Regions are only logically connected to code pieces. It is possible to use a region to protect different code pieces in separate tasks and/or jobs. Regions are user enforced, if a user program makes the mistake to execute code belonging to a region without using the system region calls ACCEPT_CONTROL or RECEIVE_CONTROL, then the region is rendered worthless.

The following operating system functions are provided for iRMX regions:

**CREATE_REGION** creates a region, returning a token for it. The waiting tasks queue mode (FIFO or priority based) must be given.

**DELETE_REGION** deletes a region, releasing all waiters (may be postponed by DISABLE_DELETION, see A.1.10).

**ACCEPT_CONTROL** lets the requesting task enter a region, but only if it is immediately available.

**RECEIVE_CONTROL** lets the requesting task enter a region if it is available, otherwise waits for it to become available. No maximum time limit can be specified !

**SEND_CONTROL** is called by a task to exit a region, and give the region to the next waiter in the waiting queue.

### A.1.7 iRMX objects

All objects in an iRMX system are designated by a token. There are lots of instances where a task needs access to an object, but does not know the token for that object. For this reason, an object can be given a user defined symbolic 'name' (a block of 12 bytes, which may contain any pattern). This name can then be stored in one or more of the object directories belonging to jobs, where it can be found by other tasks.

The following operating system functions are provided for iRMX objects:

**CATALOG_OBJECT** stores the name of an object plus the corresponding token in the specified object directory. If a task was waiting for the object to be catalogued, this task is released from it's sleeping state, giving it the wanted token.

**UNCATALOG_OBJECT** removes a token and name pair from the specified object directory.

**LOOKUP_OBJECT** searches a specified object directory for a given symbolic name, returning the object's token if found. Optionally, the calling task is put into the sleep state if the name is not yet present in the object directory (a maximum waiting period can be specified).

**GET_TYPE** returns the object type code for a given token. The object type code is a 16 bit unsigned number, with codes in the range 0..32767 reserved for Intel. Codes in the range 32768..65535 are reserved for user defined extension types (see A.1.11).

### A.1.8 iRMX exception handlers

Exception handlers are called automatically by the iRMX operating system if an error is encountered during the execution of a command. Errors are divided in two groups: 'program errors' which are mainly parameter errors (unknown token, value out of range, etcetera), and 'environmental errors' which are mainly caused by 'over-asking' the system (out of pool memory being the most common error). For each of the two groups, the user can decide to leave the error handling to the exception handler or the calling task itself (which then has to check the exception code returned by all the system calls).

The following operating system functions are provided for iRMX exception handlers:

**SET_EXCEPTION_HANDLER** sets the exception handler start address and exception mode (which error groups to handle) for the calling task.

**GET_EXCEPTION_HANDLER** gets the current exception handler start address and exception mode for the calling task. Usually used to temporarily switch exception handlers and/or exception mode.

**SIGNAL_EXCEPTION** is used by an operating system extension routine (see A.1.11) to signal an error back to the offending task or the exception handler for that task. An error code word, the number of the parameter which caused the error, the status word for the floating point coprocessor (if used) and a new value to be loaded in the user stack pointer must be supplied. By selecting the correct value for the stack pointer, SIGNAL_EXCEPTION can return to either the exception handler or to the user routine that caused the exception.

**RQ_ERROR** is a library routine (not a standard iRMX call) used to process error codes.

### A.1.9 iRMX interrupts

Interrupts are used to handle real time events. Hardware invoked interrupt routines can start an interrupt handler task, which in most cases is necessary because most of the iRMX calls are not available to the hardware interrupt routines.

High performance interrupt handling can be achieved by letting the interrupt routine and the interrupt handler tasks communicate via buffers that hold a user specified amount of data. The interrupt routine then only needs to invoke the interrupt handler task if it has finished handling a buffer, thereby reducing the number of task switches considerably.

The following operating system functions are provided for iRMX interrupt routines and tasks:

**SET_INTERRUPT** assigns an interrupt routine to a specified hardware interrupt level. The start address of the interrupt routine, the data segment address for the interrupt routine and a flag indicating whether the calling task will be the interrupt handler task must all be given.

**RESET_INTERRUPT** cancels the assignment of an interrupt routine to a hardware interrupt level and deletes the interrupt handler task (may be postponed by DISABLE_DELETION, see A.1.10).

**ENTER_INTERRUPT** is called by the interrupt routine to retrieve the data segment to use (specified by SET_INTERRUPT).

**EXIT_INTERRUPT** is called by the interrupt routine to send a hardware 'End Of Interrupt' signal and resume execution of the interrupted task.

**SIGNAL_INTERRUPT** is called by the interrupt routine to send a hardware 'End Of Interrupt' signal, awake the interrupt handler task (by sending a unit to an internal interrupt semaphore) and resume execution of either the interrupted task or the interrupt handler task, whichever has the highest priority (in almost all cases this will be the interrupt handler task).

**WAIT_INTERRUPT** is called by the interrupt handler task to wait at the interrupt semaphore for the interrupt routine to call SIGNAL_INTERRUPT (counting semaphore, so there may be an accumulation of SIGNAL_INTERRUPT calls).

**ENABLE** enables a specified hardware interrupt level if it has a routine attached to it.

**DISABLE** disables a specified hardware interrupt level.

**GET_LEVEL** returns the highest interrupt level which is currently being serviced.

### A.1.10 iRMX deletion control

All objects in iRMX have a so-called 'deletion depth' counter (unsigned byte), which must be zero to enable the normal deletion of the objects. If a task tries to delete an object with a non-zero deletion depth, then the task is placed in a sleep state until the deletion depth is made zero by another task, after which the object is deleted and the task is awakened again.

The following operating system functions are provided for iRMX deletion control:

**DISABLE_DELETION** increases the deletion depth of an object, making it impossible to delete the object.

**ENABLE_DELETION** decreases the deletion depth of an object. If the deletion depth reaches zero, the object can be deleted by the normal delete calls.

**FORCE_DELETE** deletes the designated object if the deletion depth is zero or one (one DISABLE_DELETION call more than there were ENABLE_DELETION calls).

### A.1.11 iRMX user extensions

There are two types of user extensions possible in the iRMX operating system. New operating system calls can be added to the system by placing the start address for the handler routine in the processor hardware interrupt table, the other extensions concern the addition of new object types by combining existing object types in so-called 'composites'.

The following operating system function is provided to add new operating system calls to the iRMX system:

**SET_OS_EXTENSION** places a pointer to a routine in a specified slot of the processor hardware interrupt table. Slots 224..255 are reserved for this purpose. A 'null' pointer may be set to decouple a routine from the interrupt table.

Composite object types are managed by so-called 'type managers' (which are normally set by SET_OS_EXTENSION). Each composite object type has it's own user defined type code (in the range 32678..65535), a number which must be used to create composites of that type and an optional 'deletion mailbox', which is used to send deleted composite objects to.

In most cases, one of the tasks in the type manager is responsible for the objects sent to the deletion mailbox and has to do some processing for each 'deleted' object before actually deleting it. If the composite object is a disk file info block, for

instance, this processing might be the writing of updated data and directory sectors to disk after closing a file. If no deletion mailbox is specified, deleting is done without informing the type manager.

A job containing extension objects cannot be deleted, because this would also delete the corresponding type manager which must be present in this job. Preferably, the system should first be 'flushed' from any existing composite objects, after which the extension, the type manager task(s) and the job can be deleted.

The following operating system functions are provided for iRMX type managers:

**CREATE_EXTENSION** creates a new object type, returning a token for it. The type code and the optional deletion mailbox must be specified. Nothing is said about the actual contents of the new object type !

**DELETE_EXTENSION** deletes an object type (may be postponed by DISABLE_DELETION, see A.1.10). If a deletion mailbox was specified, all existing objects of that type will automatically be sent to the given mailbox (which means that they will be 'taken away' from the processes using them, which may give a host of error messages around the system).

**CREATE_COMPOSITE** creates a composite object, returning a token for it. The object type code and a list of tokens comprising the composite object must be given. This list may be partially empty, the tokens may be of any valid type (including other composite object types !).

**DELETE_COMPOSITE** deletes a composite object, sending it to the deletion mailbox if one is specified for the extension type. If this function is called by the type manager, then the composite is deleted without sending it to the deletion mailbox.

**INSPECT_COMPOSITE** returns a pointer to the token list for the given composite object.

**ALTER_COMPOSITE** replaces specified tokens in a composite object's token list with other tokens.

## A.2 Texas Instruments' Microprocessor Pascal System

This multitasking operating system is specifically written for the TMS 9900 series microprocessors from Texas Instruments. The system is designed around an extended version of the Pascal language, where almost all parts of the system are written in Pascal.

Microprocessor Pascal (which I will call 'MP' from here on) does not use the 'monitor' system to achieve multitasking, but rather uses an approach that looks much like the iRMX jobs/tasks approach, the standard Pascal layers Program - Procedure have been extended to System - Program - Process - Procedure, where processes and procedures may be nested at will. Memory management is done by extending the Pascal 'heap' management, where heaps may be nested, and memory may be allocated in blocks of arbitrary size.

MP is designed to be written and debugged on a host computer, after which it is transferred to the target system, which will probably be much smaller. In fact, the 9900 chip is the one chip implementation of a minicomputer designated with the type number 990. The 9900 is a 16 bit microprocessor with an addressing range of only 64 kilobytes (small addressing ranges are not uncommon in minicomputers, a normal PDP-11 only has 128 kilobytes to work with !). The register set is placed in memory, which aids in ultra fast task switching by simply reloading an on-chip pointer to the current register set (this is also the only thing which is really fast with this processor, input and output is done serially through a bit-oriented so-called 'communications register unit' or CRU).

All data spaces are located in the Pascal heap, including semaphores, stacks, programs etcetera. Addressing of data is for the most part done during compilation of the software by the MP compiler (no object directories needed - but rather inflexible).

Priorities range 0..32767, with 0 being the highest priority, which is reserved for the bootstrap/reset process. Priority 32767 is reserved for the idle process, which is started when the processor has nothing else to do. Priorities in the range 1..15 are connected to the hardware interrupts, processes executing at these priorities are so-called 'device processes'. The scheduler places device processes before other device processes with the same priority in the ready queue, while the normal processes are placed following all other ready processes with the same priority (which is the strategy employed by all other multitasking systems described here). The reason for this strange strategy lies in the fact that the hardware/software combination allows the enabling of an interrupt level before the interrupt process at that level has finished executing. With some external hardware (an Intel 8259, for instance) a single interrupt level can be divided into separate levels. This is made possible with this 'strange' LIFO scheduling algorithm.

Standard MP does not support mailboxes, regions, object catalogues, deletion control and operating system extensions. Semaphores are counting (maximum count 32767), but with only one unit at a time. A semaphore queue always uses the FIFO algorithm (cannot be priority based).

Timed waits and delays are not amongst the standard functions, but can be included in the system during configuration. Timed waits (TWAIT call) only operate if the process is at the head of a semaphore waiting queue.

Each interrupt can trigger one of two semaphores. An interrupt triggers the normal interrupt semaphore only if there is someone waiting at that semaphore, otherwise a so-called 'alternate' interrupt semaphore is triggered. This idea to use two semaphores for each interrupt is used in an extended form within the MMTCP because it provides a more flexible way to catch 'overflow' interrupts.

Separate interrupt routines and interrupt handler tasks are not supported (not needed anyway, as the hardware performs a task switch automatically in response to an interrupt - very fast indeed). It is possible, however, to connect a user written (assembly language) interrupt routine to a hardware interrupt. This assembly language handler can call MP routines, but this is not specifically supported (you have to know exactly what you are doing, otherwise the system will crash).

From the process related MP procedures only the following is a bit unusual, and is not directly available in iRMX:

**SWAP** is used to implement time slicing. It is normally called periodically by a device process (in most cases the real time clock). It works by searching the ready processes list for the first non-device process, removing this process from the list, and re-inserting it just before the first process with a lower priority.

The following MP procedures related to semaphores are available:

**SIGNAL** is a completely standard 'V' operation, sending one unit to the semaphore counter.

**WAIT** is a completely standard 'P' operation, halting the calling process until the semaphore has at least one unit accumulated.

**INITSEMAPHORE** initialises a semaphore, giving it an initial counter value. This call returns a valid variable of the type 'SEMAPHORE', to be used by all other semaphore related calls.

**CHKSEMAPHORE** is a boolean function returning the value TRUE if the specified semaphore is a valid (read: initialised) semaphore.

**TERMSEMAPHORE** deletes a semaphore, making it's space available for another INITSEMAPHORE call (the maximum number of semaphores is fixed during compilation).

**SEMAVALUE** is an integer function returning the current counter value of the specified semaphore. If positive, this equals the number of accumulated 'units', if negative, it gives the number of waiting processes. Care should be taken by using this function, because it is very well possible that another process is made running between this call and the testing of the returned value - this process can change the semaphore state !

**SEMASTATE** is a function returning a value of the type 'SEMAPHORESTATE', which is (AWAITED, ZERO, SIGNALED). It functions the same way as SEMAVALUE, and should be used with the same care.

**WAITSIGNAL** does a WAIT on a specified semaphore, and after that, immediately does a SIGNAL on a second specified semaphore, in one indivisible step.

**CSIGNAL** sends a SIGNAL to the specified semaphore, but only if the semaphore has waiters. A boolean 'VAR' variable indicates if there were any waiters.

**CWAIT** does a WAIT on the specified semaphore, but only if this semaphore has units accumulated (it will always return immediately). A boolean 'VAR' variable indicates if there were units accumulated.

The following MP procedures related to interrupt handling are available:

**EXTERNALEVENT** assigns a primary semaphore to a specified hardware interrupt level.

**NOEXTERNALEVENT** detaches the primary semaphore from a specified hardware interrupt level.

**ALTEXTERNALEVENT** assigns an alternate semaphore to a specified hardware interrupt level.

**NOALTEXTERNALEVENT** detaches the alternate semaphore from a specified hardware interrupt level.

**ASSEMBLYEVENT** connects a register set (16 16-bit words at an arbitrary memory location) and assembly language interrupt handler to a specified hardware interrupt level.

**NOASSEMBLYEVENT** disconnects an assembly language interrupt handler from the specified hardware interrupt level. MP will handle the interrupt again.

**INTLEVEL** returns the level of the interrupt currently in service (1..15). It returns -1 if there is no interrupt in service.

**MASK** disables all hardware interrupts (except level zero, which is the system boot interrupt).

**UNMASK** enables all hardware interrupts with higher priority than the calling process.

**SETMASK** enables all hardware interrupts with higher priority than specified in one of the call's variables. A second 'VAR' variable receives the interrupt mask level as it was before this call.

The following MP procedures become available when the real time clock services are included in the system during configuration:

**CLKINT** is not a call, it is a separate program. It must be started during system initialisation with as parameters the number of milliseconds between clock interrupts, the number of milliseconds between 'SWAP' calls and the Communications Register Unit address of the real time clock hardware.

**DELAY** will put the calling process 'asleep' for a specified number of milliseconds.

TWAIT performs a timed WAIT on a specified semaphore. It does not work if there is already someone else waiting at the semaphore. A 'VAR' variable indicates whether the specified semaphore received a SIGNAL within the given waiting time, or whether this time (specified in milliseconds) expired without a SIGNAL being given.

Texas Instruments' Microprocessor Pascal was developed with a single microprocessor in mind. It has never been used on another processor type, but is nonetheless interesting to include in this comparison because it contains functions not available in the iRMX operating system.

The most noticeable additions are the 'SWAP' function to implement time slicing, the 'WAITSIGNAL' call to do an indivisible WAIT and SIGNAL on two semaphores and the conditional CWAIT and CSIGNAL functions. The idea to have alternate interrupt semaphores is interesting too (and is used in the MMTCP).

## A.3 LEX (developed in Eindhoven)

LEX stands for 'Local EXecutive', and is a small multi-tasking operating system used to drive the slave i/o processors in a multiprocessor UNIX implementation, developed by our group in cooperation with a group at the Nijmegen University (Netherlands).

LEX is written in 80186 assembly language and C. It handles processes (priorities 0..32767), semaphores (counting, max. count 32767), memory segments (dynamic allocation and de-allocation), mailboxes and interrupt handlers. Only the last two will be described here.

Mailboxes in LEX contain a fixed number of slots, which is specified at compile time. Each slot can hold a single 32 bit message. The following LEX calls are available for mailboxes:

**Rmail** clears a mailbox, releases processes waiting to read from the mailbox and releases processes waiting to write into the mailbox (if a mailbox is full, processes requesting to write can be put asleep until space becomes available).

**Vmail** sends a message to a mailbox. A 'mode' parameter determines the action taken when the mailbox is full:

* The process can be put asleep in the mailbox's writers waiting queue until space becomes available.

* A status flag can be used to indicate the failure (returning immediately to the caller).

**Pmail** is called to receive a message from a mailbox. A 'mode' parameter determines the action taken when the mailbox is empty:

* The process can be put asleep in the mailbox's readers waiting queue until it is at the head of this queue and there is a message written to the mailbox.

* A status flag can be used to indicate the failure (returning immediately to the caller).

In the present configuration, LEX can have a maximum of 8 hardware interrupts with corresponding interrupt handler routines. Interrupts can be coupled to interrupt handler routines in two ways:

* **'Direct'**, which means that the environment of the interrupted process is not saved and no task switch is possible. The interrupted process is always restarted when the interrupt handler ends its processing.

* **'Indirect'**, where the environment of the interrupted process is saved and task switching is possible. The interrupt handler may call any of the semaphore or mailbox handling procedures.

Being a relatively simple system (only a few kilobytes of object code), LEX has not much functions to add to those iRMX and MP offer. The only new idea is the mailbox writers waiting queue, which may have it's advantages above the scheme of the 'infinite' mailbox that iRMX offers (a process going haywire cannot flood the system with meaningless messages). LEX has a 'proceed' call that does almost exactly the same as the 'SWAP' call in MP, and can be used for time slicing.

## A.4 Bell labs' UNIX

UNIX is a multiuser/multitasking operating system, in that order. It is certainly not meant as a real time multitasking operating system, and for that reason, it does not provide much of the functions available in iRMX. If functions are available, they are implemented in a (sometimes radically) different way. Because UNIX and it's relatives are used in an increasing number of computer systems, the MMTCP should be capable of providing the basic multitasking functions used within these operating systems.

The UNIX operating system uses a virtual memory scheme to store the data and code segments of it's users (by the way, UNIX calls a code segment 'text'). The code needed to play this game of swapping data in from- and out to disk at the most profitable times clobbers the UNIX kernel programs in such a way that they become very hard to read (being written in C, with hardly any comments at all does not help either).

The multitasking calls are not available to the users of the UNIX system, they are only available to the system programs themselves. This is done to protect the system from misuse, because the implementation of semaphores makes it easy to crash the system (for instance by blocking the scheduler !).

Semaphores (UNIX calls them 'channels') have no memory nor a real waiting queue in the UNIX system. Waiters for a channel are all transferred from the waiting 'queue' to the ready queue at the first 'wakeup' call for that channel. The ex-waiters are placed in the ready queue ordered according to a temporary priority, given to them when they were put asleep. A channel with no waiters ignores all 'wakeup' calls.

Because all waiters are made ready at the same time, they all have to check whether the condition they waited for really has been set. If waiters had been waiting for the release of a resource, then the first ex-waiter to reach the running state will occupy the resource, blocking the the other ex-waiters (which will have to put themselves asleep again).

A channel is identified by a user provided number (in most cases the virtual memory address for a related variable). This is in contrast to the other operating systems, where the operating system provides a 'token' or identifier of some kind to be used when accessing the semaphore after creation. This is also where the security problem lies - UNIX will not stop you if you send 'wakeup' signals to all the available channels.

Being a virtual memory based operating system, UNIX has to do a lot more process accounting than the other operating systems described in this chapter. Making a process ready is not so simple either, as the process data and text segments may be swapped out during the waiting period. The process accounting functions amount to keeping track of the time the process has actually been using the processor, the time the process has been resident in memory since swapping in and the time passed since the process has been swapped out to disk. If a process has children, then these times for the children are also accumulated in the parent's process block.

Large UNIX installations do process 'profiling' (keeping track of the resources the process uses) to influence the time slicing and virtual memory algorithms.

The process priority consists of two parts: a so called 'nice' variable and the actual priority itself. The 'nice' priority can be changed by the user, but only a so-called 'superuser' can increase his/hers 'nice' priority. The real priority is initialised directly from the 'nice' setting, but during processor usage is decreased gradually each second by the clock interrupt handler, leading to an intelligent 'soft' timeslicing algorithm for the user processes. System processes have a 'nice' priority above a certain level, which prevents the clock interrupt to change their real priority. Priorities in UNIX are signed bytes, with the most negative value (-128) being the highest priority.

The virtual memory handler is a separate process inside the UNIX kernel, called 'sched' for scheduler. The scheduler is an endless loop, which starts by waiting for a process that can be swapped in. If there is enough memory available to store the process, the data and (if needed) text segments are loaded and the process is stored in the so-called 'runqueue' of processes that are runnable.

If there is not enough memory, then the scheduler checks the process list for processes that are waiting at a channel while having a low priority. If these are found, they are swapped out until there is enough memory space to swap the new process in. If there are no such processes, the scheduler starts swapping processes out that have been in memory longer than a preset time interval (but this only if the process wanting to be swapped in has been out on the swap disk longer than another preset time interval). If none of these two methods free enough memory to swap the new process in, it simply has to wait (the scheduler then goes asleep for one second before it starts checking again).

The scheduling algorithm is probably the most changed part in the UNIX kernel. Each system integrator thinks he/she can write a better one, resulting in completely different schedulers (the one I described runs on the UNIX system on which LEX also resides). Fortunately, the scheduler is a separate kernel process, which only uses process data and data gathered by other processes (the real time clock, for instance) and uses the standard system calls 'sleep' and 'wakeup'.

The UNIX kernel has it's own way to handle time delays. The UNIX kernel maintains a so-called 'callout' table which contains an indicator for the time delay wanted, a procedure to call when the delay time has expired, and a pointer to an argument to give to the procedure. The process that uses a callout is not put asleep (as the 'SLEEP' call in iRMX does), but rather commands the operating system to call a specified procedure when the delay time has expired.

The 'SLEEP' call can be simulated by letting the routine that is run in response to the callout execute a 'wakeup' for a channel the target process has put itself asleep on.

Simulating the callout with a 'SLEEP' is possible by creating a separate process containing only the wanted procedure and putting this process asleep for the specified time, after which the process wakes up, executes the procedure and (optionally) kills itself.

Process descriptors and callouts are stored in arrays of fixed size. This has the disadvantage that there exist upper limits to the number of processes and callouts, and that a lot of space is wasted if these arrays are not filled completely.

'Pipes' are used by the UNIX operating system to transfer data between processes that run concurrently. The processes do not know they are communicating via pipes, because the operating system kernel has initialised the pipes together with the processes themselves (rather inflexible). Pipes are buffered on disk and are normally used to create 'strings' of processes that handle data in separate steps, transferring data from one process to the next in the string (in one direction only).

The pipes provided by the MMTCP can be used for this purpose and then offer the possibility to transfer data between processes running on different processors. They also can be created and deleted at will by the running processes, like the 'stream files' described in the next appendix.

## B. iRMX Kernel, BIOS and EIOS emulation

This appendix takes a look at the possibilities and difficulties with emulating the iRMX-86 kernel, BIOS (Basic I/O System) and EIOS (Extended I/O System) when using the multitasking coprocessor.

The MMTCP is based for a large part upon the functions provided by iRMX, and was intended to emulate iRMX in the first place. Because of this, it is necessary to check whether the current specifications for the MMTCP make this emulation impossible (in which case we will have to do our homework again).

## B.1 iRMX Kernel emulation

To emulate the iRMX kernel, 'job' management is needed. jobs are a way to partition memory in the iRMX system. jobs are organised in a tree-like fashion, with a root job (initialised upon system powerup), parents and children.

Each job has as one of its parts a 'pool' of memory to work with. When a job is created, it is given a minimum and maximum amount of memory that it may have. This pool actually only consists of size limits, it is not the real memory space. Memory is allocated in segments, which are allocated and de-allocated dynamically.

Initially, a job gets the minimum possible amount of memory allocated. If this memory is exhausted, the job can borrow memory from its parent job (and this one can borrow memory too, to satisfy this request, all the way up to the root job). If memory is deallocated, it is given back to the parent job until the minimum allocation is reached again (the parent returns memory to its parent too, etcetera). The minimum amount of memory borrowed and returned can be specified when the system is set up. This memory management can be done by a procedure in the kernel which can use a coprocessor supplied region semaphore to provide the necessary mutual exclusion.

Each job has an 'object directory', which is used to store symbolic names for the objects in the system. Tasks running in a job can add, delete and search for entries in this object directory. This can be done by kernel procedures which can use a region semaphore to provide the necessary mutual exclusion (one region per object directory). When searching an object directory, iRMX allows a task to wait until the wanted object is placed in the directory. This can be achieved by using a semaphore associated with each object directory, at which processes can wait which do not find the object they waited for. This (no memory, no queue) semaphore can be triggered when a new entry is placed in the object directory, causing the waiting processes to search again (other schemes can be used too, of course).

Tasks (processes) running in a job can be emulated directly. All process related iRMX calls are reproduced in the MMTCP, the same goes for the semaphores, mailboxes and regions. The jobs themselves are <u>not</u> emulated by the MMTCP because they are related to memory management, which is on purpose not included within the MMTCP (there are too much different algorithms used to do this, and all of them claim to be the 'best'). Therefore, jobs must be handled by user written software (using MMTCP provided basic functions, of course).

Deletion control is reproduced almost completely by the coprocessor. The only thing failing is the following: If a process in iRMX tries to delete an object with a non-zero deletion holdoff counter, then this process is placed in a waiting queue until the deletion depth is made zero, or until the object is deleted with FORCE_DELETE. This behaviour is not directly supported by the coprocessor, but it can be simulated by introducing a non-counting, no-memory semaphore for each object. A process trying to delete the object while the deletion depth is non-zero is placed in the waiting queue of this semaphore, to be released when an ENABLE_DELETION call decreases the the deletion holdoff counter to zero, or until a FORCE_DELETE call deletes the object.

Exceptional condition (error) management should be done by the host software kernel. The coprocessor only returns an exception (or error) code to the calling

process, errors should be intercepted in the kernel software, and should start the correct exception handlers.

Interrupt management is different from that within iRMX, because the interrupts are handled differently in the coprocessor hardware. Directly coupled interrupts can use the TRIG_INT_SEMAPHORE call to simulate the SIGNAL_INTERRUPT call. The indirectly coupled interrupts can be used to start processes which substitute for the iRMX interrupt tasks. It may be better, however, to change the interrupt philosophy to better suit the hardware interrupt handling done by the coprocessor.

Operating system extensions are used in the iRMX kernel to introduce new functions in the operating system. These 'extensions' are really a specification of how these functions should be added to the system, and how they should be called by the user software.

Type managers are processes which are used to introduce new data types into the iRMX operating system (like the 'user', 'device connection' and 'file connection' data types used by the iRMX BIOS, described below). These functions can make new instances of the new data type entities, change and delete them. For the deletion process, the 'tokens' for the entities to be deleted are optionally sent to a 'deletion mailbox', for which a normal coprocessor-provided mailbox can be used.

## B.2 iRMX BIOS

The iRMX BIOS (Basic Input/Output System) mainly consists of so-called 'asynchronous' function calls. This means that a command given to the BIOS runs concurrently with the calling process. Using the BIOS is done in several phases:

* The calling task does a normal procedure call to the BIOS procedures. This starts the first part of the processing, which is called 'sequential', because it is not done concurrently with the calling process. The following two parameters are always present:

  1) A pointer to a memory location which is to receive the condition code for the sequential part of the BIOS call (this is standard procedure for all iRMX function calls).

  2) A mailbox which is going to receive the outcome of the 'asynchronous' part of the bios call. This outcome can be in two forms, the first being a 'token' for a so-called 'result segment', which contains a detailed description of the outcome of the call, the second being a token for a device or file connection, described later.

  During the sequential part of the function call, the given parameters are checked for inconsistencies, and any errors in them are reported in the memory location described under 1). If no errors are found, a command 'package' is made and given to the process which will execute the function (probably using a mailbox).

* The process which executes the asynchronous part of the call starts to work as soon as it receives a message in its command mailbox, and uses the data stored in the command package. When it has finished, it has two options. It can place a token for a device or file connection into the response mailbox (if it was the meaning that such a token should be returned, and that no error was detected). The other option is to create a memory (result-) segment, using memory in the caller's job memory pool, and place the result there.

A task can have a maximum of 255 asynchronous BIOS calls standing out at the same time. A result segment should be deleted when it has been read. To speed things up a lot, the WAIT_IO function can be used following the calls A_SEEK, A_READ and A_WRITE (the most commonly used calls). This function lets the task wait at the response mailbox, until a response is placed there. The result segment is returned immediately to the BIOS for re-use, so that it need not be deleted (this saves a lot of overhead). WAIT_IO is a normal RECEIVE_MESSAGE call, with some extra work done by the driving software.

It is perfectly possible that a process executing the asynchronous parts of a BIOS function call may change the order of execution of the commands, to get a better performance (ordering disk read and write commands in the cylinder order, so that the head needs not move in a completely random way across the disk surface, for instance). It is therefore possible that results arrive in the response mailboxes in a different order than in which the commands were given.

The iRMX BIOS has three file types:

* **Named files** are the files stored on the random access background memory (disk, bubbles). They are stored in a tree-like structure, not unlike UNIX, with a root directory and subdirectories. Named files can be shared and have access control. Owner, group and world 'users' are recognised and can be given different access rights to the files (a subdirectory is a file too !).

* **Physical files** occupy a complete device. The device is seen by the operating system as one gigantic stream of bytes, without subdivisions. These files can be used to control terminals, tape drives, printers, and are also used to read and write foreign disk formats (and during the formatting process of a normal iRMX disk too). No access and/or sharing control is possible, it will have to be done by the user programs (or the Extended Input/Output System, the EIOS). Note that sharing itself is possible !

* **Stream files** are used between processes and are normally used to send byte oriented data from one process to another process. They can be implemented by using pipes in the coprocessor, as long as they are used by only a single reader. iRMX allows multiple readers and writers for a single stream file, although the BIOS manual does not state what will happen in that case.

A 'device connection' is an object which represents a device in the system. It is created with A_PHYSICAL_ATTACH_DEVICE, which returns a token for the newly attached device. The connection is broken by calling A_PHYSICAL_DETACH_DEVICE. Only one device connection is allowed per physical device. The stream files are all stored on the virtual device 'stream' (for consistency reasons only, there is no real device involved !).

A 'file connection' can be obtained with a call to one of the following functions: A_ATTACH_FILE, A_CREATE_FILE or A_CREATE_DIRECTORY. For these three functions, a file- or device connection and a 'path' should be given. If the path is empty, and a file connection is given, then a second connection to the given file is obtained, which can be used for shared access to the file. If the path contains a '^' sign, it is used to denote the 'parent' directory of the file ('..' in UNIX and MS-DOS). During this call, the access rights of the caller are computed and stored in the file connection data structure (at least, that is what I think that is done). The file connection can be broken with A_DELETE_CONNECTION. Files and (empty) directories can be deleted with A_DELETE_FILE.

Having obtained a file connection, the file can be opened by using the A_OPEN call. Using this call, the wanted type of access (read, write or read-and-write) and the allowed sharing possibilities (none, readers-only, writers-only or full sharing) must be given. As for sharing, no record lockout or even file lockout is implemented by the BIOS, this is left up to the users. The A_CLOSE call closes the file for access, but the connection remains intact (A_OPEN can be called again).

While a file is open, the calls A_READ (for reading), A_WRITE (for writing), A_SEEK (to move the read-write index) and A_TRUNCATE (to 'chop off' the end of the file) can be used to read and manipulate the file.

A_UPDATE can be used to force the writing of the file buffers to the device. During system installation, the user can specify that this has to be done once every XX seconds, and/or that it has to be done XX seconds after the last access to the

device. This can easily be simulated by using a separate process which is timed with the DELAY function provided by the MMTCP.

Making this system suited for a multiprocessor environment is not very difficult, if the processes remain relatively static (do not move between the processors). Block input and output (from and to disk drives, for instance) can be handled by using DMA between the host processors, DMA can also be used to transfer command and result blocks between the host processors, byte input and output can be handled by the pipes implemented in the coprocessors. The coprocessors automatically can handle system global semaphores, mailboxes and regions. By giving each processor its own memory (job) manager with a separate root job, memory management can be done independently for each host processor.

## B.3 iRMX EIOS

The EIOS (Extended Input/Output System) builds upon the Kernel and the BIOS and adds the following features:

* Buffering of input and output ('read-ahead' and 'write-behind').

* Synchronous instead of asynchronous operation (each task awaits completion of each I/O command). This does not give a serious speed penalty because the I/O is buffered.

* Cataloguing of file connections under logical names with the same syntax as device names.

* 'IO-jobs' which are a special type of job, with the possibility to use the EIOS, and which automatically notify the parent IO-job if a task inside the IO-job terminates (a message segment's token is sent to a mailbox specified by the parent IO-job during creation of the 'child' IO-job, the message segment contains the reason why the task terminated).

The main reason for having an EIOS is the simplicity of the synchronous operation (most programmers are more used to 'normal' operating systems, which in most cases work this way), and the 'best effort' buffering of I/O. In some cases the 'read-ahead' and 'write-behind' buffering algorithms are not the most efficient, and better algorithms can be implemented by using the BIOS calls directly, bypassing the EIOS. Mixed use of EIOS and BIOS calls is allowed, but certain rules are to be obeyed when doing so.

There will be no problems when emulating an iRMX system containing the EIOS with an MMTCP, because the EIOS adds no new basic functions to the complete system. The EIOS only extends the functions already there, using the special extension primitives provided by the Kernel. The EIOS can be seen as one of the many possible implementations of a multiuser operating system built upon the iRMX Kernel and BIOS. Intel speaks of the EIOS as an example for a 'do-it-yourself' multiuser operating system.

**Literature**

1) Andrews, Don W. and Schultz, Gary D.:
'A Token Ring Architecture for Local Area Networks: an Update'
IBM Corporation, Communications Products Division, Research Triangle Park, North Carolina, USA.
Proceedings of the COMPCON Fall 82 Conference,
Pages 615-624.

2) ECMA 89 - Token ring standard description.

The following manuals have been used for appendix A:

Intel Corporation:
iRMX Programmers Reference Manual, Part 1
(for release 6)
Order number: 146195-001
Intel Corporation,
3065 Bowers Avenue,
Santa Clara, California 95051

Texas Instruments:
9900 Microprocessor Pascal Executive User's Manual
(Microprocessor Series)
No order number or address known.

Data for the LEX and UNIX subchapters has been found in the program listings with the kind help from my fellow students Lucien Duijkers and René Hoozemans.

**Table of MMTCP functions** (sorted alphabetically):

**Table of MMTCP functions**

**Index** (chapters 1, 2 and 3):

## A.



## B.



## C.



## D.

## E.



## F.



## G.



## H.

## I.



## L.

## M.

## N.



## O.



## P.

## Q.



## R.



## S.

## T.

## U.



## V.



## W.