MASTER

The task control unit for the multi-micro processor specification and design

Le, T.N.

Award date:
2001

Link to publication
The Task Control Unit
for the Multi-Micro Processor
Specification and Design

Ing. T.N. Le

Coach: Dr.ir. A.C. Verschueren
Supervisor: prof.ir. M.P.J. Stevens
Date: May 2001
**ABSTRACT**

Up to now, most computer systems consist of one single microprocessor and an operating system. The operating system must create a multi-tasking environment. With this processor, several multi-processor systems are built. Standard, these systems consist of a number of microprocessors and an operating system that arranges these microprocessors. The intention is executing a number of processes simultaneously. The disadvantage is that a complex operating system, like Windows NT, will be needed. For solving this problem, a single-chip called Multi-Micro processor (MµP) is designed fitted with a number of rise-alike local processing units (LPU's) and a number of shared global processing units (GPU's). Further, some functions of the standard operating system are implemented into the hardware in the MµP to arrange the LPU's. This hardware part is called the Task Control Unit (TCU). The advantage is that the operating system needed for driving the MµP is much simpler, comparing the operating system needed for a multi-processor system.

The LPU's are intended for executing of simple instructions. For executing of complex instructions, the LPU's will send the instructions to one of the GPU's via a block called Function Switch (FS). The intention of the FS is decoding the received instruction to determine for which GPU the instruction is intended. For the moment, two load-store units (LSU's), two floating-point units (FPU's) and two complex-integer arithmetic-logical units (CI-ALU's) are defined. The LSU's are used for loading and storing of data. Because loading and storing of data cost a few clock cycles, the LSU's are used so that the LPU's can execute other instructions that e.g. do not need the loaded data directly. The FPU's are used for executing of instructions that use floating points while the CI-ALU's are used for executing of arithmetic instructions. Because the TCU is also one of the GPU's, it can execute some special instructions, like getting tasks' priority. Especially, instructions for providing of both the on-core and the off-core communications. When a GPU has produced a result after the instruction has been executed, the result will be sent to a block called Result Switch (RS). In turn, the RS will decode the result to determine for which LPU that had sent instruction the result is intended.

Mainly, the TCU has to control the program and the data flow of the MµP by sending the task number (TN) of the runnable tasks to the available LPU's and executing of some special instructions. On a multi-MµP system, the TCU also has to execute instructions coming from other MµP's for providing the off-core communications. Therefore, it has to do the administration of tasks in which they can be added and or removed. In addition, the communication resources have to be administrated. It has to make continuity of tasks possible by making processors available for the execution of the tasks. It has to block the tasks temporally that cannot be executed. Finally, it has to provide the communication resources.

For designing the TCU, the TCU is divided into three main blocks called the TaskQ, the Core Unit and the Network Manager. The TaskQ is used for executing of runnable tasks by sending the tasks' TN to the available LPU's. This block is already implemented using the tool called IDaSS, which is abbreviated from Interactive Design and System Simulation. On a multi-MµP system, the block Network Manager is used for off-core communication. It is a serial communication, which protocol is similar to that of the Transputer T9000. The third block is the Core Unit, which is the main block of the TCU.

For the TCU, the term 'local request' is used, which consists of the instructions for accessing to local resource, the instructions for accessing to remote resource or a special instruction. Remotely, the request consists of the instruction for accessing to a local resource. Besides, there are four activity lines implemented on each MµP, which can be triggered by peripherals. These activity lines are called 'events'. For receiving local request, the block called "Command Receiver" is used. For receiving the remote request, the Network Manager is while a block called "Activity Detection" is used to catch the activity lines. For sending the result back to the tasks sending the local requests, a block called "Result Transmitter" is used. As last, a block called the "Execution Core" is needed, which is the most important block on the TCU. All of these blocks are implemented on the Core Unit. Because all of these blocks, except the Execution Core, are already implemented, only the Execution Core is needed to design.

The intention of the Execution Core is handling of all local requests, remote requests and events. Therefore, the controller called "Request Execution" is needed to control the program and data flow. It means that the controller will check sequentially whether there is a local request, a remote request and/or an event is present. Further, an execution register (E-register) is needed. On this register, the controller can determine whether there is a local request, a remote request or an event is present. A local request can be a request for accessing to a local resource, for accessing to a remote resource or for executing of some special instructions, like getting the tasks' priority. A remote request can only be for accessing to local resources. As last, peripherals can activate one or more activity lines, after which the attached semaphores will be executed by increasing the predefined number of units.
For execution, the controller will trigger the block “Local Resource Task Handling”, if a local request is intended for accessing to a local resource. In turn, this block will send the request to the block “Execute Resource Request” for execution. The block “Execute Resource Request” is used for executing of all requests for accessing to local resources from both the local tasks and the remote tasks. If the local request is intended for accessing to remote resource, the controller will trigger the block “Remote Resource Request Handling” so that the request can be handled by sending the request to the remote MμP handling the resource via the Network Manager. For executing of special instructions, the controller will trigger the block “Decode & Execute Instruction”. If a result is produced, like after a write-operation, the result will be placed onto the result register (R-register) of the block “Result Transmitter” to send back to the local task sending the request via the Result Switch. After the request is executed completely, the task sending the request will be restarted by setting its field “Runnable” on the TaskQ to 1b and setting its state on the task list (TL) to tsRTR for updating the TL. In addition, time slicing of running and runnable tasks, which have the same priority, will be handled by the block Execution Core.

Further, in a multi-MμP system, remote requests will be received by the block “Message Handling”, after which it will be stored onto the remote-request-task list (RRTL) until the request is executed completely or until the request is cancelled. For checking the liveliness of waiting local tasks, this will be executed by the block “Local Tasks’ Liveliness Checking”. Because different messages can be sent onto the network for communicating with other MμP’s, the block “Local Tasks’ Message Timeout” is needed for handling the local tasks’ message timeout and the block “Remote Tasks’ Message Timeout” is needed for handling the remote tasks’ message timeout.

As last, for handling the remote requests, the block “Remote Request Handling” will send a request on the RRTL to the block “Execute Resource Request” for execution. If a result is produced, a remote-wait-result (RWR) message including the result will be sent back to the remote task via the Network Manager.

For the task administration, the TL on the TCU keeps the administration of 63 local tasks instead of 64, because the last task is the NIL-task. The NIL-task does nothing and has the lowest priority. The local tasks have 16 levels of priority; level 0 is the highest priority while level 15 is the lowest priority. Because all tasks on remote MμP’s are allowed to access to local resources, a four words per block remote-request-task list (RRTL), a double linked list, is used for keeping the administration for all remote requests until they are executed completely or until a request is cancelled by the concerning or another remote task. Because the number of remote requests is not known beforehand, the RRTL will be kept in the main memory, in a dynamic structure. To save the space, the number of the task blocks (= number of remote requests) is limited by 16K, which will be indexed by 14 bits to allow maximum $2^{14} = 16K$ remote tasks to wait for local resources. The last 64 tasks are mapped to the local tasks, allowing a maximum of $(16K - 64) = 16128$ remote tasks waiting for local resources.

For providing both the on-core and the off-core communications, communication resources can be created and removed. These resources can be semaphore, mailboxes and or pipes. The semaphores are used for general purposes, the mailboxes are used for exchanging of 32-bits messages and the pipes are used for transferring of block data. For the moment, only 128 resources can be created and removed that have to be administrated on the TCU of each MμP. The number of semaphores, mailboxes and pipes are not fixed. As long as 128 resources are not created or there are enough memory left, these resources can be created.
Table of contents

ABSTRACTS

1. INTRODUCTION........................................................................................................... 1

2. ANALYZING THE NUCLEUS.......................................................................................... 4
  2.1 Task states ................................................................................................................. 4
  2.1.1 Time-slicing of tasks ............................................................................................ 5
  2.2 Task administration ................................................................................................. 6
  2.3 The Dispatcher ......................................................................................................... 7
  2.4 The interrupt handling ............................................................................................. 7
  2.5 Analyzing the communication resources .................................................................. 7
  2.5.1 Semaphores ......................................................................................................... 7
  2.5.2 Mailboxes ............................................................................................................ 8
  2.5.3 Pipes ..................................................................................................................... 8

3. SPECIFICATION OF THE TASK CONTROL UNIT ...................................................... 9
  3.1 Principle of a MultiP system ..................................................................................... 9
  3.1.1 Functions of the Task Control Unit ...................................................................... 10
  3.1.2 Constraints of the Task Control Unit .................................................................. 10
  3.2 State transitions of a task ....................................................................................... 11
  3.3 The Core Unit ........................................................................................................... 12
    3.3.1 Time generation .................................................................................................. 12
    3.3.2 Tasks administration ......................................................................................... 12
    3.3.3 Communication resources ............................................................................... 12
      3.3.3.1 Semaphore .................................................................................................... 12
      3.3.3.2 Mailbox ........................................................................................................ 12
      3.3.3.3 Pipe .............................................................................................................. 13
      3.3.3.4 Communication resource administration .................................................... 13
  3.4 The TaskQ ................................................................................................................. 13
    3.4.1 Time-slicing of tasks ......................................................................................... 13
  3.5 The Network Manager ............................................................................................. 13
  3.6 RAM of the Task Control Unit ............................................................................... 14
  3.7 Basic instructions of the Task Control Unit ............................................................ 14

4. DESIGNING THE TASK CONTROL UNIT .................................................................. 15
  4.1 Designing the Execution Core .................................................................................. 16
  4.2 Request Execution .................................................................................................... 16
  4.3 Defining some registers and lists ............................................................................ 19
    4.3.1 E-register ........................................................................................................... 19
    4.3.2 Status word ....................................................................................................... 20
    4.3.3 Resource administration .................................................................................... 23
    4.3.4 Local task administration ................................................................................. 25
    4.3.5 Remote task administration .............................................................................. 26
    4.3.6 Local resource waiting list ................................................................................ 28
    4.3.7 C-register .......................................................................................................... 29
    4.3.8 A-register .......................................................................................................... 29
    4.3.9 RI-register ......................................................................................................... 30
    4.3.10 R-register ........................................................................................................... 30
  4.4 Local Request Handling ............................................................................................ 31
    4.4.1 Sending result back to local task ....................................................................... 32
    4.4.2 Waking up a local task ...................................................................................... 32
    4.4.3 Waking up a certain local task .......................................................................... 33
  4.5 Decode & Execute Instruction .................................................................................. 34
    4.5.1 NOP ..................................................................................................................... 34
    4.5.2 CREATE_TASK ................................................................................................. 34
    4.5.3 END_TASK ........................................................................................................ 36
      4.5.3.1 Removing a task from the local resource waiting list .................................. 37
4.5.3.2 Removing a task from the timeout waiting list ........................................... 39
4.5.3.3 Removing a task from the TaskQ/TL ......................................................... 39
4.5.4 CHANGE_PRIO ................................................................................................. 39
4.5.5 GET_PRIO ........................................................................................................ 40
4.5.6 SLEEP .............................................................................................................. 40
4.5.7 AWAKE ........................................................................................................... 41
4.5.8 DELAY ............................................................................................................. 41
4.5.8.1 Inserting a task into the timeout waiting list .............................................. 42
4.5.8.2 TIMEOUT ..................................................................................................... 45
4.5.9 CREATE_RESOURCE .......................................................................................... 45
4.5.9.1 Create Semaphore ....................................................................................... 46
4.5.9.2 Checking the availability of the required resource data blocks ................. 47
4.5.9.3 Checking the previous resources ................................................................. 49
4.5.9.4 Checking the required resource data blocks .............................................. 50
4.5.9.5 Restarting waiting tasks ............................................................................. 51
4.5.9.6 Create Mailbox ............................................................................................ 52
4.5.9.7 Create Pipe ................................................................................................ 53
4.5.10 Removing a resource .................................................................................... 54
4.5.11 Writing to a resource .................................................................................... 54
4.5.12 Reading from a resource ............................................................................... 55
4.5.13 Execute Resource Request ........................................................................... 56
4.5.13.1 Sending result back to a remote task ....................................................... 57
4.5.14 Execute Mailbox ............................................................................................ 57
4.5.14.1 Block Task ................................................................................................ 59
4.5.14.2 Restart Task ............................................................................................... 60
4.5.14.3 Execute Reading Task ............................................................................... 61
4.5.14.4 Wake Up Writing task ............................................................................. 62
4.5.14.5 Execute Writing Task ............................................................................... 64
4.5.14.6 Wake Up Reading Task ............................................................................ 64
4.5.15 Execute Pipe .................................................................................................. 65
4.5.16 Execute Semaphore ...................................................................................... 65
4.5.16.1 Behavior description of "Execute Reading Task" .................................... 66
4.5.16.2 Behavior description of "Execute Writing Task" .................................... 68
4.6 Local Resource Request Handling .................................................................... 69
4.6.1 The behavior description ................................................................................ 69
4.6.2 Inserting task into resource waiting list ....................................................... 70
4.6.3 Placing ocal request onto RI-register ............................................................. 71
4.7 Remote Resource Request Handling ............................................................... 71
4.7.1 Principle of message timeout ........................................................................ 73
4.8 Delaying Task Handling .................................................................................... 73
4.9 Time slicing ....................................................................................................... 74

5. REMOTE REQUEST HANDLING .......................................................................... 76

6. EVENTS HANDLING ............................................................................................ 78

7. MESSAGE HANDLING ......................................................................................... 79
7.1 Process RWC-message ...................................................................................... 80
7.2 Process CIA-message ......................................................................................... 81
7.3 Process CW-message ......................................................................................... 81
7.4 Process RA-messages ....................................................................................... 82
7.5 Process RWR-message ...................................................................................... 83
7.6 Process PWL-message ...................................................................................... 84
7.7 Process EWA-message ....................................................................................... 85

8. LOCAL TASKS' LIVELINESS CHECKING ............................................................ 87
8.1 Behavior description .......................................................................................... 87

9. LOCAL TASKS' MESSAGE TIMEOUT HANDLING ............................................. 89

10. REMOTE TASKS' MESSAGE TIMEOUT HANDLING ........................................ 91
10.1 Removing a task from the RRTL ..................................................................... 92
1. INTRODUCTION

Up to now, there are several multi-processor systems built with multitasking facilities, like a PC with a number of Pentium-processors or other processors which can be driven by e.g. an operating system like Windows NT. The disadvantage is that a complex operating system, like Windows NT, will be needed. For solving this problem, a single-chip called Multi-Micro processor (MμP) is developed fitted with a number of risc-alike local processing units (LPU's) and a number of shared global processing units (GPU's). In figure 1a [Loo], the global architecture of the MμP is shown.

![Global architecture of the MμP.](image)

For the moment, the MμP contains four identical risc-alike LPU's for executing simple instructions. They all have their own instruction pointer (IP). They are fully operational processors. It means, they are able to get instructions directly from the Instruction Cache Unit (ICU). When the fetched instruction is complex and the LPU is not able to execute the instruction, it will send the instruction through to the Function Switch (FS). In turn, the FS will decode the instruction and sends it to the GPU that can execute the instruction. The GPU's are intended for executing of complex instructions. They contain only Executing Modules (EM's). Examples of the GPU's can be the Floating Point Units (FPU's), the Load-Store Units (LSU's) and the Complex Integer Arithmetic Logical Units (CI-ALU's), which are capable to execute one instruction at a time that comes from the FS. Because the TCU is one of the GPU's, it can also execute of some special instructions. Also for accessing to resources, the FS can send the instruction to the TCU. After the instruction has been executed, the result is sent back to the Result Switch (RS). After decoding the result, the RS will send it to the corresponding LPU that had sent the instruction.

The LPU is capable to fetch instruction directly from the Instruction Cache, but it is not capable to fetch data directly from the Data Cache. Therefore, the LSU is needed. Because fetching data costs several clocks, the LPU can use the LSU so that it can continue executing the instructions that do not need the loaded data directly, while loading the data. For storing the data, it can use the LSU while continuing executing other instructions without concerning whether the data is written.

Each LPU is capable to execute a single task or to do nothing. It is possible to run one single program (task) on multiple LPU's at the same time. Therefore, a GPU called the TCU is needed for controlling. The TCU will also control the LPU's for task switching and the communications between different tasks. Task switching means that a LPU is executing a task and that the TCU assigns another task to it, while the first task has not been executed completely. Therefore, the Register Cache Unit (RCU) is needed for storing all information needed of the first task, which after the LPU will load the registers of the second task. A more specific overview of the MμP is shown in figure 1b [Loo] on the next page.
A standard computer system consists of an application layer, an operating system layer, a nucleus layer and a hardware layer. In figure 1c, these layers are shown for the standard computer system and for the MμP system.

The purpose of the nucleus layer is to create an environment in which parallel tasks can exist, regardless of the number of available processors in the system. This layer is normally implemented completely in software and has several functions. The idea of the MμP is to implement some functions of the nucleus, which are time-consuming, in hardware to control the program and the data flow of the MμP. The advantage is that because of the implementation in hardware the system will be faster than in software. Besides, the MμP will need a simpler operating system comparing with a multi-processor system.

For the moment, the basic concept of the MμP is already on paper [Theelen]. Several units of the MμP are already implemented in IDaSS [Verschueren]. IDaSS is an inactive design and simulation environment for digital circuits, which describes a design as a tree-like hierarchy of schematics. The TCU is the most important part of the MμP.

Up to now, there are two students, Silvia Nedea and Tom van Loo, who had contributed for designing the TCU. However, their designs are still stray ideas, which are not implemented in IDaSS and or tested.
To get a concrete design, a specification of the TCU will be formulated and the design will be made.

This is the main goal of this final project. Because of time shortage the implementation using the tool IDaSS will not be done. But, the design of the MµP will be described comprehensively so that the design can be implemented easily in the future.

The purpose of this graduation work is making some studies of the principle of the nucleus and the design works of the students Silvia Nedea and Tom van Loo. This is described in chapter 2. The aim of this chapter is, understanding the principle of the nucleus of the standard operating system. Some analyses will be done so that some decisions can be made which functions of the nucleus will still be implemented in software and which functions are recommended to implement in hardware. After this, in chapter 3 the specification for the TCU will be formulated so that the functions of the TCU can be established concretely. After that, the design of the TCU will be made. This is described in chapter 4 up to chapter 11.

The intention of this graduation work is not to make a completely new design for the TCU, but a good and worked-well design of the TCU has to be developed. So, when ideas and or designs of the works of these two students are usefully, they will be used in the new design.
2. ANALYZING THE NUCLEUS

In this chapter, several important functions of the nucleus layer of the standard operating system will be analyzed. With this information, a decision can be made which functions will be implemented in the hardware and which functions will remain in the software. On the Multi-P, the hardware part of these functions will be called the ‘Task Control Unit (TCU)’. So, the specification of the TCU will be formulated in chapter 3. The aim of this work is to improve the performance of the single-MultiP system comparing with a multi-processor system: The complexity of the nucleus (software part) will be reduced and the speed of the system will be increased.

The TCU is a co-processor, one of the Global Processing Units (GPU), which partially implements the nucleus layer of the computer system in hardware instead of completely in software. Only functions, which are time-consuming will be implemented in hardware. The purpose of this layer is to create an environment in which parallel tasks can exist, regardless of the number of available processors in the system.

The functions that the standard nucleus [Geurts] has to perform can be summarized as:

- Doing the administration of tasks in which they can be added and or removed,
- Making continuity of tasks possible by making processors available for the execution,
- Temporary blocking of tasks that can not execute,
- Provide mechanisms for internal communication between tasks, and
- Provide mechanisms for communication with external tasks in a multi-MultiP system.

In literature, the terms ‘process’ and ‘thread’ are used. In the nucleus, a process can create and destroy a subprocess called ‘thread’ or ‘lightweight process’. Threads will have the same properties as the mother process. Because of the low time-consuming of this function and to keep the TCU less complex, it is no need to implement this function in hardware. This means that the TCU will only receive elementary processes or threads, which will be called ‘tasks’ in this design. Therefore, a task will not be allowed to create another task normally. However, a small amount of the present tasks will be allowed to create another task.

In the next sections, the functions of the standard nucleus are analyzed.

2.1 Task states

To a task, the states RUNNING and BLOCKED can be assigned. A task is blocked when it has to wait for something, which is not available yet. Or, when it is put asleep conditionally or unconditionally. With a certain action or event a task can be put in or out this state. A task is running when it is runnable and a processor is available. These task's states are depicted in figure 2.1a.

When a task from a BLOCKED-state will go over to a RUNNING-state, it needs a processor. However, a processor is not always available. So, a READY-TO-RUN state will be assigned to this task. The task's states of figure 2.1a are expanded and are shown in figure 2.1b on the next page.
When administrating the tasks, it will be usefully when a wait list is used for each state. For switching a task from the state READY-TO-RUN to the state RUNNING, a component called Dispatcher will be needed. Each time that a processor is available, the Dispatcher will be activated. On the MuP, the Dispatcher is called the TaskQ. When there is more than one task with the state READY-TO-RUN waiting for execution, a scheduler will be needed to sort and chain them, using their priorities to decide which task can be executed firstly. A scheduler will also be needed for the tasks with the state BLOCKED, which want go over to the state READY-TO-RUN.

2.1.1 Time-slicing of tasks

When there are more tasks present in the waiting list than the number of processors (LPU’s) is available with the state READY-TO-RUN and with the same priority, then the first four tasks will be executed. After fixed time duration, these running tasks will be blocked temporally and put back in the waiting list, just behind the other waiting tasks. After that, the first four tasks in the waiting list will be executed. This mechanism is called “time-slicing” and is shown in figure 2.1.1.

With this mechanism, it looks like that all tasks with the same priority will be executed simultaneously while there are less processors available.
2.2 Task administration

For the TCU, the task administration will be kept with a list of task descriptors (TD). It describes the states of a task in the following data structure [Geurts]. In figure 2.2a, the data structure of the TD is shown using the C-language.

```c
typedef struct {
    List_TD;        // List of task descriptors (TD) of all available tasks
    *Set_TD;       // Number of lists to keep all tasks with the same status.
    TN;            // Task Number (TN) used as identifier of the task and will be used by tasks communication
    TaskStatus;    // Status of the tasks with the following states:
                    // {BLOCKED, RUNNING}
    Priority;      // Priority of the tasks
} TaskDescriptor;
```

Figure 2.2a. Data structure of the Task Descriptor.

For the standard nucleus, some memory space will be reserved to store a collection of facilities of a processor that is accessible to a task. However, for the TCU, the Register Cache Unit will take care of it. To keep a convenient arrangement, this data structure will be addressed via a so-called 'central table'. This special reserved memory space makes all system data accessible. It can be e.g. the version of the operating system, date and time, all present tasks arranged in a task structure, all present resources, memory allocation, etc... An example of memory organization of the central table is shown in figure 2.2b.

![Figure 2.2b. Memory organization of all tasks in the central table.](image)

This table is depicted using the C-language. Keep in mind that this table is implemented in hardware completely on the TCU. In the central table, there is a pointer *TD, which refers to a list List_TD of all present Task Descriptors somewhere in the memory in which all tasks are described and or organized. Therefore, when a task has been executed completely, it will be removed easily from the list. In case of a new task, it will be inserted into the list. From this list the tasks will be sorted using their states and arranged in a linked list as *Set_TD. The advantage of a linked structure is the ease to change the order of the waiting tasks, using their priority, in a certain chain. After reordering the tasks, the tasks that have e.g. the state READY-TO-RUN with the highest priority will be put in front of the waiting list so that the first task in the chain will be handled first. Subsequently, the tasks with the state BLOCKED will be sorted and chained. Because the Local Processing Unit (LPU) is executing instructions continually, a NIL-task with the lowest priority is inserted at the end of the linked list. With this task, a LPU can be put in an IDLE-state. Another advantage of a linked structure is the ease to insert a task into or to delete a task from the structure. Consequently, two instructions like INSERT_TASK and DELETE_TASK will be needed.
2.3 The Dispatcher

The purpose of the Dispatcher is assigning the available processors to the tasks with the state READY-TO-RUN. When a processor is available, the Dispatcher will be activated and decides which task will be executed.

The Dispatcher of the standard nucleus works as follow:
- Sending the new process to an available processor,
- Saving the volatile environment of the current process, which is released by the processor,
- Restoring the volatile environment of the new process from the READY-TO-RUN waiting list,
- Starting the new process.

Normally, these functions are implemented in the nucleus in software. On the MµP, they are implemented partially in the local processor (LPU), the Register Cache and partially in the TCU. The process is called the 'task' and the Dispatcher is called the 'TaskQ'. An implemented scheduler in the TaskQ will reorder the present tasks according to their priorities. When a processor is available, the TaskQ will assign a task with the highest priority by sending its task number to the processor. This processor will save the volatile environment of the current task into the Register Cache Unit, before loading (restoring) the volatile environment of the new task. After the new task has been loaded, the processor will start this task. Therefore, as function of the TaskQ, the TCU has to check the availability of the processors (LPU's) continually and sends a task number to the available processor.

When there are no more tasks that have to be executed, suppose all tasks are blocked, a NIL-task is sent to the concerning LPU to put it into the IDLE-state.

2.4 The interrupt handling

There are two possibilities for implementing interrupts into a computer system. One interrupt line for all peripherals or an interrupt line for each peripheral. The last one will be used restrictedly in small computer systems. In computer system with a powerful microprocessor, number of pins of the processor will be increased rapidly. However, on the MµP, no interrupts are used. There are four input lines reserved and implemented for activity detection. For each activity line, the TCU needs only to know which semaphore should get how many units on activity detection.

2.5 Analyzing the communication resources

For the moment, there are four LPU's implemented. This number can be expanded or reduced arbitrary in the future. This means, a number of tasks can be parallel executed which resulting in synchronization, sharing a communication resource or communication between tasks. Synchronization is needed when a task has to wait e.g. for the result of the other task. When there is only one resource available, different tasks have to share the available resource together. Sharing a resource is very important in multitasking environment because of efficiency and need. For efficiency purposes, it is useless to make a copy of the resource for each task. When different tasks have to cooperate, they need to use data or results commonality. The communication will be needed when two or more tasks need to exchange data or results with each other. For this purpose, semaphores, mailboxes and or pipes can be used.

2.5.1 Semaphores

Semaphore S is an integer variable that, apart from initialization, is accessed only through two primitives: P and V. The primitive P will be used for testing and the primitive V will be used for incrementing the integer value of the semaphore S. In general, these primitives are defined as follows:
Modifications to the integer value of the semaphore S with these primitives must be indivisibly. It means, when one task modifies the semaphore value, no other tasks can modify simultaneously the same semaphore value.

There are two kinds of semaphores: A counting and a binary semaphore. The binary semaphore will be used for controlling for the tasks like in the one-to-one model while the counting semaphore will be used for multiple tasks in the case of one-to-many or many-to-many models. When there are a limited number of resources, counting semaphores can be used to control the access. If the integer value of the semaphore is '0', all resources are used. So, tasks that wish to use a resource will be blocked until the semaphore value becomes greater than '0'.

On the TCU, the semaphores are used for general synchronization purposes. Because the counting semaphore can also be used as a binary semaphore, only counting semaphores will be used. In addition, semaphores can be used for communication between different CPUs in a multi-MuP system. In this case, a local network between the CPUs will be needed, which arrangement the communication.

2.5.2 Mailboxes

When two or more tasks want to communicate indirectly with each other, a mailbox can be used. It is based on the producer-consumer concept with a buffer. A mailbox can be viewed abstractly as an object into which messages can be placed (produced) by tasks and from which messages can be removed (consumed). It has a shared data structure and consists of a header, which describes the mailbox, and a number of slots into which messages can be placed and from which messages can be removed. Each mailbox has a unique identification. A task can communicate with other tasks via a number of different mailboxes. In addition, the possibility to let a task communicate with a task in another MuP has to be built in. Mailbox is one of the resources to let two tasks communicate with each other.

In general, for a mailbox the primitives send() and receive() are defined as follows:

- send_msg (A, message) // For sending a message to mailbox A
- receive_msg (A, message) // For receiving a message from mailbox A

With these primitives, a task can send and receive messages. When a message is received, it will be deleted by the concerning mailbox automatically. It is fulfilled for the TCU to use these two primitives for realizing the communication resource mailbox. When a mailbox is full, no new messages can be sent. In addition, when a mailbox is empty, no messages can be received. In both cases, the task attempts to access to the mailbox will be blocked until the mailbox is not full or until it is not empty, respectively.

2.5.3 Pipes

Pipe is a special form of mailbox. Essentially, it is a queue of bytes between tasks. It permits a reliable unidirectional communication between two tasks. Contrary to a mailbox, the pipe has only one slot in which blocks data can be stored and from which blocks data can be consumed. The pipe has a fixed size. It will be used for transferring of block data within an MuP and or between different MuP's. One task will produce a stream of bytes and send it to the pipe. The other can consume this stream in same direction as produced according to the FIFO-principle. In addition, a task can be blocked when the sending task attempts to write a block data to a full pipe or the receiving task attempts to read the block data from an empty pipe.
3. SPECIFICATION OF THE TASK CONTROL UNIT

In chapter 2 "Analyzing the nucleus", several important functions of the nucleus are analyzed. Some of them will still be implemented in software because of complexity or lack of flexibility. Only basic functions, which are time-consuming will be implemented in hardware in one of the Global Processing Units (GPU's) called the "Task Control Unit (TCU)". It means there will be still a nucleus (software part) needed for an MµP system. Only, this nucleus will be much simpler comparing with multi-processors systems. In this chapter, the specification of the TCU will be formulated in a top-down methodology using the information of chapter 2. Firstly, the principle of the MµP system will be described generally in section 3.1. The intention is to get a global picture of the system so that the importance of the global processing unit TCU can be understood. Before the specification of several functions of the TCU will be formulated in section 3.3 to section 3.7, the possible state transitions of a task will be described in section 3.2.

One of the most important aims for designing the MµP is splitting up the standard nucleus layer into a hardware and a software part as shown in figure 1c on page 2. Therefore, the term “nucleus” will be used for the software part and the term “TCU” or “Task Control Unit” will be used for the hardware part.

3.1 Principle of a MµP system

When switching on the power, the MµP-chip will be reset (power on reset). Subsequently, the Boot-Up task will be started and executed by one of the four Local Processing Units (LPU's), which is sent by the TCU. Boot-Up task is the first task on the task list. As result, a nucleus is formed, which is a software layer between the hardware layer and the operating system layer as shown in figure 1c on page 2.

Further, some initializations of hardware and/or of peripherals will be taken place. After that, the nucleus will wait for new instructions or tasks from the operating system layer. In turn, the TCU will wait for new tasks from the nucleus.

In general, the nucleus will create and destroy processes. Subsequently, these processes can create other processes, called subprocesses or lightweight processes also called threads. For the TCU, the thread will be called "task", which is officially not allowed to create other tasks. However, sometimes it is needed to allow a task to create other tasks.

On the TCU, there is a task list, which keeps the administration of up to 64 tasks numbering from 0 to 63 with 16 levels of priority. The priority number 0 denotes the highest priority while the priority number 15 denotes the lowest priority. The first task number (TN) 0 is reserved for the Boot-Up task while the last task number 63 is reserved for the NIL-task. Consequently, only 62 tasks can be used for general purposes. The NIL-task has always the lowest priority and the state READY-TO-RUN. When there are no more tasks present in the task list, the TCU will send the NIL-task to the available processors to put them into the IDLE-state or turning them off.

When the nucleus creates a new task, the task number will be put on the task list while the content (instructions and/or data) of this task will be stored in the main memory temporarily. On the TCU, a scheduler will be implemented to sort and chain all of the present tasks according to their priority. When a processor is available, the TCU will assign this processor to a task with the highest priority and with the state READY-TO-RUN by sending the concerning task number to the processor. After the processor has loaded the registers of the new task, it will fetch the instruction code (opcode) from the instruction cache called ‘ICache’ and the operand(s) from the data cache called ‘DCache’ using the Load/Store Unit (LSU), which is also one of the GPU’s. Therefore, the opcode and the operand(s) are fetched from the main memory.

To be able to keep up to 64 tasks with 16 levels of priority, the TCU has to be fitted with several lists and/or queues for administrating the tasks with the same state, like READY-TO-RUN or BLOCKED. When there are more tasks with the state ‘READY-TO-RUN’ and with equal priority than the number of processors (LPU's) is available, time-slicing mechanism will be used. Now, the MµP is equipped with four processors. It means the first four tasks will be executed for a fixed time. After that, they will be put back on the wait list so that the next four tasks or less can be executed. This procedure will be repeated until all of these tasks are executed completely. With this mechanism, it seems that all tasks are executed simultaneously.

So far, the LPU operates in the NORMAL mode. The other mode is the SWITCHING mode. In this mode, the LPU will save the volatile environment of the current executing task into the Register Cache before loading the
volatile environment of the new task when the TCU sends a new task number. However, if the new task number is equal to the current task number, the LPU will stay in the NORMAL mode.

### 3.1.1 Functions of the Task Control Unit

To fulfill the constraints of the MuP, the TCU has to be fitted with the following functions:
- Doing the administration of tasks in which they can be added and or removed. In addition, communication resources have to be administrated.
- Making continuity of tasks possible by making processors available for the execution of the tasks.
- Temporary blocking of tasks that can not execute.
- Providing communication resources.

For the design, these functions will be implemented in the following units:
- Core Unit,
- The TaskQ, and
- Network Manager

The interfaces of these units are depicted in figure 3.1.1

![Figure 3.1.1. Interfaces of different units on the TCU.](image)

Most of the major operations will take place in the Core Unit. The TaskQ and the Network Manager are used as a kind of inputs and/or outputs for the Core Unit.

In the following sections, the specification of these units will be formulated.

### 3.1.2 Constraints of the Task Control Unit

For designing the TCU, the following constraints have to be fulfilled:
- Four local processors (LPU’s) have to be controlled by sending task numbers (identifier) to them.
- Up to 64 local tasks, numbering from 0 to 63, have to be administrated. Task with the task number 0 is the Boot-Up task and task with the task number 63 is the NIL-task.
- Each task has a priority. There are 16 levels of priority, numbering from 0 (highest) to 15 (lowest). Selection of the tasks is based on this priority. The TCU can execute instruction to change the priority.
- Up to 16K – 64 remote tasks have to be administrated. Because the number of tasks is not known beforehand, the list will be kept in the main memory.
- Time-slicing mechanism has to be implemented for the case when there are more runnable and running tasks with equal priority than the number of processors (LPU’s) is available.
- Up to 128 communication resources has to be administrated, like mailboxes, pipes and semaphores.
  - Short and long counting semaphores have to be implemented.
  - Mailboxes have a many-to-many model; only the primitives `read` (receive-operation) and `write` (send-operation) will be implemented; the messages are 32 bits in width; the number of messages
is variable and is equal to $2^N$. No long mailboxes will be used. This is an option for the future work.

- Pipes are used for block data transferring and have a many-to-many model; only the primitives `read` (receive-operation) and `write` (send-operation) will be implemented; Buffer size for the data blocks is variable. Both the data as the buffer have to be kept in the main memory.
- Four activity lines, a kind of interrupt, have to be administrated. No Interrupt Handlers are needed and tasks waiting for these lines will be treated as NORMAL tasks.
- Timer/timeout has to be implemented. This will be needed when a task needs a timeout or for time-slicing purposes. In addition, for the message timeout, a timer is needed.
- A network protocol for communicating with other MµP's in a multi-MµP system.

### 3.2 State transitions of a task

At operating system level, there are three basic states through which an executing task has to be passed: CREATION, EXECUTION and TERMINATION. First, a new task will be created by the nucleus and sent to the TCU. Next, this task will be executed by one of the available local processor and arranged by the TCU, after which a result can be produced. When executing a task, the state EXECUTION has to split up into three sub-states RUNNING, RUNNABLE and BLOCKED, as depicted in figure 2.1b on page 5, to be able to administrate all present tasks. But, to fulfill the constraints of the TCU, the state BLOCKED (or WAITING) has to split up into seven other sub-states as shown in figure 3.2.

<table>
<thead>
<tr>
<th>Sub-state</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL</td>
<td>Sleep; Putting a task asleep unconditionally</td>
</tr>
<tr>
<td>WTO</td>
<td>Waiting for timeout; A task puts itself or another task asleep for the given timeout time</td>
</tr>
<tr>
<td>WLR</td>
<td>Waiting for local resource</td>
</tr>
<tr>
<td>WCA</td>
<td>Waiting for command acknowledgement</td>
</tr>
<tr>
<td>WRR</td>
<td>Waiting for remote resource</td>
</tr>
<tr>
<td>WEWA</td>
<td>Waiting for end wait acknowledgement</td>
</tr>
<tr>
<td>CRL</td>
<td>Checking resource liveliness</td>
</tr>
</tbody>
</table>

*Figure 3.2. Splitting up the BLOCKED-state into seven sub-states.*

It is possible for a task to put itself or another task asleep unconditionally. For example, when a task has to wait for the result of the other task, it will be put asleep temporarily. It can only be wake up by other task. In this case, it can only be awoken when the result is produced. Sometimes, it is necessary that a task needs to time out for a while. After the requested timeout time is reached, the concerning task will be wake up unless it is forced wake up by another task. When a task attempts to access to a resource and it is not available yet, this task will be blocked until the requested resource is available or until it is forced wake up by another task.

In a multi-MµP system, a task is capable to access to resources on the other MµP's, the remote resources. For accessing, a message including the request is sent onto the network (via the Network Manager) towards the remote MµP handling the resource, which after the task's state is changed to waiting-for-command-acknowledgement (WCA). If the requested resource is not available yet, the request is placed on the remote waiting list. Therefore, the task's state is changed to waiting-for-remote-resource (WRR). So, the task will be blocked until the request has been executed completely or until the request is cancelled. Or until the task is forced wake up. When canceling the request, the task's state is changed to waiting-for-end-wait-acknowledgement (WEWA) after the cancel-message is sent onto the network. The task sending the request will be wake up by changing its state to ready-to-run when the end-wait-acknowledgement message has been received from the MµP handling the resource. As last, when the request is placed onto the remote waiting list the remote resource liveliness must be checked periodically. Therefore, the task's state is changed to checking-resource-liveliness (CRL).
3.3 The Core Unit

The Core Unit is the most important unit of the TCU. Outside the TCU, it has the Function Switch and the activity lines as external input ports while the Result Switch forms the external output port. Via the Function Switch, the Core Unit will receive instructions, which are intended for the TCU. For communicating with other MμP's, the Network Manager is needed. When the communication resources pipes and/or large mailboxes are used, main memory is needed for storing and/or loading of data via the DCache. When the TCU produces result, the result will be output to the Result Switch.

3.3.1 Time generation

The Core Unit has to generate time for time-slicing purposes and for tasks, which need a certain timeout. Also for the communication with other MμP's, a message timeout is needed.

3.3.2 Tasks administration

On the TCU, the Core unit has to administrate 16K tasks totally, 64 local tasks and up to 16K – 64 remote tasks. For administrating the local tasks, a fixed task list called TL is needed and will be kept on the TCU itself. Because the number of remote tasks is unknown beforehand, the information of these tasks is kept in the main memory, in a dynamic data structure. Therefore, a double linked list is needed. For keeping the remote tasks, a task list called remote-request-task list (RRTL) is needed.

3.3.3 Communication resources

For communication between tasks, several communication resources have to be implemented. During operation, these resources can be created and removed by the TCU. On multi-MμP systems, the communication between tasks on different MμP's must be as easy as between tasks on the same MμP. For these purposes, the communication resources semaphores, mailboxes and pipes are used. For each MμP, the TCU has to manage up to 128 resources, which means that the number of mailboxes, pipes and/or semaphores is unknown. Together, it must be at most 128 resources. As long as this number is not reached and there is still RAM available, a new resource can be created.

3.3.3.1 Semaphore

For the MμP, semaphores are used for general synchronization purposes, possibly in combination with activity lines. In the last case, the TCU needs only to know which semaphore should get how many units on activity detection.

The communication between different MμP's must be as easy as the communication between different tasks within an MμP. The administration must be kept on the TCU completely. As long as the limit of 128 resources is not expired and there is still RAM available, semaphores can be created and removed by the TCU.

3.3.3.2 Mailbox

Mailboxes are used for indirect communication between different local and or remote tasks. For each mailbox, a many-to-many model is used. Therefore, the number of accesses is not restricted. The number of slots or messages for each mailbox is variable depending on the request of the user. However, this number must be the power of 2: $2^N$ while N is a natural number. The width of the message is 32 bits. Only the write-operation (the primitive send()) and the read-operation (the primitive receive()) are used. With these operations, mailboxes can be written and read with or without a condition. When a receiver wants to send back an acknowledgement after reading a message, the receiver has to send (a write-operation) a new message as acknowledgement to the...
sender. It means the primitive reply() will not be used. In addition, the long mailbox will not be used in this work. Long mailbox is a mailbox with more than 16 slots. This is an option in the future work.

As long as the limit of 128 resources is not reached and there is still RAM available, mailboxes can be created.

3.3.3.3 Pipe

For transferring block data within an MIP or between different MIP's, pipes are used. The administration will be kept on the TCU while the data is kept in the main memory. It means a pointer is needed to refer to the concerning block data.

For the pipes, a many-to-many model will be used, like for mailboxes. Therefore, the number of accesses is not restricted. The number of pipes is not fixed depended on the number of created resources, with a maximum of 128 resources. The pipes' buffer size is variable, but the data block must be fit in the buffer.

As long as the limit of 128 resources is not reached and RAM is still available, pipes can be created. Like semaphores and mailboxes, pipes can be read and written with or without a condition.

3.3.3.4 Communication resource administration

On the MIP, different communication resources, like semaphores, mailboxes and pipes are used for internal and external communications. The TCU of each MIP must be able to keep administration of up to 128 resources. The built-in RAM must be chosen big enough for 128 resources and for other purposes. So, when all 128 resources are created or there is no more RAM is available, requesting for creating a new resource will be rejected.

Because both the local and the remote tasks are allowed to access to local resources, waiting lists for the concerning resources must be administrated.

3.4 The TaskQ

The main goal of the TaskQ is assigning available processors to the runnable tasks with the state READY-TO-RUN. Therefore, the TaskQ has to check continually whether there is a processor available. For the moment, four local processors have to be controlled. From the Core Unit, the TaskQ will only receive tasks with the state READY-TO-RUN. An embedded scheduler will sort and chain these tasks according to their priorities.

3.4.1 Time-slicing of tasks

When there are more running and runnable tasks with equal priority than the number of processors is available, time-slicing mechanism has to be applied so that these tasks can be executed simultaneously as depicted in figure 2.1.1 on page 5.

3.5 The Network Manager

The TCU of each MIP has to equip with a Network Manager. This is a serial communication unit, intended for communication between different MIP's. For the moment, the Network Manager is fitted with four serial links. This number is chosen arbitrary and can be changed for the needs in the future. Therefore, the Network Manager has to be fitted with a communication protocol. Because it is a serial communication, the communication protocol similar to that of the Transputer T9000 is used.
3.6 RAM of the Task Control Unit

For keeping administration of several lists and queues, e.g. for tasks and communication resources, the TCU has to be fitted with RAM. In addition, for temporarily storing of information or for other purposes, RAM will be needed. Because pipes are used for transferring of block data, the main memory must be used. Accessing to this memory must be taken place via the DCache Unit. In contrast with the local processors, no Load/Store Units (LSU’s) will be used for storing and loading of data. The RAM has to be chosen big enough so that the TCU can function efficiently.

3.7 Basic instructions of the Task Control Unit

As mentioned in chapter 2, the TCU is one of the GPU’s which can execute instructions and produces results. In figure 3.7, the basic instructions of the TCU are given.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Parameter1</th>
<th>Parameter2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>---</td>
<td>---</td>
<td>No operation. Do nothing</td>
</tr>
<tr>
<td>CREATE_TASK</td>
<td>DEST_TN</td>
<td>---</td>
<td>To allow some tasks to generate another task.</td>
</tr>
<tr>
<td>END_TASK</td>
<td>DEST_TN</td>
<td>---</td>
<td>Remove a TN of a task from the task queue</td>
</tr>
<tr>
<td>CHANGE_PRIO</td>
<td>DEST_TN</td>
<td>NEW_PRIO</td>
<td>Change the priority of a task.</td>
</tr>
<tr>
<td>GET_PRIO</td>
<td>DEST_TN</td>
<td>---</td>
<td>Read priority of a task</td>
</tr>
<tr>
<td>SLEEP</td>
<td>DEST_TN</td>
<td>---</td>
<td>Put the task itself or another tasks asleep.</td>
</tr>
<tr>
<td>AWAKE</td>
<td>DEST_TN</td>
<td>---</td>
<td>Awake a task. Making a task runnable.</td>
</tr>
<tr>
<td>DELAY</td>
<td>DEST_TN</td>
<td>DELAY_TIME</td>
<td>Request a timeout for a task without to block the task itself or another task.</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>DEST_TN</td>
<td>TO_TIME</td>
<td>Request a timeout for a task without blocking the task itself or another task.</td>
</tr>
<tr>
<td>CREATE_SEM</td>
<td>RN</td>
<td>TYPE</td>
<td>Create a short or a long semaphore</td>
</tr>
<tr>
<td>CREATE_MB</td>
<td>RN</td>
<td>NR_SLOTS</td>
<td>Create a mailbox with the given number of slots</td>
</tr>
<tr>
<td>CREATE_PIPE</td>
<td>RN</td>
<td>POINTER</td>
<td>Create a pipe. Pointer is the start address of the pipe</td>
</tr>
<tr>
<td>READ_MB</td>
<td>RN</td>
<td>---</td>
<td>Read data from a mailbox</td>
</tr>
<tr>
<td>WRITE_MB</td>
<td>RN</td>
<td>MESSAGE</td>
<td>Write data into a mailbox</td>
</tr>
<tr>
<td>READC_MB</td>
<td>RN</td>
<td>---</td>
<td>Read data from a mailbox conditionally</td>
</tr>
<tr>
<td>WRITEC_MB</td>
<td>RN</td>
<td>MESSAGE</td>
<td>Write data into a mailbox conditionally</td>
</tr>
<tr>
<td>READ_P</td>
<td>RN</td>
<td>READ_POINTER</td>
<td>Read data from a pipe</td>
</tr>
<tr>
<td>WRITE_P</td>
<td>RN</td>
<td>DATA</td>
<td>Write data into a pipe</td>
</tr>
<tr>
<td>READC_P</td>
<td>RN</td>
<td>READ_POINTER</td>
<td>Read data from a pipe conditionally</td>
</tr>
<tr>
<td>WRITEC_P</td>
<td>RN</td>
<td>DATA</td>
<td>Write data into a pipe conditionally</td>
</tr>
<tr>
<td>READ_S</td>
<td>RN</td>
<td>NR_UNITS</td>
<td>Decrease number of units from a semaphore</td>
</tr>
<tr>
<td>WRITE_S</td>
<td>RN</td>
<td>NR_UNITS</td>
<td>Increase number of units from a semaphore</td>
</tr>
<tr>
<td>READC_S</td>
<td>RN</td>
<td>NR_UNITS</td>
<td>Decrease number of units from a semaphore</td>
</tr>
</tbody>
</table>

Figure 3.7. Basic instructions of the TCU.
4. DESIGNING THE TASK CONTROL UNIT

As described in the specification in chapter 3, the TCU will be implemented in three main units as the TaskQ, the Core Unit and the Network Manager, as shown in figure 3.1.1 on page 5. To fulfill the constraints of the TCU, this scheme will be expanded as shown in figure 4a.

![Figure 4a. The conceptual architecture of the TCU.](image)

From this scheme and with the information of chapter 3, designing the TCU can be started. Fortunately, the blocks TaskQ and “Network Manager” are implemented (partially). Except the sub-blocks “Encode &Decode Message” and “Link Manager” of the “Network Manager” is still not implemented. In addition, the blocks “Command Receiver”, “Activity Detection” and “Result Transmitter” are already implemented in IDaSS. This means, that only the block “Execution Core” has to be designed.

Normally, the design of the “Execution Core” will be done in two sequential steps. Firstly, the behavior will be described using flowchart diagrams in a top-down way. In addition, corresponding data structures will be given. The advantage of such method is that the behavior can be mapped directly to a conceptual architecture or vice versa. For example, the scheme of figure 4a can be mapped onto a flowchart diagram with three subroutines as shown in figure 4b.

![Figure 4b. Behavior description for the TCU.](image)

Because the design of the TCU can be changed and/or be tuned in the future, it is preferably to implement the TCU on an algorithmic level to be able to improve the correctness efficiently. Secondly, the TCU will be implemented in details using IDaSS, when the improvement has been completed. However, trying to finish the first design for the “Execution Core”, only the behavior of the “Execution Core” will be designed, in such a way so that the design can be implemented easily by mapping the behavior description to an arbitrary programming code.
4.1 Designing the Execution Core

The intention of Execution Core is executing requests, which consist of local requests, remote requests and events. Local requests are sent by local tasks via the Function Switch while remote requests are sent by remote tasks received via the Network Manager. Besides, also the events have to be handled, which are requested by peripherals via the four activity lines. These are the three main activities of the Execution Core.

Further, the local tasks must be wake that had put itself or another task asleep for the given timeout time. When the number of running and runnable tasks that have the same priority is greater than the number of available LPU's, these tasks must be time-slice.

So far, only local activities are described that the TCU has perform in a single MIIP system. On a multi-MIIP system, the TCU on each MIIP has also to perform the next activities. When an MIIP communicates with other MIIP's, different messages will be sent to each other. Therefore, these messages must be received and handled.

When a local task had sent a message onto the network, it will set the timeout time for this message. Within this timeout time, the response of the target MIIP must be received. Therefore, the local tasks' message timeout must be handled. If a local task's request had been placed on the remote waiting list for a while, the local MIIP must let the remote MIIP know periodically whether the local is still alive or not. As last, when a remote task's request had been placed on the local waiting list and a message had been sent back, the timeout time for this message is set. It means the message timeout time for the remote tasks must be handled.

4.2 Request Execution

To be able to handle all requests and all events, the block "Request Execution" is needed and will function as the controller of the Execution Core. Its behavior description is shown in figure 4.2a.

```
1. IF (local request)
   2. Local_Request_Handling();
3. IF (timeout task)
   4. Timeout_Task_Handling();
5. IF (time-slicing task)
   6. Time_Slicing();
7. IF (remote request)
   8. Remote_Request_Handling();
9. IF (event)
   10. Events_Handling();
11. IF (message received)
    12. Message_Handling();
13. IF (waiting for remote resource)
14.    Local_Tasks_Liveliness_Checking();
15. IF (local task's message timeout occurred)
16.    Local_Tasks_Message_Timeout_Handling();
17. IF (remote task's message timeout occurred)
18.    Remote_Tasks_Message_Timeout_Handling();
```

Figure 4.2a. Behavior description of "Request Execution".

Because the Request Execution has to know what it has to do in each cycle, an 8-bits register called 'execution register' (E-register) is needed for controlling. On this register, several blocks can set a certain bit indicating what the Request Execution has to do. For more information about the E-register, see subsection 4.3.1 on page 19.

If there is local request, the local-request (LR) bit on the E-register is equal to 1b, the subroutine "Local Request Handling" is called for execution. Its behavior is described in section 4.4 on page 31. The local request is intended for accessing to a local resource, for accessing to a remote resource or it is a special instruction. They are described in section 4.6 on page 69, in section 4.7 on page 71 and in section 4.5 on page 34 respectively. If the there are local tasks that had requested for a certain timeout time to put itself or another task asleep, the
subroutine “Timeout Task Handling” is called for wake up. Its behavior is described in 4.8 on page 73. When there running and runnable tasks and they have the same priority, the subroutine “Time-Slicing” is called to time slice the tasks. Its behavior is described in section 4.9 on page 74.

When a remote task wants to access to a local resource, it will send a message including its request onto the network. If the remote-request (RR) bit on the E-register is high, it means a remote request is available. Therefore, the subroutine “Remote Request Handling” is called for execution. Its behavior is described in chapter 5 on page 76.

When a peripheral activates one of the event lines, the block “Activity Detection” will detect this and set the corresponding bit on the activity register (A-register). For more information about the A-register, see subsection 4.3.8 on page 29. It will also set the E-bit on the E-register. Therefore, the subroutine “Events Handling” is called for execution. Its behavior is described in chapter 6 on page 78.

In a multi-MµP system, tasks can communicate with each other by sending different messages onto their network. On each MµP, the TCU has fitted with a block called “Message Handling” to receive and handle all of these messages. When the message-received (MR) bit on the E-register is high, it means a new message is available. Its behavior is described in chapter 7 on page 79.

When a local task had requested for accessing to a remote resource and its request was placed on the remote waiting list, the local TCU has to let the remote MµP know periodically whether the local sending the request is still alive or not. Therefore, the subroutine “Local Tasks’ Liveliness Checking” is called for execution. Its behavior is described in chapter 8 on page 87.

When a local task’s request is sent onto the network, the timeout time for this message is set. When the timeout time is occurred, the subroutine “Local Tasks’ Message Timeout Handling” is called for handling timeout. Its behavior is described in chapter 9 on page 89. This principle is also used for the remote tasks’ message timeout. Its behavior is described in chapter 10 on page 91.

Using the information as written above, a conceptual architecture for the Execution Core is made and is shown in figure 4.2b on the next page.
The Execution Core has to be fitted with different registers and different lists, which are marked as gray blocks. In section 4.3 on the next page, they will be described in more details.

Further, in chapter 11 on page 93 some global functions of the Network Manager will be given.
4.3 Defining some registers and lists

For execution of requests, different registers and lists are needed. Firstly, the command register (C-register) is needed for holding new incoming local request. For new incoming remote request, it will be stored directly on to the remote-request-task list (RRTL). For events, the activity register (A-register) is used to indicate which line is active. For the block “Execute Resource Request” the register resource-instruction (RI) is needed for execution, else many accesses will be made to the task list (TL) or to the RRTL. It is a kind of cache. Because the block “Result Transmitter” is not part of the Execution Core, a result register (R-register) is needed for storing the result for sending back to the local task.

For control purposes, an execution register (E-register) is needed so that the controller “Request Execution” of the block “Execution Core” can determine what it has to do. In addition, for reporting purposes, an 8-bits status word SW is needed so that it can be reported whether a request is executed successfully or not.

For keeping some administration, the TL is needed for administrating of 63 local tasks and a resource list (RL) with corresponding resource data block (RD) for administrating of up to 128 local resources. Besides, the RRTL is needed for administrating of up to 16K – 64 remote tasks that want access to local resources.

In the next subsections, these registers and lists will be described comprehensively.

4.3.1 E-register

For control and handle all requests, a 9-bits Execution register (E-register) is defined. On this register, the Execution Core can determine what it has to do. The data structure of the E-register is shown in figure 4.3.1.

<table>
<thead>
<tr>
<th>LR</th>
<th>RR</th>
<th>E</th>
<th>TO</th>
<th>TS</th>
<th>LMT</th>
<th>RMT</th>
<th>LC</th>
<th>MR</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Local request</td>
<td>Remote request</td>
<td>Local task's timeout</td>
<td>Time slicing</td>
<td>Local message timeout</td>
<td>Remote message timeout</td>
<td>Liveliness checking</td>
<td>Message received</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.3.1. Data structure of the E-register.

When a local task sends a request to the Core Unit via the Function Switch, the Command Receiver will receive and stores it onto the C-register and sets the bit local-request (LR) indicating a new local request is available. After the request has been executed completely or when the request is placed onto the task list, the LR-bit will be reset (0b). As long as the LR-bit is low (0b), local tasks can send new requests.

When a new request is received via the Network Manager that is sent by a remote task, the block “Message Handling” will receive and stores the request directly onto the RRTL. After the requested resource is checked for existence, the bit remote-request (RR) is set to 1b indicating a new remote request is available. After the request has been executed completely, an error has been occurred or the request is cancelled, this bit can be set to 0b. However, as long as requests are present on the RRTL, this bit is kept high, because the controller “Execution Core” handles only one request in each cycle.

The bit E (event) will be set high by the block “Activity Detection” when there is one or more activity lines are active. The A-register is used for holding the active lines until the requests have been executed. It means the attached semaphore(s) will get the predefined number of units. The requests are coming from peripherals.

There are four types of timeout times, which will be handled by the controller “Execution Core”. The timeout time for time-slicing purposes, the timeout time the local tasks had requested, the timeout time for waiting of acknowledgements of messages that had been sent onto the network and the timeout time that is needed for checking the local tasks’ liveness.

For local task that requests for a timeout to put itself or another task asleep for the given timeout time, the TO-bit is defined. For time slicing, the TS-bit is defined. Each time the time-slice timer has expired the time-slice time, it set the TS-bit to 1b. The time-slice time is the time the task(s) will be executed for a predefined time duration after which other task(s) will be executed (time sliced). For keeping the message timeout for both the local and the remote tasks, the local-message-timeout (LMT) bit and the remote-message-timeout (RMT) bit are
defined respectively. Each time the message timer has reached the predefined timeout time, it will set the LMT-bit high for a local task and setting the RMT-bit high for a remote task.

When local tasks request for accessing to remote resources and the resources are not available yet, their requests are placed onto the remote RRTL. To let the remote MIP's handling the resources know whether the local tasks are still alive, the local MIP will send a check-if-alive (CIA) message onto the network. Because the check will be executed periodically, the timer will set the liveliness-check (LC) bit to 1b each time it has reached the predefined timeout time.

When a new message has been received via the Network Manager, the block "Encode & Decode Message" will set the message-received (MR) bit on the E-register to 1b indicating a new message is available, after the message has been decoded.

For initialization, all bits on the E-register are set to 0b.

4.3.2 Status word

On the MIP, there is an interface between the Function Switch (FS) and the TCU and an interface between Result Switch (RS) and the TCU. When there is a request intended for the TCU, the FS will send it through to the TCU. After the TCU has executed the request and a result has been produced, the TCU will send the result back to the task via the RS. In figure 4.3.2a, the interfaces of the TCU with the FS and the RS are shown.

![Figure 4.3.2a Interfaces of the TCU with the FS and the RS.](image)

Each time the FS sends a request to the TCU, the 8-bits status word (SW) is included. In addition, the read-status-word (RD_SW) bit and the write-status-word (WR_SW) bit are included. If the RD_SW-bit is low (0b), the received SW is a dummy. It means the SW is not needed for executing the request. Standard, the TCU has to administrate 63 local tasks. On the task list, the TCU has to keep the SW of each task by making a copy of the received SW. Because the RD_SW-bit is low, which means that the SW of the received request is not available, the TCU has to store the SW that is equal to 00h. If the RD_SW-bit is high (1b), it means the received SW is available. Therefore, the TCU needs to make a copy of the received SW. For administration, the RD_SW-bit is not needed to keep.

Further, if the WR_SW-bit is low, the task sending the request does not expect the SW when the request has been executed completely and the result is sent back. It does not matter whether the SW has been updated or not. When a result is produced, the TCU can include the SW in the result to send back to the task. However, because the WR_SW-bit is low, the task will ignore the SW. If the WR_SW-bit is high, the task does expect the
The Multi-Micro Processor

The Task Control Unit: Specification and Design

SW. By checking the WR_SW-bit, the task can determine whether it has to read the SW or not. Therefore, the TCU has to keep up the WR_SW-bit on the task list. However, its content may not be changed. Therefore, for sending the result back, the SW and the WR_SW-bit are included in the result.

For execution, the TCU needs to make a copy of the received SW. In figure 4.3.2b, the data structure of the SW is shown.

| ERR | RAE | TAE | OOM | CF | NR | RC | *
|-----|-----|-----|-----|----|----|----|------
| 7   | 6   | 5   | 4   | 3  | 2  | 1  | 0    |
| 0 = OK | Resource access error | Task access error | Out of memory | Condition failed | No response | Request cancelled | Not in use |

Figure 4.3.2b. The data structure of the status word (SW).

For the moment, the next bits are defined:

**ERR:**
- 0b: Request has been executed successfully.
- 1b: Execution is failed. One of the next bits has to be set to indicate what went wrong

**RAE:** Resource access error

**TAE:** Task access error

**OOM:** Out of memory

**CF:** Condition is failed

**NR:** No Response

**RC:** Request is cancelled

The meanings of bit 4 down to bit 1 are trivially. They are used for general purposes. Only bit number 0 is not in use for the moment.

For example, when remote task sends a request for accessing to a local resource, it can happen that there is no more memory left for storing the request. When accessing to a resource conditionally, the required condition can be failed. It can happen that the target MUP is still alive, but it does not respond while a message has been sent. When accessing to a remote resource and the its liveness is checked, the MUP handling the resource can send a message back indicating the requested resource does not exist anymore, the target is dead. In addition, when the local task’s request had been placed on the remote waiting list, the local task can send the cancel-message to the MUP handling the resource to cancel its request.

When accessing to a resource is not successfully, the RAE-bit is set to 1b indicating resource access error. After that, it must be possibly to indicate the exact reason. For the moment, the next reasons are defined:

**RIR:** Resource Is Removed

**IRN:** Incorrect RN

**IML:** Insufficient Memory Left

**DNA:** required resource Data blocks Not Available

**RIA:** Resource Is Alive

Because bit 4 down to bit 0 of the SW are not needed anymore when indicating the resource access error, they can be used to indicate the above reasons as shown in figure 4.3.2c.

<table>
<thead>
<tr>
<th>ERR</th>
<th>RAE</th>
<th>TAE</th>
<th>RIR</th>
<th>IRN</th>
<th>IML</th>
<th>DNA</th>
<th>RIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4.3.2c. Data structure of the SW for accessing to a resource.

Bit number 5, the TAE-bit, may not be changed, because the task sending the request need to know whether the meanings on bit 4 down to bit 0 are intended for accessing to a resource or accessing to a task.

For the moment, only 128 resources can be created and removed. They are numbered from 0 up to 127. When accessing to a resource and the RN of the requested resource is higher than the RN 127, the MUP handling the resource will set this RNE-bit high indicating the RN does not exist. When a task is placed on the local resource waiting list and the requested resource is removed, the waiting task will be wake up. To indicate what went wrong, the RNE-bit can also be used. It can happen, when accessing to a resource and that the RN and the IC do not match with each other. For example, the IC is intended for accessing to a mailbox while the RN is of a
The Multi-Micro Processor The Task Control Unit: Specification and Design

papers to. Therefore, the IRN is set indicating the incorrect RN. When a task attempts to create a new resource, its request can be rejected. The reason is that the resource memory is insufficient left to create e.g. a new pipe. Therefore, the IML-bit is set. For creating a long semaphore, a mailbox or a pipe while the required resource data blocks are not available, the DNA-bit will be set. When checking the resource liveliness, the RIA-bit can be used to indicate whether the resource is still alive or it has been removed. In the last case, the RNE-bit is set to 1b.

Further, for accessing to a task not successfully, bit 4 down to bit 0 can be used to indicate the more specific reason. For the moment, the next reasons are defined:

- **ITN**: Incorrect TN
- **TIU**: Task already In Use
- **NAC**: Not Allowed to Create new task
- **TIA**: Task Is Alive
- **TID**: Task Is Dead

In figure 4.3.2d, the data structure of the SW is shown for accessing to a task.

<table>
<thead>
<tr>
<th>ERR</th>
<th>RAE</th>
<th>TAE</th>
<th>LAYER</th>
<th>Error 1</th>
<th>Error 2</th>
<th>Error 3</th>
<th>Error 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
</tbody>
</table>

*Figure 4.3.2d. Data structure SW for accessing to a task.*

Bit number 6, the RAE-bit, may not be changed, because the task sending the request has to know whether the reasons from bit 4 down to bit 0 are intended for accessing to a resource or to a task.

When executing a request, like the instruction END_TASK or the instruction GET_PRIO, it can happen that the TN of the destination task does not exist (anymore). Alternatively, when the TN of the destination is greater than 63, because there are only 64 tasks present on each Mf-lP. However, task number 0 is the boot-up task and the task number 63 is the NIL-task. Therefore, the destination task's state is IDLE. Therefore, the ITN-bit is set. When creating a new task, the new task may not be existed beforehand. Only tasks with the state IDLE may be created. Else, the TIU-bit is set. On each Mf-lP, the TCU has to administrate up to 63 tasks. When the TN of the destination task is equal to the TN of these tasks, the TCU will set the ITN-bit indicating the incorrect TN. Also when sending a message to a remote task and the TN of the task for which the message is intended does not match. As last, the not-allowed-to-create-new-task (NAC) bit can be used to indicate that a certain task is not allowed to create a new task. When accessing to a task and it does not exist anymore, the TID-bit is set. If it is still alive, it does still exist, the TIA-bit is then set.

When there are more than five errors are defined for accessing a resource or for accessing a task, the bit number 4 can be used to indicate the first four errors if it is equal to 0b. If this bit is equal to 1b, the next four errors can be indicated. This option is shown in figure 4.3.2e and in figure 4.3.2f.

<table>
<thead>
<tr>
<th>ERR</th>
<th>RAE</th>
<th>TAE</th>
<th>LAYER</th>
<th>Error 5</th>
<th>Error 6</th>
<th>Error 7</th>
<th>Error 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
<td>0/1</td>
</tr>
</tbody>
</table>

*Figure 4.3.2e. Data structure of the second layer for accessing to a task.*

*Figure 4.3.2f. Data structure of the second layer for accessing to a task.*
4.3.3 Resource administration

As described in the specification in chapter 3, the TCU has to be able to keep the administration of up to 128 resources. So, depend on the availability of the resource memory up to 128 resources can be created. In figure 4.3.3a, the data structure of the resource list (RL) is shown with corresponding resource data block (RD).

**Figure 4.3.3a. Resource list (RL) with corresponding resource data block (RD).**

The intention of the resource list (RL) is keeping the administration of up to 128 local resources. In addition, the waiting local tasks and or the waiting remote tasks have to be administrated that wait for a certain resource. This means that for each resource a pointer called First_TN is needed to refer to the first waiting task and a pointer called Last_TN that refers to the last waiting task. Therefore, a doubled linked list is used for this purpose. To be able to index up to 16K tasks these pointers are 14 bits in width. Together with the TL that will be described in subsection 4.3.4 on the next page and the RRTL that will be described in subsection 4.3.5 on page 26 the local resource waiting list (RWL) can be realized.

The RL has a fixed length of 128 blocks so that up to 128 resources can be created. For indexing all blocks, it is also the resource number, a 7 bits pointer RL_Index is needed. For storing data of each resource, a resource data memory block (RD) is needed, which also has a fixed length of 128 blocks. Because each block on the RD corresponds with each block on the RL, the index RL_Index can also be used for indexing all blocks on the RD.

Initially, for each resource on the RL a block on the RD is reserved for storing data. So, block number 0 of RD corresponds with the resource number 0, block number 1 of RD corresponds with the resource number 1, etc. However, when a mailbox with a resource number 0 is created with two slots, block number 0 on the RL is used for storing the information of the mailbox while block number 0 and block number 1 on the RD are used for
storing the messages of the mailbox. It means that no other resource can be created on block 1 of the RL except a short semaphore, because block number 1 on the RD is already in use by the mailbox number 0. Short semaphore means that its data is not 32 bits in width as the corresponding block on the RD is used usually, it is just 12 bits in width by combining the fields “Nr_Slots”, “Nr_Used” and the field “Next_Message” on the concerning block of the RL. The field “Last Operation” is used to indicate whether it is a short semaphore (“Last Operation” = 0b) or it is a long semaphore (“Last Operation” = 1b).

Each block of the RL is divided into two sub-blocks, Block 0 and Block 1. Block 0 is used for indicating the resource type and for keeping the waiting tasks while Block 1 is used for storing the resource’s information. The field “RT” indicates the resource type. Is it a semaphore the RT = b, is it a mailbox the RT = 01b and is it a pipe the RT = 10b. For a semaphore, also the field “Last Operation” in block 1 is used to indicate whether it is a short semaphore (“Last Operation” = 0b) or it is a long semaphore (“Last Operation” = 1b). For RT = 11b, it is not defined for the moment. In the future work, this value can be used for indicating the long mailbox. For each resource, two pointers are needed for keeping up waiting tasks that wait for a certain resource. The pointer First_TN refers to the first local or remote waiting task while the pointer Last_TN refers to the last local or remote waiting task. This is needed so that waiting tasks can be wake up when the requested resource is available. When there are no tasks waiting for a certain resource, these pointers are set to 03FFFH (NIL) to indicate the start or the end of the list.

For initialization, all blocks on the resource list RL are initialized as 128 short semaphores as shown in figure 4.3.3b.

```
<table>
<thead>
<tr>
<th>RT</th>
<th>First_TN</th>
<th>Last_TN</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>03FFFH (NIL)</td>
<td>03FFFH (NIL)</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Nr_Units</th>
<th>Last_Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b</td>
<td>0b</td>
</tr>
</tbody>
</table>
```

Figure 4.3.3b. Initialization of the resource list RL.

The field “RT” on Block 0 is set to 00b indicating a semaphore while the field “Last_Operation” on Block 1 is set to 0b indicating a short semaphore. All other fields are set to 0 and or to NIL.
4.3.4 Local task administration

As described in the specification, the TCU must be able to administrate of up to 63 local tasks. Because the last task (task number 0) is the NIL-task, the block "Execution Core" of the TCU has only to administrate 63 tasks. In figure 4.3.4a, the data structure of the task list (TL) is shown.

The TL has a fixed length of 63 blocks for administrating of 63 local tasks, one block for each task. It means block number 0 is intended for task number 0, block number 1 is intended for task number 1, etc. For indexing all 63 tasks, a 6-bits index pointer called TL_Index is needed.

Because local tasks can be blocked temporarily because of waiting for a certain local resource or waiting for a certain timeout, different fields are needed. When waiting for a timeout finish time, the finish timeout time must be kept so that concerning task can be wake up after the requested timeout time is occurred. Because different tasks can request for a timeout time using the instruction TIMEOUTO and only one timeout timer is used for all tasks, a double linked list is used so that waiting tasks can be wake up easily. Therefore, the pointer called Prev_TO_TN is needed to refer to the previous waiting task while the pointer called Next_TO_TN is needed to refer to the next waiting task. Because only local tasks can request for a timeout time, 6-bits is enough for each pointer for indexing. For initialization, these pointers are set to NIL (03FH) to refer to the beginning or to the end of the list. For keeping the message timeout time, the TO-bit is defined. For further information, see chapter 9 on page 89 and chapter 10 on page 91.

For tasks that are waiting for a certain local resource also a doubled linked is used so that they can be wake up when the requested resource is available. Therefore, the pointer called Prev_TN is needed to refer to the previous waiting task while the pointer called Next_TN is needed to refer to the next waiting task. Because both the local and the remote tasks can wait for a local resource, 14 bits will be needed for each pointer for indexing. When a task has been wake up, its request will be executed. Therefore, the whole request has to be held. For a remote task, its request is kept on the RRTL while the request of a local task is kept on the TL. In figure 4.3.4b, the data structure of the original local request is shown.

<table>
<thead>
<tr>
<th>State</th>
<th>TO</th>
<th>CT</th>
<th>Waiting for timeout</th>
<th>Request</th>
<th>Waiting for local resource</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3b</td>
<td>able to create other task</td>
<td>1b</td>
<td>32b</td>
<td>6b</td>
</tr>
<tr>
<td>Task's state</td>
<td>Local task's message timeout</td>
<td>Time to wake up the timeout task</td>
<td>Previous waiting timeout task</td>
<td>Next waiting timeout task</td>
<td>See figure 4.3.4b below for further information</td>
</tr>
</tbody>
</table>

Figure 4.3.4a. Data structure of the task list TL.

For tasks that are waiting for a certain local resource also a doubled linked is used so that they can be wake up when the requested resource is available. Therefore, the pointer called Prev_TN is needed to refer to the previous waiting task while the pointer called Next_TN is needed to refer to the next waiting task. Because both the local and the remote tasks can wait for a local resource, 14 bits will be needed for each pointer for indexing. When a task has been wake up, its request will be executed. Therefore, the whole request has to be held. For a remote task, its request is kept on the RRTL while the request of a local task is kept on the TL. In figure 4.3.4b, the data structure of the original local request is shown.

<table>
<thead>
<tr>
<th>TN</th>
<th>SW</th>
<th>WR_SW</th>
<th>RRA</th>
<th>IC</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6b</td>
<td>8b</td>
<td>1b</td>
<td>5b</td>
<td>10b</td>
<td>32b</td>
<td>32b</td>
</tr>
<tr>
<td>task number</td>
<td>status word</td>
<td>write SW</td>
<td>result return address</td>
<td>instruction code</td>
<td>RN + MUP address (see section 4.5.12)</td>
<td>required parameter / RIN</td>
</tr>
</tbody>
</table>

Figure 4.3.4b. Data structure of a local request.

Normally, the field "Parameter 2" is used for storing the required parameter of the local request. For accessing to a remote resource, a copy of the whole is sent onto the network towards the remote MUP handling the resource. If the requested resource is not available yet, the request will be placed onto the remote RRTL. Therefore, the remote MUP handling the resource will send the placed-on-waiting-list (PWL) message including the remote index number (RIN) back. The RIN is just the index number of the remote block on which the
request is stored and is 14-bits in width. For each communication, the RIN must be included into the message. Because the request has been sent successfully, the content of the field “Parameter 2” is not needed anymore. Therefore, it can be used for storing the RIN. By checking the task’s state, it can be determined whether the RIN is known or not. As last, for storing the request on the TL the field “TN” is left out. It is already indexed by the pointer TL_Index.

Further, a 3-bits task’s state is needed to administrate all possible states of a task. In figure 4.3.4c, all states of a local task are shown.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsRTR</td>
<td>Ready to run</td>
</tr>
<tr>
<td>tsWTO</td>
<td>Waiting for timeout. Only used for waiting for a timeout</td>
</tr>
<tr>
<td>tsSL</td>
<td>Asleep / blocked unconditionally</td>
</tr>
<tr>
<td>tsWLR</td>
<td>Waiting for local resource</td>
</tr>
<tr>
<td>tsWRR</td>
<td>Waiting for remote resource</td>
</tr>
<tr>
<td>tsWCA</td>
<td>Waiting for command acknowledgement</td>
</tr>
<tr>
<td>tsCRL</td>
<td>Checking resource liveliness</td>
</tr>
<tr>
<td>tsWEWA</td>
<td>Waiting for end wait acknowledgement</td>
</tr>
</tbody>
</table>

Figure 4.3.4c. Local task’s states.

For the moment, only these states are defined. If needed, more states can be added.

As last, the create-task (CT) bit on the TL is needed to indicate which task is allowed to create another task. When a task sends an instruction to the TCU requesting for creating another task, this bit must be checked. If this bit is low, the request must be cancelled and the SW updating to indicate what went wrong. In this case, both the TAE-bit and the NAC-bit are set to 1b.

Standard, only the Boot-up task, task number 0, is allowed to set or to reset CT-bit. It means, task with the CT-bit = 1b can create another task. When creating, it is also allowed to set the CT-bit of the new task high.

4.3.5 Remote task administration

Because all tasks on remote MPU’s are allowed to access to local resources, a four words per block remote-request-task list (RRTL), a double linked list, is used for keeping the administration for all remote requests until they are executed completely. Alternatively, until a request is cancelled by the concerning or another remote task.

Because the number of remote requests is not known beforehand, the RRTL will be kept in the main memory, in a dynamic structure. To save the space, the number of the task blocks (= number of remote requests) is limited by 16K, which will be indexed by 14 bits to allow maximum \(2^{14} = 16K\) remote tasks to wait for local resources. The last 64 tasks are mapped to the local tasks, allowing a maximum of \((16K - 64 =) 16128\) remote tasks waiting for local resources. Because each task block is four words in width, it gives a maximum memory requirement of \((4 \text{ words} \times (16K - 64) =) 63K\) words. Local tasks are indicated with indexes \(03FC0H\) to \(03FFE_H\). Index \(3FFF_H\) indicates ‘no task’ (for beginning and end of lists).

The memory structure for remote tasks is initialized with a (single linked) free list. Because remote task blocks are allocated from this free list and returned to the free list when no longer needed, the actual size of the remote task memory is as long as the initial free list (which is easy to parameterize). For referring to a free block on the RRTL, a 14-bits pointer called Free_Block is needed, as is the base address (which must lie at a 64K word boundary unless an offset adder is used). When a remote task is placed onto the RRTL, the value of the pointer Free_Block of this task will be included into the message used as remote index number (RIN) to send back to the remote task. For all communication between the remote task sending the request and the MPU handling the resource, the RIN must be included into the message for speeding up the reply generation. The data structure of the RRTL is shown in figure 4.3.5a on the next page.
a) Data structure of the RRTL.

<table>
<thead>
<tr>
<th>Block State</th>
<th>Prev TN</th>
<th>Next TN / Free Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 28 27</td>
<td>0</td>
<td>13 0</td>
</tr>
</tbody>
</table>

See figure 4.3.5b

Word 0

Word 1

<table>
<thead>
<tr>
<th>Remote TN</th>
<th>RN</th>
<th>SW</th>
<th>WR_SW</th>
<th>IC</th>
<th>TO</th>
<th>Parameter / result word</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bytes</td>
<td>31</td>
<td>20</td>
<td>19</td>
<td>12</td>
<td>11</td>
<td>10 1 0</td>
</tr>
<tr>
<td>Include MUP address</td>
<td>local requested RN</td>
<td>Status word</td>
<td>write SW</td>
<td>instruction code</td>
<td>Time out</td>
<td>if needed</td>
</tr>
</tbody>
</table>

b) RRTL memory block

<table>
<thead>
<tr>
<th>Free_Block</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Word [0]</th>
<th>Word [1...3]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.3.5a. The data structure of remote-request-task list (RRTL).

Each block on the RRTL is divided into two sub-blocks. The first sub-block is formed by WORD 0 for keeping the administration while the second sub-block is formed by WORD 1, WORD 2 and WORD 3 for storing the information of the remote request. On the first sub-block, the field “Block State” indicates the block’s state of each block. It is four bits in width. Totally, up to 16 block’s states can be defined. However, for the moment, only six block’s states are defined as shown in figure 4.3.5b.

<table>
<thead>
<tr>
<th>Block State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsTBF</td>
<td>Task block free: This task block is not in use by a remote task.</td>
</tr>
<tr>
<td>rsOWLA</td>
<td>On waiting list, alive: This task block contains a waiting remote task. It is alive, because it just started either waiting or a ‘check if alive’ message is received from it.</td>
</tr>
<tr>
<td>rsOWLC</td>
<td>On waiting list, checking: Entered when a ‘check if alive’ message is sent, because this block has had its cyclic visit and since that time the remote task has not sent a ‘check if alive’ message.</td>
</tr>
<tr>
<td>rsOWLS</td>
<td>On waiting list, suspect: It is suspected that something is wrong. Entered, because the message ‘target is alive’ is still not received since the message ‘check if alive’ is sent.</td>
</tr>
<tr>
<td>rsRS</td>
<td>Result sent: Entered when the remote-wait-result (RWR) message is sent and is waiting for the result acknowledgement (RA).</td>
</tr>
<tr>
<td>rsRSC</td>
<td>Result sent, checked: Entered when the RA message is still not received since the RWR message is sent. Because the sent RWR-message can get lost somehow, it will be re-sent, but its acknowledgement is not awaited anymore.</td>
</tr>
</tbody>
</table>

Figure 4.3.5b. Defined block’s states.

For the moment, the field “RN” is 12 bits in width. Normally, only seven bits are needed for indexing up to 128 local resources. The other five bits can be used in the future for indexing the larger number of local resources. Alternatively, for indexing the block’s state when there are more than 16 states are needed.

Further, the field “Prev_TN” is needed to refer to the previous local or remote task that is waiting for accessing to a local resource. If a task is the first or the only task on the local resource waiting list, this field has to be set to NIL (03FFFFH) to indicate the beginning and or the end of the waiting list. In addition, the field “Next_TN” is needed. It refers to the next local or to the next remote waiting task. This field will be set to NIL (03FFFFH) if it is the last waiting task. Alternatively, there is only one waiting task. Besides, this is also used for free block
chaining for realizing a single linked free list. After initialization, this pointer is set to refer to the next free block on the RRTL.

On the second sub-block, the whole request of the remote task is stored. On the field "Remote TN" the task number of the remote task, sending the request is kept including its MIP address. For the communication, this field is very important especially for sending the result back after the request has been executed completely. On WORD 2 the requested local resource number RN, the status word SW and the instruction code is kept. As for the TL, which is already described in the previous subsection on page 25, the write-status-word (WR_SW) bit must be kept indicating whether the SW has to be updated or not. As last, on WORD 3, the required parameter for executing the request is stored and or the result word is held after the request has been executed successfully.

4.3.6 Local resource waiting list

There is a relation between the resource list RL, the task list TL and the request-task list RRTL. As described in subsection 4.3.4 on page 25 and in subsection 4.3.5 on page 26, the TL keeps the administration of 63 local tasks while the RRTL keeps the administration of all waiting remote tasks. Mainly, the RL is needed for administrating of 128 resources. For realizing a waiting list on which both the local and the remote tasks can wait for a local resource, two extra pointers are needed for each resource, the pointer First_TN and the pointer Last_TN. The pointer First_TN refers to the first waiting task while the pointer Last_TN refers to the last waiting task. In turn, on both the TL and the RRTL the pointer Prev_TN refers to the previous waiting and the pointer Next_TN refers to the next waiting. On this way, a double linked local resource waiting list is realized.

An example, the local task with the task number 05_H and 10_H and the remote tasks with the task number 064_H and 075_H are waiting for the same mailbox with the resource number 2. On the TL, the local tasks are indexed by the pointer TL_Index. In addition, on the RRTL, the remote tasks are indexed by the pointer Free_Block. For the remote tasks sending the requests, this pointer is called the remote task index (RIN). So, for administrating the local resource waiting list, the value of the pointer TL_Index and the value of the pointer RIN are used. The order that a task places its request decides its priority, Suppose, the requested resource is available, after which these tasks will be handled according to the sequence 05_H, 075_H, 064_H and 10_H. In figure 4.3.6a, the waiting list with concerning waiting tasks is shown before execution.

![Figure 4.3.6a. Example of the local resource waiting list](image-url)
When the RRTL has to refer to a local task, the TN indexed by the pointer TL_Index is increased by the offset 03FC0H. Else, a remote task is referred. This is needed, because the local tasks are mapped to 03FC0H up to 03FFE1H.

In addition, in figure 4.3.6b the situation is depicted graphically.

**Figure 4.3.6b. Depicting the waiting lists graphically.**

### 4.3.7 C-register

Each new incoming local request will be held on the command register (C-register) until the request is executed completely or when the request is blocked temporally. In the last case, the whole request is stored on the TL. In figure 4.3.7, the data structure of the C-register is shown.

<table>
<thead>
<tr>
<th>TN</th>
<th>SW</th>
<th>WR_SW</th>
<th>RD_SW</th>
<th>RRA</th>
<th>IC</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6b</td>
<td>8b</td>
<td>1b</td>
<td>1b</td>
<td>5b</td>
<td>10b</td>
<td>32b</td>
<td>32b</td>
</tr>
</tbody>
</table>

*Figure 4.3.7. Data structure of the C-register.*

TN is the task number of the local task sending the request. For storing the request, this field is left out. SW is the status word of the task sending the request. For execution, the SW can be read before the execution and updated after the execution if it is required. RRA is the return result address of the LPU handling the local task on which the result will be stored. IC is the instruction code. On Parameter 1, the task number TN or the resource number RN can be stored. For accessing to a remote resource also the address of the MPU handling the resource is included. On the field “Parameter 2”, the data is stored that is needed for execution.

### 4.3.8 A-register

For handling the requests of peripherals, a 4-bits activity register (A-register) is needed for holding the values of the four activity lines. When peripherals activate one or more of the activity lines, the block “Activity Detection” will set (1b) the concerning bit(s) on the A-register. For executing, the predefined number of units will be increased (V-operation) on the concerning semaphore. After execution the concerning bit(s) will be reset (0b) so that new requests can be placed.
4.3.9 RI-register

To avoid that the block “Execute Resource Request” has to make many accesses to the TL and or to the RRTL, the request is placed onto the resource-instruction (RI) register before it will be executed. The RI-register operates like as a cache. In figure 4.3.9, the data structure of the RI-register is shown.

<table>
<thead>
<tr>
<th>Avail</th>
<th>TN</th>
<th>SW</th>
<th>WR_SW</th>
<th>IC</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1b</td>
<td>14b</td>
<td>8b</td>
<td>1b</td>
<td>10b</td>
<td>32b</td>
<td>32b</td>
</tr>
</tbody>
</table>

*Figure 4.3.9. Data structure of the RI-register.*

By checking the field “Avail” the block “Execute Resource Request” can determine whether there is a new request available or not. The TN is not only needed to be able to report to the task sending the request when a request has been executed completely, it is also needed for sending the result back to the block “Result Transmitter”. Because the TCU has to administrate up to 16K tasks and the block “Execute Resource Request” has to execute requests of both the local and the remote tasks, the field “TN” is 14-bits in width. For sending the result back, the TN, the SW, the WR_SW-bit, the RRA and eventually the result word are sent back, if it is a local task. On the TCU, there are 64 local tasks. For addressing, they are mapped on the addresses from 03FC0H up to 03FFEH, which are 6-bits in width. For sending the result back, only the 6-bits value is needed. Because the RRA is not kept on the RI-register, it must be fetched from the TL. For a remote task, the result message will be sent onto the network. In the message, the TN plus its MPU address, the SW, the WR_SW-bit and eventually the result word are included.

4.3.10 R-register

For sending the result back to the local task, the result must be placed onto the result register (R-register), because the block “Result Transmitter” operates simultaneously with the Execution Core. In figure 4.3.10, the data structure of the R-register is shown.

<table>
<thead>
<tr>
<th>Avail</th>
<th>TN</th>
<th>Result Word</th>
<th>RRA</th>
<th>SW</th>
<th>WR_SW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1b</td>
<td>6b</td>
<td>32b</td>
<td>3b</td>
<td>8b</td>
<td>1b</td>
</tr>
</tbody>
</table>

*Figure 4.3.10. Data structure of the R-register.*

Because the block “Result Transmitter” operates simultaneously with the block “Execution Core”, it can check the AVAIL-bit to determine whether there is a new result available or not. As mentioned by the RI-register, the TN, the SW, the WR_SW-bit the RRA and eventually the result word are needed for sending back. The WR_SW-bit is just a copy of as it is received. By checking this bit, the local task sending the request can determine whether it has to read the SW or ignore it.
4.4 Local Request Handling

The block “Local Request Handling” is used for handling of all local requests that are sent by local tasks. These requests are received by the block “Command Receiver” and are held on the command register (C-register) until they are executed completely or until the request is placed on the waiting list. But if a request is intended for accessing to a local resource, it will be placed onto the resource-instruction (Rl) register. When the requested resource is not available yet, the request is kept on the TL. For further information, see subsection 4.5.13 “Execute Resource Request” on page 56. In addition, if it is intended for accessing to a remote resource it will be sent onto the network. For further information, see section 4.7 “Remote Resource Request Handling” on page 71.

The intention of the block “Local Request Handling” is handling of all local requests; it means for accessing to local resources, for accessing to remote resources and executing of some special instructions. In figure 4.4, the behavior description of this block is shown.

![Figure 4.4. Behavior description of “Local Request Handling”.

When decoding the request, two types of request can be determined. Is the request intended for accessing to a resource or is it a special instruction. For accessing to a resource, it must be determined whether it is intended for accessing to a local or to a remote resource by checking the resource number (RN) of the requested resource, which is one of the parameters of the request. Because the address of the MultiProcessor handling the resource must be included in the RN decoding can be done by checking this address. If it is zero, it means the local resource is requested. Therefore, the subroutine “Local Resource Request Handling” is called. See section 4.6 on page 69 for further information. Else, a remote resource is requested and the subroutine “Remote Resource Request Handling” is called. See section 4.7 on page 71 for further information. The second type of request is intended for executing of some special instructions. In section 4.5 “Decode & Execute Instruction” on page 34, it will be described in more details. For executing an instruction, the request will be executed immediately. Therefore, a result word can be produced and the task sending the request can be wake up. Therefore, the subroutine “Sending Result To Local Task” is called for sending the result back and the subroutine “Wake Up Local Task” is called for waking up the task sending the request. In subsection 4.4.1 on page 32 and in subsection 4.4.3 on page 33, their behaviors are described respectively.

Because a task sending the request can be blocked temporally when accessing to a resource, like placing the task on a waiting list, a result cannot always be produced after the subroutine “Local Resource Request Handling”, or the subroutine “Remote Resource Request Handling” has been executed. Therefore, the subroutine “Sending Result To Local Task” cannot call for sending the result back. In addition, the subroutine “Wake Up Local Task” cannot be called to wake up the task. Sending the result back and waking up the task can only be executed when a request has been executed, a request has been cancelled or when an error has occurred. Internally, each subroutine has to call the subroutine “Sending Result To Local Task” for sending back the result and or the subroutine “Wake Up Local Task” or the subroutine “Wake Up” that is described on page 32 to wake up the task. As last, to be able to receive a new request the bit local-request (LR) on the E-register is set to 0b.
4.4.1 Sending result back to local task

After the request has been executed, a result can be sent back to the local task sending the request. In figure 4.4.1, the behavior of the subroutine “Sending result to local task” is described.

![Diagram](https://via.placeholder.com/150)

*Figure 4.4.1. Behavior description of “Sending result to local task”.*

When executing a request, a result word can be produced. Therefore, the result (the TN, the SW, the WR_SW-bit, the RRA and eventually the result word) has to be sent back to the local task sending the request.

On the MuP, each LPU has a number of registers called Result Return Address (RRA). A number of these registers are used for general purposes. Each time a local task sends a request to the TCU, a RRA is included into the request. If RRA ≤ 1, it means the task sending the request does not expect a 32-bits result word. Therefore, the result is not needed to send back. However, if the WR_SW-bit is high it means the SW must be updated after execution. Therefore, the result must be sent back regardless the value of the RRA. In this case, the result word is a dummy and is included in the result. If the RRA > 1, it means the task does expect a 32-bits result word. Therefore, the result must be sent back regardless a result word has been produced or not. If no result word has been produced, a dummy result word will be included in the message.

As described in subsection 4.3.10 on page 30, the block “Result Transmitter” can check the AVAIL-bit on the R-register to determine whether a new result is available or not. If this bit is high, a new result is available. So, no new result may be written to the R-register. It means, when attempting to write a new result to the R-register, it must be waited until the AVAIL-bit on the R-register is low. After the result has been placed onto the R-register, the AVAIL-bit must be set to 1b so that the result can be sent back.

4.4.2 Waking up a local task

When a request has been executed, the task sending the request can be wake up. In figure 4.4.2, the behavior of the subroutine “Wake Up” is described.

| 1. TL(TN).State | = tsRTR; |
| 2. TaskQ(TN).Runnable | = 1; |

*Figure 4.4.2. Behavior description of “Wake Up”.*
For waking up a local task, only its state on the TL is needed to change to tsRTR for administration purposes and its field “Runnable” on the TaskQ is set to 1b to make the task runnable.

### 4.4.3 Waking up a certain local task

When an instruction has been executed by the block “Decode & Execute Instruction”, the task sending the request can be wake up or blocked. In addition, a result can be sent back to the task. It is depended on the instruction. In figure 4.4.3a, all instructions that will be executed by the block “Decode & Execute Instruction” are analyzed.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Sending result and why?</th>
<th>Wake up or Blocked?</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOP</td>
<td>Yes, EI / UR</td>
<td>W</td>
</tr>
<tr>
<td>CREATE TASK</td>
<td>Yes, EI / UR</td>
<td>W</td>
</tr>
<tr>
<td>END TASK</td>
<td>Yes, EI / UR</td>
<td>W/B</td>
</tr>
<tr>
<td>CHANGE PRIO</td>
<td>Yes, EI / UR / R</td>
<td>W</td>
</tr>
<tr>
<td>GET PRIO</td>
<td>Yes, EI / UR</td>
<td>W</td>
</tr>
<tr>
<td>SLEEP</td>
<td>Yes, EI / UR</td>
<td>W/B</td>
</tr>
<tr>
<td>AWAKE</td>
<td>Yes, EI / UR</td>
<td>W</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>Yes, EI / UR</td>
<td>W/B</td>
</tr>
<tr>
<td>CREATE SEM</td>
<td>Yes, EI / UR</td>
<td>W</td>
</tr>
<tr>
<td>CREATE MB</td>
<td>Yes, EI / UR</td>
<td>W</td>
</tr>
<tr>
<td>CREATE PIPE</td>
<td>Yes, EI / UR</td>
<td>W</td>
</tr>
<tr>
<td></td>
<td>EI = Error indication;</td>
<td>W = Wake up</td>
</tr>
<tr>
<td></td>
<td>UR = Update SW required;</td>
<td>B = Blocked</td>
</tr>
<tr>
<td></td>
<td>R = Result</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.4.3a. Analyzing the special instructions.

As shown in figure 4.4.3a, a result can be sent back after executing an instruction. There are two conditions for sending back the result. Only when the RRA > 1 or when the WR_SW-bit is high. Because the value of the RRA and or the value of the WR_SW-bit must be checked before sending the result back, it does not matter whether a result has been produced or not. Therefore, the subroutine “Sending Result To Local Task” can always be called after the execution of each instruction. Its behavior is already described in subsection 4.4.1 on page 32.

Nonnally, the task must be wake up after its request has been executed. Only for the instruction END_TASK, SLEEP and the instruction TIMEOUT, the task sending the request can be blocked after execution. As shown in figure 4.4 on page 31, the subroutine “Wake Up Local Task” is called for each instruction to wake up the sending task. Therefore, this subroutine must check whether the instruction has been executed on the sending task itself or on the destination task. When executed on the sending task itself, this task must be blocked (temporally) and not wake up. In figure 4.4.3b, the behavior of the subroutine “Wake Up Local Task” is described.

```
1. IF ((TN != DEST_TN) OR (IC != END_TASK) OR (IC != SLEEP) OR (IC != TIMEOUT)) {
2.   TL(TN).State = tsRTR;
3.   TaskQ(TN).Runnable = 1;
4. }
```

Figure 4.4.3b. Behavior description of “Wake Up Local Task”.

So, only when the instruction END_TASK, the instruction SLEEP or the instruction TIMEOUT has not been executed, the task sending the request must be wake up. Therefore, its state on the TL is set to tsRTR and its field “Runnable” on the TaskQ is set to 1b to make the task runnable. Else, it must be checked whether the instruction is executed on the sending task itself or on the destination task. If on the destination, the TN of the sending task and the DEST_TN are not equally. Therefore, the sending task must be wake up.
4.5 Decode & Execute Instruction

For executing some special instructions, the subroutine "Decode & Execute Instruction" will decode and execute these instructions. Because only one of the instructions can be executed, a case-statement of the programming language C is used for decoding the instructions. The behavior description of "Decode & Execute Instruction" is shown in figure 4.5.

```
switch (IC) {
    case NOP: NOP; break;
    case CT: CREATE_TASK (DEST_TN); break;
    case ET: END_TASK (DEST_TN); break;
    case CP: CHANGE_PTO (DEST_TN, NEW_PTO); break;
    case GP: GET_PTO (DEST_TN); break;
    case SL: SLEEP (DEST_TN); break;
    case AW: AWAKE (DEST_TN); break;
    case DL: DELAY (DEST_TN, DELAY_TIME); break;
    case TO: TIMEOUT (DEST_TN, TO_TIME); break;
    case CRS: CREATE_SEM (RN, TYPE); break;
    case CRM: CREATE_MB (RN, NR_SLOTS); break;
    case CRP: CREATE_PIPE (RN, POINTER); break;
    default: break;
}
```

Figure 4.5. Behavior description of "Decode & Execute Instruction".

To avoid misunderstanding between the task number (TN) sending the request (the source) and the task number (TN) of the task on which changes can be taken place (the destination), the term "TN" will be used to indicate the source and DEST_TN will be used to indicate the destination.

In the next sub-sections, these instructions will be described in more details.

4.5.1 NOP

The instruction No Operation (NOP) just does nothing. It means that nothing will be changed, except a few clock cycles will be lost. In other words, some delay will be realized.

As inputs, only the task number (TN) and the instruction code (IC) are needed for execution. For execution, the block "Decode & Execute Instruction" only has to decode the instruction.

4.5.2 CREATE_TASK

Normally, 62 tasks on the TaskQ are initialized to non-existent. Therefore, their states are set to IDLE. The other task is the boot-up task. With the instruction CREATE_TASK(DEST_TN, PRIO) it is able to let a running task creates another task.

As inputs, the TN sending the request, the SW, the IC, Parameter 1 holding DEST_TN and Parameter 2 holding the priority are needed. The behavior of the instruction CREATE_TASK is shown in figure 4.5.2 on the next page.
If the state of the creating task is not equal to tsIDLE, it means this task is already in use. Therefore, the SW is updated to indicate what went wrong. In this case, the ERR-bit on the SW is set to 1b indicating an error has been occurred. Because accessing to a task an error has been occurred, the TAE-bit is set to 1b so that bit number 4 down to bit number 0 can be used to specify the error in more details. Therefore, the TIU-bit is set to 1b indicating task-already-in-use.

If the state of the creating task on the TL is equal to tsIDLE (not in use) and the task sending the request is allowed to create a new task, its CT-bit on the TL is equal to 1b, its priority on the TaskQ is set to PRIG (Parameter 2). However, if the task sending the request is not allowed to create a new task, its CT-bit on the TL is equal to 0b, the NAC-bit on the SW has to set to 1b indicating that a task is not allowed to create another task. In addition, the ERR-bit and the TAE-bit have to be set to 1b.
4.5.3 END_TASK

With the instruction END_TASK(DEST_TN), a local task can remove itself or another existing local task from both the TaskQ and the TL, irrespective of the state of the task at that moment. When the target task does exist or it is in use, its state on the TL is set to tsIDLE and its field “Runnable” on the TaskQ is set to 0b. Therefore, its activity has to be cancelled immediately. This activity can be waiting for accessing to a remote resource, waiting for a remote message, waiting for accessing to a local resource or waiting for a timeout time.

As inputs, the TN of the task sending the request and the status word SW to be able to report what went wrong will be needed. Further, the instruction code IC and Parameter 1 (=DEST_TN) are needed. The behavior description of this instruction is shown in figure 4.5.3.

![Figure 4.5.3. Behavior description of instruction END_TASK.](image)

If the task number DEST_TN of the removing task is not in use, TL(DEST_TN).State = tsIDLE, the SW is updated to indicate what went wrong for sending back to the local task sending the request. In this case, the TID-bit is set to 1b indicating the TN does not exist anymore, it is dead. Else, it means the target task is in use and can therefore be removed from the TL and from the TaskQ by calling the subroutine “Removing task from TaskQ/TL”. Its behavior description is described in subsection 4.5.3.3 on page 39.

However, before an existing task can be removed from the TaskQ and from the TL, its current activity must be cancelled immediately. Because the TCU can only execute request by request sequentially, its current activity can be waiting for a remote resource, waiting for a remote message, waiting for a local resource and or waiting for a timeout time. In all other cases, the task is runnable or is blocked. Therefore, the task can be removed immediately by calling the subroutine “Removing task from TaskQ/TL”.

If the removing task is waiting for accessing to a remote resource or for a message, the cancel-wait (CW) message is sent onto the network towards the target resource including the remote index number (RIN), if it is known on the TL. When a remote-wait-command (RWC) message had been sent and the acknowledgement has not been received, then the RIN is unknown on the TL. Because the target task will be removed unconditionally, the acknowledgement of the CW-message is not awaited. After that, its message's timeout time must be cancelled by setting its TO-bit on the TL to 0b. If the removing task is waiting for accessing to a local resource, the subroutine “Removing task from resource waiting list” is called for removing the task from the resource waiting list. Its behavior is described in subsection 4.5.3.1 on the next page. If the removing task is waiting for a timeout time, the target task will be removed from the timeout waiting list by calling the subroutine “Removing task from TO waiting list”. Its behavior description is described in subsection 4.5.3.2 on page 39.
4.5.3.1 Removing a task from the local resource waiting list

A local task can be removed from the local resource waiting list in four situations. In figure 4.5.3.1a, these situations are depicted.

In the first situation, there is only one waiting task on the waiting list. That is also the removing task. Therefore, the pointer First_TN and the pointer Last_TN of the concerning local resource is waiting is equal with each other. In the other situations (2 up to 4), these pointers are not equal with each other. In situation 2 up to situation 4, there are at least two waiting tasks present on the waiting list. In the second situation, the removing task is the first waiting task on the waiting list and is referred by the pointer First_TN of the concerning resource. In the third situation, the removing task is the last waiting task on the waiting list and is referred by the pointer Last_TN of the concerning resource. In the last situation, the removing task lies in between two other waiting tasks. For processing, the pointer First_TN and the pointer Last_TN of the concerning resource are not needed.

In figure 4.5.3.1b on the next page, the behavior of the subroutine “Removing task from resource waiting list” is described. Therefore, the resource list RL, the task list TL and the remote-request-task list RRTL are needed.
1. Temp_RN = TL(DEST_TN).Request.RN;
2. IF (RL(Temp_RN).First_TN == RL(Temp_RN).Last_TN) { // First situation.
3.   RL(Temp_RN).First_TN = NIL;
4.   RL(Temp_RN).Last_TN = NIL;
5. }
6. ELSE
7.   IF (RL(Temp_RN).First_TN == DEST_TN) { // Second situation
8.     RL(Temp_RN).First_TN = TL(DEST_TN).Next_TN;
9.     TL(DEST_TN).Next_TN = NIL;
10.   IF (RL(Temp_RN).First_TN >= 03FC0H) // Local
11.     TL(DEST_TN).Next_TN = NIL; // local waiting task
12.   ELSE
13.     RRTL(RL(Temp_RN).First_TN).Prev_TN = NIL; // remote waiting task
14. }
15. ELSEIF(RL(Temp_RN).Last_TN == DEST_TN) {
16.     TL(DEST_TN).Last_TN = NIL; // local waiting task
17.     TL(DEST_TN).Prev_TN = NIL;
18.   ELSE
19.     RRTL(RL(Temp_RN).Last_TN).Next_TN = NIL; // remote waiting task
20. }
21. ELSE { // Fourth situation
22.   IF (TL(DEST_TN).Prev_TN >= 03FC0H) // Line nr. 1
24.   ELSE
26.   IF (TL(DEST_TN).Next_TN >= 03FC0H) // Line nr. 2
28.   ELSE
30. }
31. ELSE
32.     TL(DEST_TN).Next_TN = NIL;
33.     TL(TL(DEST_TN).Prev_TN = NIL;
34. }

Figure 4.5.3.1b. Behavior description of "Removing task from resource waiting list".

As shown in figure 4.5.3.1a on the previous page there are four situations in which a local task can be removed from the local resource waiting list. In line 2 up to line 4 of figure 4.5.3.1b, there is only one waiting task and that is the removing task. Therefore, the pointer First_TN and the pointer Last_TN are equal. So, only the pointer First_TN and the pointer Last_TN of the concerning resource are needed to be set to NIL. From line 5 to line 34 there are at least two waiting tasks present on the waiting list.

The code of line 6 up to line 13 removes a waiting task, which is the first task on the waiting list. Firstly, the pointer First_TN is set to refer to the next waiting task that is referred by the next-pointer Next_TN of the removing task, which after the pointer Next_TN is set to NIL. In line 10 the concerning task is removed from the task list TL if it is a local task. As described in subsection 4.3.5 "Remote Task Administration" on page 26 local tasks are addressed from 03FC0H up to 03FFEH. Else the task will be removed from the remote-request-task list RRTL. This is done in line 13.

The code from line 15 to line 22 a waiting task is removed, which is the last task on the waiting list. It is the same as removing the first task on the waiting list of the second situation. The difference is that the pointer Last_TN is set to refer to the previous waiting task that is referred by the previous pointer Prev_TN of the removing task.

In the fourth situation the code from line 23 to line 34 a waiting task is removed from the waiting list that lies between two other local and or remote waiting tasks. In this case, two main actions are made. Setting the previous waiting task that is referred by the previous pointer Prev_TN of the removing task to refer to the next waiting that is referred by the next pointer Next_TN of the removing task. This is written in line 24 up to line
27. It must also be determined whether it is a local task or it is a remote task. In the second action, the next waiting task is set to refer to the previous waiting task and is written in line 28 up to line 32. It is the same as in the first action, only in the reverse direction. As last, the previous pointer Prev_TN and the next pointer Next_TN of the removing task are set to NIL. This is done in line 33 and in line 34.

4.5.3.2 Removing a task from the timeout waiting list

The behavior for removing a local task that is waiting for a timeout time from the timeout waiting list behaves the same as for removing a local task from the local resource waiting list as described in the previous subsection 4.5.3.1. The difference is that only local tasks can be awaited for a timeout time. Therefore, the TL, the register First_TO_TN and the register Last_TO_TN are needed. In figure 4.5.3.2, its behavior description is shown.

```
1. IF (First_TO_TN == Last_TO_TN) // First situation.
2. First_TO_TN = NIL;
3. Last_TO_TN = NIL;
4. ELSE
5. IF (First_TO_TN == DEST_TN) {
   6. First_TO_TN = TL(DEST_TN).Next_TN;
   7. TL(DEST_TN).Next_TN = NIL;
   8. TL(First_TO_TN).Prev_TN = NIL;
   9. }
10. ELSIF (Last_TO_TN == DEST_TN) {
   11. Last_TO_TN = NIL;
   13. TL(Last_TO_TN).Next_TN = NIL;
   14. }
15. ELSE {
   18. TL(DEST_TN).Next_TN = NIL;
   20. }
```

Figure 4.5.3.2. Behavior description of "Removing task from TO waiting list".

4.5.3.3 Removing a task from the TaskQ/TL

After the destination task has been removed from a waiting list or its activity has been cancelled, the task must be removed from the TaskQ by setting its field "Runnable" to 0b, like TaskQ(DEST_TN).Runnable = 0b. Because the task list TL is needed for keeping the administration of all local tasks, it must also be updated by setting its state to tsIDLE, like TL(DEST_TN).State = tsIDLE.

4.5.4 CHANGE_PRIO

With the instruction CHANGE_PRIO(DEST_TN, NEW_PRIO) a local task can change its priority or the priority of another local task. When a task is running and its priority is changed to a lower priority, the TaskQ can block this task. However, if the task's priority is changed to a higher priority, the TaskQ can set the task running if it was runnable.

As inputs, the TN and the status word SW are needed to be able to report what went wrong. In addition, the instruction code IC, Parameter 1 for holding DEST_TN of the task from which the priority has to be changed and Parameter 2 for holding NEW_PRIO is needed. The behavior description of this instruction is shown in figure 4.5.4 on the next page.
If the task number DEST_TN of the task from which the priority has to be changed is not in use, its state is equal to tsIDLE. Therefore, the SW will be updated indicating what went wrong for sending back to the local task. In this case, the TID-bit is set to 1b indicating the TN does not exist anymore, it is dead. Else, the priority of this task will be changed to NEW_PRIO.

### 4.5.5 GET_PRIO

With the instruction GET_PRIO(DEST_TN) the priority of an arbitrary local task can be obtained. As inputs, the TN, the status word SW, the instruction code IC and Parameter I for holding DEST_TN from which the priority has to be requested will be obtained. The behavior description of this instruction is shown in figure 4.5.5.

If the task number DEST_TN of the target task is not in use, the SW will be updated indicating what went wrong for sending back to the local task. Else, the priority of this task is fetched from the TaskQ for sending back to the local task sending the request.

### 4.5.6 SLEEP

With the instruction SLEEP(DEST_TN) a local task can put itself or another existing task asleep unconditionally. A task is forced to block. Therefore, its current activity must be cancelled immediately. When a task puts itself asleep, only its state on the task list TL has to be changed to tsBLOCKED for keeping the administration. On the TaskQ, the field “Runnable” is then changed to 0b to block the task. When a task puts another task asleep, not only the target task is blocked also its current activity has to be cancelled. This activity can be waiting for accessing to a remote resource, waiting for a remote message, waiting for accessing to a local resource and or waiting for a timeout time.

As inputs, the TN, the status word SW, the instruction code IC and Parameter I for holding DEST_TN from which it will be put asleep are needed.
The behavior description for this instruction is the same as for the instruction END_TASK, which is already described in subsection 4.5.3 on page 36. The only difference is that the state of the target task on the TL is changed to tsBLOCKED using the code ‘TL(DEST_TN).State = tsBLOCKED’, because the target task is blocked unconditionally and not ended as described in subsection 4.5.3.3 on page 39.

4.5.7 AWAKE

With the instruction AWAKE(DEST_TN) a blocked task can be forced to wake up. It is a kind of ‘cancel-request’

As inputs, the task number TN, the status word SW to able to report what went wrong, the instruction code IC and Parameter 1 for holding DEST_TN on which it will be wake up are needed.

The behavior description for this instruction is the same as for the instruction END_TASK, which is already described in subsection 4.5.3 on page 36. The only difference is that the result must be sent back to the blocked after the SW has been updated to indicate what went wrong. In this case, the RC-bit on the SW is set to 1b indicating the request has been cancelled. After that, the blocked task must set runnable and not ended as described in subsection 4.5.3.3 on page 39. Therefore, the subroutine “Removing task from TaskQ/TL” in figure 4.5.3 must be replaced by the subroutine “Sending Result To Local Task” for sending the result back and the subroutine “Wake Up” waking up the task. The behavior of these subroutines are already described in subsection 4.4.1 on page 32 and in subsection 4.4.2 on page 32 respectively.

4.5.8 DELAY

With the instruction DELAY(DEST_TN, DELAY_TIME) a local task can put itself or another task asleep for the given delay time while the instruction SLEEP(DEST_TN) puts itself or another task asleep unconditionally. When a task puts itself asleep for the given delay time, its state on the task list TL is then changed to tsWTO and on the TaskQ its field “Runnable” is set to 0b to block the task. After that, the target task will be inserted into the timeout waiting list. However, if the task puts another task asleep for the given delay time, also the current activity of the target task must be cancelled immediately. This activity can be waiting for accessing to a remote resource, waiting for a remote message or waiting for accessing to a local resource.

Another variant of the instruction DELAY() is the instruction TIMEOUTO. With the instruction TIMEOUTO() a task can request for a given timeout time without blocking itself or another task. This instruction will be described in subsection 4.5.8.2 on page 42.

As inputs, the task number TN, the status word SW, the instruction code IC, the Parameter 1 for holding DEST_TN on which it will be put asleep and the Parameter 2 for holding DELAY_TIME are needed. The behavior description for this instruction is shown in figure 4.5.8.
If the task number DEST_TN of the target task is not in use, TL(DEST_TN).State = tsIDLE, the status word SW is updated to indicate what went wrong for sending back to the local task. In this case, the TID-bit is set to 1b indicating the task does not exist anymore, it is dead.

If the target task does in use and it is waiting for accessing to a remote resource or for a message, a cancel-wait (CW) message including eventually the remote index number (RIN) is sent onto the network without waiting for its acknowledgement. But, if the target task had sent a remote-wait-command (RWC), then the RIN is still not known on the TL. Therefore, the RIN cannot be included, but just a dummy. After that, the subroutine "Insert task into TO waiting list" is called to insert the target task into the timeout waiting list. Its behavior description is described in subsection 4.5.8.1. When waiting for accessing to a local resource the subroutine “Removing task from resource waiting list” is called for removing the task from the local resource waiting list. Its behavior is already described in subsection 4.5.3.1 on page 37. In addition, the subroutine “Insert task into TO waiting list” has to be called for inserting the task into the timeout waiting list.

4.5.8.1 Inserting a task into the timeout waiting list

When a task has to be inserted into the timeout waiting list, there are four possible situations and are shown in figure 4.5.8.1a.

In the first situation, the timeout waiting list is empty while in situation 2 up to situation 4 there is at least one waiting task present on the list. In figure 4.5.8.1b on the next page, its behavior description is shown.
1. TL(DEST_TN).State = tsWTO;
2. TaskQ(DEST_TN).Runnable = 0;
3. IF (First_TO_TN == NIL) {
   4. TL(DEST_TN).Finish-time = current_TO_time + TO_TIME;
   5. First_TO_TN = DEST_TN;
   6. Last_TO_TN = DEST_TN;
}
8. ELSIF ((DELAY_TIME) ≤ (TL(First_TO_TN).Finish-time – current_TO_time)) {
   9. TL(DEST_TN).Finish-time = current_TO_time + TO_TIME;
   10. TL(First_TO_TN).Prev_TO_TN = DEST_TN;
   11. TL(DEST_TN).Next_TO_TN = First_TO_TN;
   12. First_TO_TN = DEST_TN;
}
14. ELSIF ((DELAY_TIME) ≥ (TL(Last_TO_TN).Finish-time – current_TO_time)) {
15. TL(DEST_TN).Finish-time = current_TO_time + TO_TIME;
16. TL(Last_TO_TN).Next_TO_TN = DEST_TN;
17. TL(DEST_TN).Prev_TO_TN = Last_TO_TN;
18. Last_TO_TN = DEST_TN;
}
20. ELSE {
21. Temp_TN = First_TO_TN;
22. WHILE ((DELAY_TIME) ≥ (TL(Last_TO_TN).Finish-time – current_TO_time))
23. \[ Temp_TN = TL(Temp_TN).Next_TO_TN \]
24. TL(DEST_TN).Finish-time = current_TO_time + TO_TIME;
25. TL(TL(Temp_TN).Prev_TO_TN).Next_TO_TN = DEST_TN;
27. \[ Line \text{nr. 4} \]
28. TL(Temp_TN).Prev_TO_TN = DEST_TN;
29. TL(DEST_TN).Next_TO_TN = Temp_TN;
30. }

Figure 4.5.8.1b. Behavior description for inserting a task into the TO waiting list.

Firstly, the task’s state on the TL is changed to tsWTO (waiting-for-timeout) and on the TaskQ the field “Runnable” is set to 0b to block the task. Next, the task is inserted into the timeout waiting list. This is done from line 3 to line 30.

As shown on figure 4.5.8.1a on the previous page, a task can be inserted into the timeout waiting list in four situations. In the first situation, the waiting list is empty. Therefore, both the pointer First_TO_TN and the pointer Last_TO_TN are set to DEST_TN to refer to the beginning and the end of the list respectively, after the finish-time for the target task is calculated. This is done in line 3 to line 7.

In the second situation up to the fourth situation, there is at least one waiting task present on the waiting list. In these situations, the finish-time for the target task must be calculated. The calculation is depended on the current time of the timeout timer. Because this timer runs continuously, the calculation must also be done continuously. Else, a delay will be occurred, which makes the calculation incorrectly. This problem is depicted in figure 4.5.8.1c on the next page.
For example, the finish-time of the target task must be calculated, which after it will be compared with the finish-time of the reference task. The reference task is one of the tasks on the timeout waiting list. If the finish-time of the target task is smaller than REF, then the target is inserted into the timeout waiting list in front of the reference task. If it is equal or greater, the task is inserted behind the reference task. Suppose, the calculated finish-time is smaller than the REF and that is equal to CALC. Now the calculation is done, the CALC is just smaller than the REF. After the CALC is stored on a temporal register, which after the comparison will be made, the time of the timeout timer CT has been increased and is e.g. equal to new CT. It means, the calculated finish-time is not equal to CALC anymore. This value can be equal to new_CALC, which is greater or equal to REF. In the last case, the target task must be inserted behind the reference task. To avoid this problem or to keep this problem small, the calculation of the finish-time will be made at the moment the comparison is made. When storing the finish-time, it is calculated again.

In the second situation, the inserting task has the lowest finish time and is therefore inserted at the beginning of the list. Therefore, two new connections are made between the inserting task and the first task on the waiting list according to line 10 and line 11. In line 12, the pointer First_TO_TN is set to refer to the inserting task.

In the third situation, the inserting task has the highest finish time and is therefore inserted at the end of the waiting list. This is done in line 14 to line 19. In line 16 and in line 17 two new connections are made between the last task on the waiting list and the inserting task, which becomes the new last task on the waiting list. Therefore, the pointer Last_TO_TN is set to refer to the inserting task in line 18.

In the fourth situation, the inserting task has a finish time that lies in between other waiting tasks. Therefore, this location has to be traced. This is done in line 21 up to line 23. Firstly, a copy of the pointer First_TO_TN is made by line 21 using the temporal pointer Temp_TN. If finish time of the inserting task is greater than the first waiting task that is referred by temporal Temp_TN, then pointer is moved to the next waiting task by loading it with the value of the next pointer Next_TN of the first waiting task. It will be repeated until the finish time of the inserting task is smaller than or equal to the finish time of the found waiting task. This is done in line 22 and line 23. For inserting the task two new connections are made between the inserting task and the next waiting task that is referred by the next pointer Next_TN of the found waiting task with the task number that is held by Temp_TN. This is done in line 25 and in line 26. For making, the new connections between the inserting task and the found waiting task this is done in line 28 and in line 29.
4.5.8.2 TIMEOUT

With the instruction TIMEOUT(DEST_TN, TIMEOUT_TIME) a task can request a timeout for itself or for another task without blocking. The intention of this instruction, a task can be forced wake up after the requested timeout time has been expired when it has waited too long for something.

As inputs, the task number TN, the status word SW, the instruction code IC, the Parameter 1 for holding DEST_TN on which it will be put asleep and the Parameter 2 for holding TIMEOUT_TIME are needed. The behavior description for this instruction is shown in figure 4.5.8.2.

Figure 4.5.8.2. Behavior description of the instruction TIMEOUT().

The instruction TIMEOUT() is almost the same as the instruction DELAY(). The only difference is that the target is running and not waiting for something. Therefore, the field “Finish-time” on the TL can be updated immediately after which the subroutine “Insert task into TO waiting list” is called to insert the target task into the timeout waiting list. Its behavior is already described in subsection 4.5.8.1 on page 42.

4.5.9 CREATE_RESOURCE

On each MμP, the TCU has to administrate up to 128 resources. Before using, they must be created and removed if they are not needed anymore. For creating, there are three instructions defined as shown in figure 4.5.9a.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_SEM</td>
<td>RN</td>
<td>INIT_COUNT</td>
<td>Create a short or a long semaphore</td>
</tr>
<tr>
<td>CREATE_MB</td>
<td>RN</td>
<td>NR_SLOTS</td>
<td>Create a mailbox with the given number of slots</td>
</tr>
<tr>
<td>CREATE_PIPE</td>
<td>RN</td>
<td>POINTER</td>
<td>Create a pipe with a pointer that refers to location of the block data in the main memory</td>
</tr>
</tbody>
</table>

Figure 4.5.9a. Instructions for creating resources.

For creating a new semaphore, the instruction CREATE_SEM (RN, INIT_COUNT) is defined. There are two parameters needed, the resource number RN of the new semaphore and the INIT_COUNT. Because both the short semaphore as the long semaphore can be created, the second parameter INIT_COUNT is needed, which is 32-bits in width. If the MSB of INIT_COUNT is equal to 0, a short semaphore will be created and a long semaphore will be created if the MSB is equal to 1. On a short semaphore, up to $2^{12} = 4096$ units can be used. On a long semaphore, up to $2^{32} = 4.3G$ units can be used. Because the lowest 31 bits of parameter INIT_COUNT are not in use, these bits can be used for initializing the number of units. It means, for a short semaphore, the number of units can be initialized with up to $(2^{12} - 1 =) 4095$. For a long semaphore, it can be initialized with up to $2^{31} - 1$.

For creating a new mailbox, the instruction CREATE_MB (RN, NR_SLOTS) is defined. As for the semaphore, also two parameters are needed, the resource number RN of the new mailbox and the number of slots the new mailbox must be fitted with. For the moment, a mailbox can be fitted with up to 16 slots. When a mailbox with more than 16 slots is needed, a long mailbox must be created. Therefore, the mailbox will be kept in the main memory. However, for the moment, only standard mailboxes are used that have up to 16 slots.
For creating a new pipe, the instruction CREATE_PIPE (RN, POINTER) is defined. For the pipe, also two parameters are needed, the resource number RN and the pointer (or the start address) that refers to the block data, which is stored in the main memory. The pipe is a buffer with a FIFO structure, on which block data can be produced/written and consumed/read. The pipe is very flexibly. It has a fixed length, but the length of the block data is variable. As long as there are enough spaces on the pipe, the whole block data can be stored on the pipe. Else, only the first part that can be fitted is stored on the pipe. The rest is rejected. The other advantage is that the whole block data is not needed to consume at once. For the moment, only a concept of the pipe will be described because of the complexity of the pipe.

As inputs, the task number TN and the status word SW are needed. Also, the instruction code IC, the result return address RRA, Parameter 1 for holding RN and Parameter 2 for holding the type and initialized count of the semaphore, the number of slots for a mailbox or the pointer for the pipe are needed.

For initialization, 128 short semaphores are created. When creating a resource, it means a resource will be changed from a certain type to another type.

When creating a resource, the availability of the corresponding resource data blocks on the RD must be checked firstly. If the required block(s) is/are available, tasks that are waiting for the current resource must be restart. They must be made runnable. After that, a new resource can be created. If not available, the SW is updated indicating what went wrong for sending back to the local task sending the request.

In the next subsections 4.5.9.1 up to 4.5.9.7, these three instructions are described in more details.

### 4.5.9.1 Create Semaphore

For creating a new semaphore, the instruction CREATE_SEM (RN, TYPE) is used. There are two parameters required. The parameter RN indicates the resource number of the creating resource and the parameter TYPE indicates whether a short or a long semaphore will be created. Therefore, the parameter TYPE is equal to 0 and to 1 respectively. For the administration, all fields on the RL can be needed. For a long semaphore, only the field "Last Operation" of Block 1 is used and is set to 1b to indicate a long semaphore. For a short semaphore, this field is set to 0b. The behavior description is shown in figure 4.5.9.1.

![Behavior description for the subroutine "Create Semaphore".](image-url)
There are two types of semaphores, a short and a long semaphore. For a long semaphore, a resource data block on the RD is needed for storing the number of units. So, the availability of the required resource data block must be checked by calling the subroutine “Checking availability of required resource data blocks”. Its behavior is described in subsection 4.5.9.2. If it is not available, the SW is updated to indicate what went wrong for sending back to the local task sending the request. Therefore, the ERR-bit is set to 1b indicating an error has been occurred. Because a resource is requested to create, the RAE-bit is set to 1b and the DNA-bit is set to indicate the required resource data block is not available.

When creating a new resource, it means the current resource will be removed. Therefore, if there are tasking waiting for the current resource, they must be restarted. If so, the subroutine “Restarting waiting tasks” is called for execution. Its behavior is described in subsection 4.5.9.5 on page 51.

For updating the resource administration, the field “RT” is set to 0b indicating it is a semaphore. If the MSB of the second parameter INIT_COUNT is equal to 0b, it means a short semaphore is requested. Therefore, the field “Last_Operation” is set to 0b and the field “Nr_Units” is set to the initialized count, which is held by the lowest 12 bits of the parameter INIT_COUNT. In addition, if the MSB is equal to 1b, it means a long semaphore is requested. Therefore, the field “Last_Operation” is set to 1b and the resource data block on the RD is set to the initialized count which is held by the lowest 31 bits of the parameter INIT_COUNT. Because the MSB of INIT_COUNT is equal to 1b when creating a long semaphore, this bit must be reset firstly before initializing.

### 4.5.9.2 Checking the availability of the required resource data blocks

Before creating a new resource, the availability of required resource data block(s) on the RD must be checked. If it is available, the AVAIL-bit is set to 1b, else to 0b. In figure 4.5.9.2, the behavior of the subroutine “Checking availability of required resource data blocks” is described.

![Diagram](image)

**Figure 4.5.9.2. Behavior description of “Checking availability of required resource data blocks”.

Firstly, it must be noted that this subroutine will be used for creating new semaphores, mailboxes and pipes.

Before checking the availability of the corresponding resource data blocks on the RD, the AVAIL-bit is set to 0b indicating the required block(s) is/are not available. After that, it must be determined for which resource the availability must be checked.
For a short semaphore, the TYPE = 0, the AVAIL-bit can be set to 1b immediately indicating the required resource data block is available. Besides, a short semaphore does not need a resource data block on the RD.

For a long semaphore, the TYPE = 1b, it does need one resource data block on the RD. To be able to determine whether the corresponding block on the RD is available or not, the type of the current resource must be checked. If the current resource is not a short semaphore, it means the corresponding block on the RD was in use by the current resource. Therefore, the AVAIL-bit can be set to 1b, because it is available. Else, the subroutine "Checking Previous Resources" is called to check the availability of the corresponding resource data block. Its behavior is described in subsection 4.5.9.3 on the next page.

For a pipe, it behaves the same as for the long semaphore. The difference is that the TYPE is not needed to check.

For a mailbox, it behaves just like for the pipe. If the corresponding block on the RD is not available, AVAIL = 0b after the subroutine "Checking Previous Resources" has been called, no new mailbox can be created. If it is available, AVAIL = 1b, the subroutine "Checking Required Blocks" is called to determine the availability of the required blocks (number of slots). Its behavior is described in on page 50. Because no new mailbox can be created if the required blocks are not available, the AVAIL-bit is reset before calling the subroutine "Checking Required Blocks" to check for the availability of the required number blocks.
4.5.9.3 Checking the previous resources

If the current resource is a short semaphore, it is not sure whether the corresponding resource data block is in use or not. For checking this, the type of the current resource must be checked. Because a mailbox can be fitted with up to 16 slots, 16 resource data blocks, up to 15 previous resources can be checked. The situation is shown in figure 4.5.9.3a.

![Figure 4.5.9.3a. Principle of checking the availability of the corresponding resource data block.](image)

If the concerning resource, RL(RN - i), is not a mailbox, it is sure that the corresponding block of the current resource is available. Therefore, the AVAIL-bit is set to 1b indicating the corresponding resource data block is available. However, if it is a mailbox and the number of slots is equal to the register i, it also means the corresponding resource data block is available. In figure 4.5.9.3b, the behavior of the subroutine “Checking Previous Resources” is described.

```
1. i = 1;
2. WHILE (((RN - i) >= 0) OR (i < 16) OR (AVAIL == 0)) {
3.    IF ((RL(RN - i).RT == 00) OR (RL(RN - i).RT == 10))
4.        AVAIL = 1;
5.    ELSIF (RL(RN - i).Nd_Slots <= i)
6.        AVAIL = 1;
7.    i++;
8. }
```

![Figure 4.5.9.3b. Behavior description of “Checking Previous Resources”](image)

Because up to 15 previous resources must be checked, the register i = [1..15]. The behavior of this subroutine consists of one while-loop. As long as the AVAIL-bit is not equal to 0b or the resource number 0 is reached, (RN - i) >= 0, or as long as i < 16, the check must be executed. If the resource with the number (RN - i) is a semaphore, the RT = 00b, or a pipe, the RT = 10b, the AVAIL-bit is set to 1b. Else, it is a mailbox. If the number slots of the mailbox is less or equal to the register i, also the AVAIL-bit is set to 1b. In all other cases, the corresponding block of the current resource is in use.
4.5.9.4 Checking the required resource data blocks

When creating a new mailbox, the required number of slots must be available. After the subroutine "Checking Previous Resources" has been called and the corresponding resource data block RD(RN) is available, the required number of resource data blocks on the RD must be checked for availability. This is depended on the required number of slots of the creating mailbox. This number must be started from the corresponding block RD(RN), which belongs to the creating resource. Else, the requested mailbox cannot be created. In figure 4.5.9.4a, the principle is shown.

![Figure 4.5.9.4a. Principle of Checking for the required resource data blocks.](image)

The requirement is that block with the number RN up to block with the number (RN + Nr_Slots - 1) must be available before creating a new mailbox. If one of these blocks is not available, the requested mailbox cannot be created.

In figure 4.5.9.4b, the behavior of the subroutine “Checking Required Blocks” is described.

```
1. index = 1;
2. Not_Avail = 0;
3. WHILE ((index <= Nr_Slots) OR (Not_Avail == 0)) {
4.   IF ((RL(RN + index).RT == 00) AND (RL(RN + index).Last_Operation == 0)) {
5.     index++;
6.   } ELSE
7.   Not_Avail = 1;
8. }
9. AVAIL = !Not_Avail;
```

![Figure 4.5.9.4b. Behavior description of “Checking Required Blocks”.](image)

Because from block number (RN + 1) up to block number (RN + Nr_Slots - 1) must be checked, the temporal register index is set to 1. To indicate the required number of blocks is not available, the bit Not_Avail is used. If it is equal to 1b, it means the number of required blocks is not available. This is the case, in which one of the checked blocks is occupied by another resource. Initially, this bit is set to 0b.

As shown in figure 4.5.9.4b, the subroutine “Checking Required Blocks” consists of a while-loop for checking the availability of the required number of slots. As long as the register index is smaller than or equal to the required number of slots or the bit Not_Avail is equal to 0b, the while-loop will be executed. If the resource corresponding to the checked block is a short semaphore, it means this block is available. Therefore, the resource type (RT) of the resource with the resource number (RN + index) on the RL is equal to 00b and the field “Last_Operation” is equal to 0b. If it is a short semaphore, the register index is increased by one for checking the next block. Else, the bit Not_Avail is set to 1b indicating the required number of blocks is not available.
For setting the result, the reverse value of the bit Not_Avail is assigned to the AVAIL-bit. If the Not_Avail-bit is equal to 0b, it means the required number of blocks is available. Therefore, the AVAIL-bit is equal to 1b.

4.5.9.5 Restarting waiting tasks

Initially, the resource list RL is initialized with 128 short semaphores. When a mailbox with three slots has been created with the resource number 5, it means resource data block number 5 up to number 7 are in use by this mailbox. Therefore, on the resource with resource number 6 and 7 no new long semaphore, mailbox or pipe can be created. Only new short semaphores can be created. It does not matter whether these two short semaphores will be used or not, there are always 128 resources created on the RL. Therefore, tasks that are waiting for the old resource must be restarted when a new resource is created. In figure 4.5.9.5, the behavior of the subroutine “Restarting Waiting Tasks” is described.

If the first waiting task is a local task, its SW is set to indicate what went wrong. In this case, the RNE-bit is set indicating the requested resource is not existed anymore. Because the SW of a local task is stored on the TL while the SW of the remote task is stored on the RRTL, setting the SW must be done separately. For sending the result back, the subroutine “Sending Result To Local Task” is called. Its behavior is already described in subsection 4.4.1 32. After that, the task will wake up by calling the subroutine “Wake Up”. Its behavior is already described in subsection 4.4.2 on page 32. For waking up the next waiting task, the temporal register Temp_Next is loaded with the value of the next pointer Next_TN of the removed task. As last, the pointer Next_TN and the pointer Prev_TN of the removed task are set to NIL indicating the beginning and the end. If the new value of the register Temp_Next is equal to NIL, it means there are no more waiting tasks. Else, the subroutine is recalled to wake up the next waiting task after the register First_TN is loaded with the value of the temporal register Temp_Next.

For a remote task, the cancel-wait (CW) message will be sent onto the network towards the remote task sending the request, after the SW has been set. However, the task’s block may not be cleaned up yet. It must be waited until the end-wait-acknowledgement (EWA) message has been received. Therefore, its block’ state is set to rsWEWA.
4.5.9.6 Create Mailbox

For creating a new mailbox, the instruction CREATE_MB (RN, NR_SLOTS) is used. There are two parameters required. The parameter RN indicates the resource number of the new mailbox and the parameter NR_SLOTS indicates the number of required slots (data resource blocks) the new mailbox must be fitted. The behavior description is as shown in figure 4.5.9.6.

**Figure 4.5.9.6. Behavior description for the subroutine “Create Mailbox”.**

Firstly, the subroutine “Checking availability of required resource data blocks” is called for checking the availability of the required number of data resource blocks depended on the required number of slots. Its behavior is already described in subsection 4.5.9.2 on page 47. If it is not available, the SW is updated to indicate what went wrong. In this case, the DNA-bit is set to 1b indicating the required resource data blocks are not available. Else, a new mailbox can be created. However, before creating it must be determined whether there are tasks waiting for the old (current) resource. If there are waiting tasks, RL(RN).First_TN = NIL, the subroutine “Restarting Waiting Tasks” is called for waking up the waiting tasks. Its behavior is already described in subsection 4.5.9.5 on page 51. After that, the new mailbox can be created by setting the field “RT” to 0b, the field “Nr_Slots” to NR_SLOTS, the field “Nr_Used” to 0, the field “Next_Message” to 0 and the field “Last_Operation” to 0b.
4.5.9.7 Create Pipe

For creating a new pipe, the instruction CREATE_PIPE (RN, POINTER) is used. There are two parameters required. The parameter RN indicates the resource number of the new pipe and the parameter POINTER indicates the start address of the buffer in the main memory where the block data can be written and read. The behavior description is as shown in figure 4.5.9.7a.

Firstly, the subroutine “Checking availability of required resource data blocks” is called for checking the required resource data block on the RD. Its behavior is already described in subsection 4.5.9.2 on page 47. If it is not available, the SW is updated to indicate what went wrong. In this case, the DNA-bit is set to 1b. Else, a new pipe can be created. However, it must be determined firstly whether there are tasks waiting for the old (current) resource. If there are waiting tasks, RL(RN).First_TN = NIL, the subroutine “Restarting Waiting Tasks” is called for waking up the waiting tasks. Its behavior is already described in subsection 4.5.9.5 on page 51. After that, the new pipe can be created by setting the field “RT” to 10b, the field “Last_Operation” to 0b and the corresponding resource data block to POINTER. As last, the read and the write pointers of the pipe are set to refer to the first word of the block data POINTER + 3. POINTER is a start address that refers to the buffer of the pipe as shown in figure 4.5.9.7b.

Because the length of each word is 32 bits in width, a data block with a length of up to $2^{32} = 4G$ bytes can be transferred. Further, a read and a write pointer are needed. They are located on the address (POINTER + 1) and (POINTER + 2) respectively. With these pointers, a number of small block data can be written for transmission. Alternatively, a block data can be read for several times, because a small amount of the block data can be read.
The read pointer refers to the location where the first available block data in the pipe can be written. The write pointer refers to the location on the pipe where the next block data can be written.

For accessing to a pipe the field “Last Operation” on the RL indicates whether the block data is empty (= 0b) or full (= 1b). As long as “Last Operation” = 0b a new block data can be transferred. For a pipe, the fields “Nr_Slots”, “Nr_Used” and “Next_Message” are not in use.

4.5.10 Removing a resource

Because the resource list RL is initialized with 128 short semaphores, the instructions CREATE_SEM, CREATE_MB and CREATE_PIPE are used to create a new resource. When a resource is not needed anymore, the instruction CREATE_SEM can be used to change the resource to a short semaphore. This is the principle for creating a new resource and is the principle for removing a resource. So, for removing a resource, the instruction CREATE_SEM, the instruction CREATE_MB and the instruction CREATE_PIPE can be used.

4.5.11 Writing to a resource

When two tasks want to communicate with each other, they have to use the communication resources mailbox, the pipe and or the semaphore. The resource semaphore will be used for general synchronization purposes. When accessing to a semaphore, the requested number of units will be added after which the task will be restarted.

With a mailbox, a task can send a 32-bits message word to the mailbox, after which the other task can read the message. For the mailbox the memory structure FIFO is used. When there are a number of messages present on the mailbox and a task wants to read a message, then the first incoming message will be read out.

When a task wants to transfer a block data to another local task or to a remote task, the sending task has to place the block data into the main memory. The first word of the block data indicates the length of the block data. After that, the actual data is stored. The sending task has to place the 32-bits pointer (start address) of the block data into a pipe. Because the width of the memory is 32 bits, a block data with a length of up to $2^{32} \approx 4B$ bytes can be transferred.

For writing data to a resource, two different instructions are used for each resource type. One instruction is without condition and one instruction is with condition. The difference is that when accessing to a resource conditionally and the requested resource is full, the task sending the request will not be blocked.

As shown in table 3.7 on page 14, there are five different instructions available for writing to a resource. Two instructions for writing to a mailbox, two instructions for writing to a pipe and only one instruction for writing (increasing) to a semaphore. When writing to a semaphore and the number of units is not available, the task will get an error message. These instructions are shown in figure 4.5.11a.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE_S</td>
<td>RN</td>
<td>NR UNITS</td>
</tr>
<tr>
<td>WRITE_MB</td>
<td>RN</td>
<td>MESSAGE</td>
</tr>
<tr>
<td>WRITEC_MB</td>
<td>RN</td>
<td>MESSAGE</td>
</tr>
<tr>
<td>WRITE_P</td>
<td>RN</td>
<td>WRITE POINTER</td>
</tr>
<tr>
<td>WRITEC_P</td>
<td>RN</td>
<td>WRITE_POINTER</td>
</tr>
</tbody>
</table>

Figure 4.5.11a. Instructions for writing data to a resource.

For each instruction, there are two parameters needed. The first parameter is the resource number RN, which is needed for each instruction. For the semaphore, the second parameter NR_UNITS is needed to indicate the number of units a task wants to increase. Comparing with the standard semaphore, the write-instruction is equal to the increasing function $V(NR\_UNITS)$. For further information, see subsection 4.5.16 “Execute semaphore” on page 65.
For the mailbox, the second parameter MESSAGE is equal to the message or to the result word that a task wants to send. For further information, see subsection 4.5.14 "Execute mailbox" on page 57.

For the pipe, the second parameter WRITE_POINTER is equal to the start address that refers to the location of the buffer on which the block data is stored for transferring. As for the buffer of the pipe, the first word indicates the length of the sending block data. For further information, see subsection 4.5.15 "Execute pipe" on page 65.

Standard, the width of the parameter RN is 7-bits in bits. Internally, the TCU uses a register called Parameter 1 for storing the resource number RN of the requested resource, which is 32 bits in width. This means that the other 25 bits are not in use. When a task wants to access to a resource on a remote MIP, the address or the ID of the remote MIP is needed for the communication. Therefore, 24 bits can be used for storing the address of the remote MIP and the other one bit is used to indicate whether it is a resource addressing or a task addressing as shown in figure 4.5.11b.

<table>
<thead>
<tr>
<th>MIP address</th>
<th>TN/ RN</th>
<th>Task / Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000_ = local</td>
<td>task / resource number</td>
<td>0 = resource</td>
</tr>
<tr>
<td>&gt;00000000_ = remote</td>
<td></td>
<td>1 = task</td>
</tr>
</tbody>
</table>

Figure 4.5.11b. Addressing of Parameter 1.

If the bit number 0 is 1b, it indicates the task addressing and 0b it indicates the resource addressing. Therefore, bit number 7 down to bit number 1 will indicate the task or the resource number. The highest three bytes (bit number 31 up to bit number 8) are used for indicating the MIP address. When the requested resource or the target task is locally, the MIP address is equal to zero. Else, it is greater than zero. However, the MIP address may also be local MIP network address.

4.5.12 Reading from a resource

After the data is written to a resource, another task can read the data from the concerning resource. For reading from a resource, there are six instructions available. These instructions are shown in figure 4.5.12.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ S</td>
<td>RN</td>
<td>NR UNITS</td>
</tr>
<tr>
<td>READC S</td>
<td>RN</td>
<td>NR UNITS</td>
</tr>
<tr>
<td>READ MB</td>
<td>RN</td>
<td>---</td>
</tr>
<tr>
<td>READC MB</td>
<td>RN</td>
<td>---</td>
</tr>
<tr>
<td>READ P</td>
<td>RN</td>
<td>READ POINTER</td>
</tr>
<tr>
<td>READC P</td>
<td>RN</td>
<td>READ POINTER</td>
</tr>
</tbody>
</table>

Figure 4.5.12. Instructions for reading data from a resource.

For reading from a semaphore, a second parameter is needed for decreasing the number of units. Comparing with a standard semaphore, the read instruction is equal to the function P(NR_UNITS). When reading from a semaphore, which content is zero or smaller than the requested number of units, this task will be blocked. For reading from a semaphore conditionally, it will not be blocked. The status word SW is then updated to indicate what went wrong after which the task sending the request is made runnable.

When reading from a mailbox, the message word will be read and placed onto the R-register to send back to the local task. Alternatively, it will be sent onto the network if a remote task had sent the request. For further information, see section 4.5.13.1 "Execute mailbox" on page 57.

When reading from a pipe, the reading task will need a buffer on which the reading block data must be stored. Therefore, the second parameter WRITE POINTER is needed to refer to the location of the buffer in the main memory. For further information, see section 4.5.15 "Execute pipe" on page 65.
4.5.13 Execute Resource Request

As shown in figure 4.2b on page 18, the block "Execute Resource Request" executes all resource requests coming from the block "Local Resource Request Handling" or from the block "Remote Request Handling". These two blocks are described in section 4.6 on page 69 and in chapter 5 on page 76 respectively. After execution, the result can be sent back to the local task via the block "Result Transmission" or to the remote task via the Network Manager. For the local task, the result contains the TN, the SW, the WR_SW-bit, the RRA and the optional result word. For the remote task, the result contains only the TN, the SW and the WR_SW-bit.

Preventing that during the execution many accesses have to be made to the task list TL or to the remote-request-task list RRTL, the request will be stored onto the resource-instruction (RI) register, which data structure is shown in figure 4.3.9 on page 30.

When the bit AVAIL-bit on the RI-register is equal to 0b, it means that the block "Execute Resource Request" is stand-by. Therefore, a new resource can be placed onto the RI-register. When AVAIL = 1b, no new request may be placed, because this block is executing a request. After execution, this bit is set to 0b.

TN is the task number of the task sending the request. When the TN lies between 00000H and 03FBF H, the request is coming from a remote task. When the TN lies between 03FC0 H and 03FFE H, the request is coming from a local task. Else, it is not a task at all. It is coming from one of the four events. In the last case, no result is needed to send back. The status word SW can be read before the execution or it can be written after the execution.

As shown in figure 4.5.11b on page 55, Parameter 1 is used for holding the resource number of the requested local resource including its Mailbox address. When the Mailbox address is equal to zero it is a local request. If it is required to read the SW, the bit RD_SW = 1b and if it is required to update (write) the SW after execution the bit WR_SW = 1b. The field instruction code IC will indicates whether it is a read or a write-operation. As last, the field "Parameter 2" is used for holding the data for the write-operation or for holding the result word for sending back to the local task or to the remote task.

The behavior description for the block "Execute Resource Request" is shown in figure 4.5.13.

Figure 4.5.13. Behavior description of "Execute Resource Request".

If the resource number RN of the requested resource does not match with the IC, the status word SW is updated to indicate what went wrong. In this case, the IRN-bit is set to 1b. After that, the result will be sent back to the task sending the request. If it is a local task, the subroutine "Sending Result To Local Task" is called. Its Behavior is already described in subsection 4.4.1 on page 32. For a remote task, the subroutine "Sending Result To Remote Task" is called. Its behavior is described in subsection 4.5.13.1 on the next page. If the request is coming from one of the events, no result is needed to send back. For further information, see chapter 6 "Event Handling" on page 91.

When the RN matches with the IC, the concerning subroutine is called to execute the request. For accessing to a mailbox, the subroutine "Execute Mailbox" is called for execution. Its behavior description is described in subsection 4.5.14 on page 58. For accessing to a pipe, the subroutine "Execute Pipe" is called for execution. Its
behavior description is described in subsection 4.5.15 on page 65. For accessing to a semaphore, the subroutine "Execute Semaphore" is called. Its behavior description is described in subsection 4.5.16 on page 65.

### 4.5.13.1 Sending result back to a remote task

After a result word has been produced when accessing to a local resource or after the SW has been updated to indicate what went wrong, the result must be sent onto the network towards to the remote task sending the request. In figure 4.5.13.1, the behavior of the subroutine “Sending Result To Remote Task” is described.

![Figure 4.5.13.1. Behavior description of "Sending Result To Remote Task".](image)

1. Sending RWR-message onto network;
2. \( \text{RRTL(TN).Block\_State} = \text{rsRS}; \)
3. \( \text{RRTL(TN).TO} = 0; \)

For sending the result to a remote task, the remote-wait-result (RWR) is sent onto the network. In this message, the TN together with its \( \text{MuP} \) address, the SW, the WR\_SW-bit and eventually the result word. After that, the block's state is set to \( \text{rsRS} \) and the TO-bit is set to 0b for starting a new message timeout time.

### 4.5.14 Execute Mailbox

There are two operations for accessing to a mailbox, the read-operation and the write-operation. When reading from a mailbox and the requested mailbox is empty, the task sending the request is then blocked. Therefore, the task is inserted into the local resource waiting list. When the reading from a mailbox conditionally the task will not be blocked, but the SW is updated to indicate what went wrong and sent back to the task. After this, the task is made runnable. For a remote task, the RWR-message will be sent onto the network. When the request has been executed successfully and there are waiting tasks for writing, one waiting task can be wake up for writing.

It will also be the same when the request is a write-operation while the requested mailbox is full. However, when the write-operation has been executed successfully and there are waiting tasks for reading, one waiting task can be wake up for reading.

As main inputs the resource list RL for administrating the resources with the concerning resource data block RD and the request (RI-register) itself are needed. For a remote task, also the RRTL is needed when a result has to be sent back. In figure 4.5.14 on the next page the behavior of the block “Execute Mailbox” is shown.
Figure 4.5.14. Behavior description of the subroutine “Execute Mailbox”.

Firstly, the type of operation must be determined. In this case, it is a read or a write-operation. When reading from a mailbox and the requested mailbox is empty, the task sending the request will be blocked. Therefore, the subroutine “Block Task” is called to insert the task into the local resource waiting list. Its behavior description is described in subsection 4.5.14.1 on page 59. But, when reading from a mailbox conditionally and the requested mailbox is empty the SW is updated to indicate what went wrong, which after the task is made runnable by calling the subroutine “Restart Task”. In subsection 4.5.14.2 on page 60 its behavior is described in more details. If the requested mailbox is not empty, the subroutine “Execute Reading Task” is called to execute the request. In subsection 4.5.14.3 on page 61 its behavior is described in more details. When there are waiting tasks that want to write to a mailbox while the mailbox was full, the first waiting task can be wake up after the read-operation has been executed successfully. Therefore, the subroutine “Wake Up Writing Task” is called to handle its request. Its behavior is described in subsection 4.5.14.4 on page 62 in more details.

When writing to a mailbox and the requested mailbox is full, the task sending the request is then blocked. By calling the subroutine “Block Task” as described in subsection 4.5.14.1 on page 59 to insert into the local resource waiting list. For writing conditionally the subroutine “Restart Task” is called to make the task runnable, as described in subsection 4.5.14.2 on page 60. If the requested mailbox is not full, the subroutine “Execute Writing Task” is called to execute the request. Its behavior is described in subsection 4.5.14.5 on page 64 in more details. When a task wants to read from a mailbox and the requested mailbox was empty, it will then be wake up after the write-operation has been executed successfully. Therefore, the subroutine “Wake Up Reading Task” is called to handle the task. Its behavior is described in subsection 4.5.14.6 on page 64 in more details.
4.5.14.1 Block Task

When a task wants to read from an empty mailbox or wants to write to a full mailbox the task will then be blocked until the requested resource is available. Alternatively, until the request is cancelled by the task sending the request or by another task. Therefore, the task has to be inserted into the local resource waiting list. In contrast with inserting a task into a timeout waiting list, there are only situations in which a task can be inserted into a local resource waiting list. A task waiting for a local resource will always be inserted at the end of the waiting list. As shown in figure 4.5.8.1a on page 42 the new waiting task will be inserted into the waiting list according to the first situation if the list is empty or according to the third situation if the list is not empty. In figure 4.5.14.1 the behavior description for the subroutine "Block Task" is shown.

```plaintext
1. IF (TN >= 03FC0H) { // Local TN ∈ [03FC0H ... 03FFEh]
2.   TL(TN).State = tsWLR;
3.   RL(RN).Last_TN = RL(RN).Last_TN;
4.   TaskQ(TN).Runnable = 0;
6. }
7. ELSE {
8.   Send PWL-message onto network;
10.  RRTL(TN).TO = 0;
11. }
12. IF (RL(RN).First_TN == NIL) {
13.   RL(RN).First_TN = TN;
14.   RL(RN).Last_TN = TN;
15. }
16. ELSE {
17.   IF (RL(RN).Last_TN >= 03FC0H) { // non-empty list and last task is local task
18.     TL(RL(RN).Last_TN).Next_TN = TN;
19.   }
20. ELSE {
22.     RL(RN).Last_TN = TN;
23.   }
24. }
```

Figure 4.5.14.1. Behavior description for subroutine "Block Task".

The subroutine “Block Task” consists of two parts. From line 1 to line 9, the task administration will be updated while line 10 to line 24 the task sending the request is inserted into the local resource waiting list.

For updating the task administration the state of the concerning task on the TL is set to tsWLR if it is a local task. Further, the whole request has to be stored onto the field “Request”, on the TaskQ its field “Runnable” is set to 0 to block the task and for connecting the task with the last waiting task on the local resource waiting list, the pointer Prev_TN of blocking task is set to the pointer Last_TN of the attached mailbox. For a remote task the placed-on-waiting-list (PWL) message including the remote index number (RIN) is sent onto the network towards the remote task indicating its request has been placed on the waiting list. After that, its state on the RRTL is changed to rsOWLA. As for local task, the pointer Prev_TN is set to the pointer Last_TN. As last, the TO-bit on the RRTL is set to 0 for starting a new timeout for the sent PWL-message.

For inserting the task into the waiting list, there are two situations. When the waiting list of the requested resource is empty both the pointer First_TN and the pointer Last_TN are set to refer to TN indicating the beginning and the end of the list respectively. For a non-empty waiting list, the type of the last waiting task on the list must be determined whether it is a local task or a remote task. If the task number TN of the last waiting task ∈ [03FC0H and 03FFEh], then it is a local task. Therefore, the next pointer Next_TN of the last waiting task on the TL is set to refer to the inserting task. Else, it is a remote task and therefore the next pointer Next_TN of the last waiting task on the RRTL is set to refer to the inserting task. Finally, the last pointer Last_TN of the requested resource is set to refer to the inserting task.
4.5.14.2 Restart Task

When a task wants to access to a local resource conditionally and the requested resource is not available at that moment, the task sending the request will not be blocked. The status word SW is then updated to indicate what went wrong which after the task is made runnable if it is a local task and a result-sent (RS) message is sent onto the network if it is a remote task. In figure 4.5.14.2 the behavior of the subroutine "Restart Task" is described.

Firstly, the SW is updated to indicate what went wrong. In this case, the CF-bit is set to 1b indicating the condition has been failed. Next, if it is a local task, TN ∈ [03FC0H...03FFE0H], the subroutine "Sending Result To Local Task" is called to send the result back to the local task sending the request. Its behavior is already described in subsection 4.4.1 on page 32. After that, the subroutine "Wake Up" is called to wake up the local task. Its behavior is already described in subsection 4.4.2 on page 32. However, if a remote task had sent the request, the subroutine "Sending Result To Remote Task" is called to send the result back to remote task. Its behavior is already described in subsection 4.5.13.1 on page 57.
4.5.14.3 Execute Reading Task

When a local or a remote task wants to read from a mailbox and it is not empty, the subroutine “Execute Reading Task” is called to execute the request. After the request has been executed successfully, the task has to be made runnable if it is a local task. For a remote task, the RWR-message will be sent onto the network towards the remote task. In figure 4.5.14.3a, the behavior of the subroutine “Execute Reading Task” is described.

As known, a mailbox can have up to 16 slots each for one message. Because the mailbox uses the FIFO memory structure, the pointer Next_Message is needed to indicate the next to read message. In figure 4.5.14.3b, an example is given of a mailbox with four slots.
the number of present messages on the mailbox is decreased by one and the field “Last Operation” is set to ‘read’.

For sending back the result it must be determined, firstly whether it is a local request or a remote request. For a local request, the subroutine “Sending Result To Local Task” is called. Its behavior is already described in subsection 4.4.1 32. For waking up the local task sending the request, the subroutine “Wake Up” is called. Its behavior is already described in subsection 4.4.2 on page 32. For a remote request, the subroutine “Sending Result To Remote Task” is called. Its behavior is already described in subsection 4.5.13.1 on page 57.

4.5.14.4 Wake Up Writing task

When a task wants to write a message to a mailbox while it is full the task is then blocked. Because another task has just read a message from the same mailbox only one waiting writing task can be wake up. That is the first waiting task that is referred by the pointer First_TN of the requested resource. For each mailbox only writing or only reading tasks can wait.

The intention is placing the request of the first waiting task onto the R1-register and updating the local resource waiting list from which the task is removed. For execution the subroutine “Execute Writing Task” is called that is described in subsection 4.5.14.5. The situation in which the first waiting task will be removed is shown in figure 4.5.14.4a.

![Diagram](image)

*Figure 4.5.14.4a. Removing the first waiting task from the local resource waiting list.*

The behavior description for the subroutine “Wake Up Writing Task” is shown in figure 4.5.14.4b on the next page.
The Multi-Micro Processor

The Task Control Unit: Specification and Design

1. Temp_TN = RL(RN).First_TN;
2. IF (Temp_TN is local task) {
   3. RI-register.Request = TL(Temp_TN).Request;
   4. IF (TL(RL(RN).First_TN).Next_TN == local task) { // Next task is a local task
      6. RL(RN).First_TN = TL(RL(RN).First_TN).Next_TN; // Line nr. 2
   7. }
   8. ELSE {
      10. RL(RN).First_TN = RRTL((RL(RN).First_TN).Next_TN); // Line nr. 3
      11. }
      12. TL(Temp_TN).Next_TN = NIL; // Line nr. 2
   13. }
   14. ELSE {
      15. RI-register.Request = RRTL(Temp_TN).Request;
      16. IF (TL(RL(RN).First_TN).Next_TN == local task) { // Next task is a local task
         18. RL(RN).First_TN = TL(RL(RN).First_TN).Next_TN; // Line nr. 3
      19. }
      20. ELSE {
         22. RL(RN).First_TN = RRTL((RL(RN).First_TN).Next_TN); // Line nr. 3
      23. }
      24. RRTL(Temp_TN).Next_TN = NIL; // Line nr. 2
  25. }
   26. IF (RL(RN).First_TN == NIL)
      27. RL(RN).Last_TN = NIL;

Figure 4.5.14.4b. Behavior description for the subroutine “Wake Up Writing Task”.

As shown in figure 4.5.14.4b, the behavior of the subroutine “Wake Up Writing Task” consists of two IF-loops. The first loop consists of line 2 up to line 25 and the second loop consists of line 26 and line 27. In the first loop, the request of the first waiting writing task will be placed onto the RI-register and updating the local resource waiting list. In the second loop the pointer Last_TN of the concerning resource will be updated. It will be set to NIL if the list is empty.

In the first IF-loop, if the removing task is a local task, its TN $\in [03FC0H ... 03FFE0H]$, the code of line 3 up to line 13 is executed. If it is a remote task, its TN $\in [0000H ... 03FC0H]$, the code of line 15 up to line 25 is executed. For a local task, the request is fetched from the TL and placed onto the RI-register. If the next waiting task is a local task its previous pointer Prev_TN is set to NIL and the pointer First_TN of the requested resource is set to refer to the next waiting local task. This is done in line 5 and in line 6. Else, the previous pointer Prev_TN of the remote task is set to NIL and the pointer First_TN of the requested resource is set to refer to the next waiting remote task. This is done in line 17 and in line 18. As last, the next pointer Next_TN of the removing local task is set to NIL. This is done in line 12.

If the removing task is a remote task, the request is fetched from the RRTL and placed onto the RI-register. If the next waiting task is a local task its previous pointer Prev_TN is set to NIL and the pointer First_TN of the requested resource is set to refer to the next waiting local task. This is done in line 19 and in line 20. Else, the previous pointer Prev_TN of the remote task is set to NIL and the pointer First_TN of the requested resource is set to refer to the next waiting remote task. This is done in line 22 and in line 23. As last, the next pointer Next_TN of the removing local task is set to NIL. This is done in line 24.
4.5.14.5 Execute Writing Task

When a task wants to write a message to a mailbox and it is empty or not full, the subroutine “Execute Writing Task” is called to execute the request. After the request has been executed successfully, the task has to be wake up. In figure 4.5.14.5 its behavior is described.

1. Next_Free_Slot = RL(RN).Nr_Used + RL(RN).Next_Message;
2. IF (Next_Free_Slot > RL(RN).Nr_Slots) = RJ-register.Parameter2;
3. RD(Next_Free_Slot - RL(RN).Nr_Slots - 1) = RJ-register.Parameter2;
4. ELSE
5. RD(Next_Free_Slot) = ‘write’;
6. RL(RN).Nr_Used++; // TN ∈ [03FC0H ... 03FFE0H]
7. RL(RN).Last_Operation
8. IF (local task) {
9. Sending_Result_To_Local_Task();
10. Wake_Up();
11. }
12. ELSE
13. Sending_Result_To_Remote_task();

Figure 4.5.14.5. Behavior description for the subroutine “Execute Writing Task”.

As shown in figure 4.5.14.3b on page 61 an example is given of a mailbox with four slots, Nr_Slot = 3 (10b). Before writing the data to the mailbox, the next free slot must be calculated. In the example there are two messages present on the mailbox at slot number 1 and number 2. Therefore, the pointer Nr_Used is equal to 2. The pointer Next_Message refers to the next to read message and that is the message at slot number 1. The next free slot Next_Free_Slot is (Nr_Used + Next_Message = 2 + 1) 3. When slot number 2 and slot number 3 are in use instead of slot number 1 and slot number 2, the pointer Next_Free_Slot will be equal to (Nr_Used + Next_Message = 2 + 2) 4. That is just greater than Nr_Slots, which means that the new message has to be placed onto the slot (Next_Free_Slot - Nr_Slots - 1 = 4 - 3 - 1 =) 0.

After the message has been written, a result can be sent back to the task and the task sending the request will be wake up. This is done from line 8 up to line 13. If a local task had sent the request, the subroutine “Sending Result To Local Task” is called. Its behavior is already described in subsection 4.4.1 on page 32. In addition, for waking up the local task, the subroutine “Wake Up” is called. Its behavior is already described in subsection 4.4.2 on page 32. For a remote task, the subroutine “Sending Result To Remote Task” is called. Its behavior is already described in subsection 4.5.13.1 on page 57.

4.5.14.6 Wake Up Reading Task

The subroutine “Wake Up Reading Task” behaves the same as the subroutine “Wake Up Writing Task”, which is already described in subsection 4.5.14.4 on page 62. If there are waiting tasks, then all of them are reading tasks or writing tasks. In the case of reading tasks, the first waiting task will be wake up and executed.
4.5.15 Execute Pipe

Normally, each pipe has a buffer with a certain length. Both the local tasks and the remote tasks can use the pipe for transferring of block data. For writing to a pipe, the task can fill the buffer completely or partially. For writing, the task will send the pointer (start address) of the buffer to the TCU on which the block data is kept for transferring. The first word of this buffer indicates the length of the block data. For writing the data to the buffer of the pipe, the TCU will read the data from the buffer the sending task will transfer. If the buffer of the pipe is big enough, the complete block data will be placed onto the buffer of the pipe. Else, a block error will be reported.

For reading, the task will send the pointer (start address) of its empty buffer to the TCU on which the block data can be stored. The first word of this buffer indicates the length of the buffer. In turn, the TCU will fill the task's buffer until it is full. If the reading block data is smaller than the buffer of the reading task, the task will then be blocked.

Because of the complexity and the flexibility of the pipe, a complete protocol for the pipe must be written in the future.

4.5.16 Execute Semaphore

The subroutine “Execute Semaphore” behaves almost the same as the subroutine “Execute Mailbox”. The difference is that one or more waiting writing tasks can be wake up after a read-operation has been executed and one or more waiting reading tasks can be wake up after a write-operation has been executed. It is depended on the requested number of units or the available number of units. Therefore, as for the mailbox the subroutine “Execute Reading Task” and the subroutine “Execute Writing Task” are needed to change. Their behaviors are described in figure 4.5.16a and in figure 4.5.16b on the next page respectively.

For executing the reading task, the subroutine “Execute Reading Task” will execute the request. After that, the result will be sent back to the task sending the request and waking up the task. Next, it has to check whether one or more waiting tasks can be wake up by executing the tasks' requests. In figure 4.5.16, the behavior of the subroutine “Execute Semaphore” is described globally.

![Figure 4.5.16. Global behavior description for the subroutine “Execute Reading Task”.

When a new request is available, it must be checked whether the attached semaphore is available. If it is a read-operation and there are reading tasks present on the local resource waiting list, it means the attached semaphore is not available. Therefore, the task sending the request is placed onto the resource waiting list. If the attached semaphore is available, the request will be executed. After execution, it must be checked whether there are
waiting reading tasks. If so, then the requested number of units of the next first waiting task must be checked. If it is available, the request of the waiting task is fetched. If it is a local task, then the request is fetched from the TL. Else, from the FIFO-buffer on the Network Manager. All requests receiving from remote tasks are kept in the FIFO-buffer in the Network Manager. For execution, the request is placed on the RI-register. Finally, the subroutine “Execute Semaphore Request” is called for execution.

In subsection 4.5.16.1 up to subsection 4.5.16.3, the behavior of the subroutine “Execute Reading Task” is described.

4.5.16.1 Behavior description of “Execute Reading Task”

In figure 4.5.16.1a, the behavior of the subroutine “Execute Reading Task” is described.

1. WHILE (RI-register.Avail == 1) {
2.     Execute_Semaphore_Request(RI-register);
3.     IF(RL(RN).First_TN == DEST_TN) {
4.         Removing_Task_From_Resource_Waiting_List(DEST_TN);
5.     } IF (RI-register.Avail == 1) {
6.         IF(RL(RN).First_TN != NIL)
7.             Get_Next_Request(RN);
8.     }
9. }

Figure 4.5.16.1a. Behavior description for the subroutine “Execute Reading Task”.

As long as the AVAIL-bit on the RI-register is equal to 1b, the while-loop will be repeated. Firstly, the subroutine “Execute Semaphore Request” is called for executing the request present on the RI-register. Its behavior is described in figure 4.5.16.1b on the next page. After execution, if the TN of the target task is equal to the pointer First_TN of the attached semaphore, it means a waiting task has been executed. Therefore, the subroutine “Removing Task From Resource Waiting” is called to remove the executed task from the local resource waiting list. Else, it means a new request has been executed. If the AVAIL-bit is equal to 1b after the execution, it means the requested number of units of the executing task was not available. Else, if the pointer First_TN is equal to NIL, it means the end of the waiting list. Else, the subroutine “Get Next Request” is called to get the request of the first waiting reading task. Its behavior is described in figure 4.5.16.1d on page 68.
In figure 4.5.16.1b, the behavior of the subroutine “Execute Semaphore Request” is described.

If the attached semaphore is a short semaphore, then the code in line 2 up to line 10 is executed, else the code of line 14 up to line 22 is executed. The behavior of these two sections is the same. The difference is that the field “Nr.Units” on the RL is needed for a short semaphore while the resource data block on the RD is needed for a long semaphore.

If the requested number of units is available, the field “Nr.Units” is then decreased by the requested number of units that is held by the second parameter on the RI-register. Further, if the remaining number of units of the attached semaphore is equal to zero, the AVAIL-bit on the RI-register is set to 0b so that the waiting tasks will not be checked for execution. For sending the result back, the subroutine “Sending Result” is called. Its behavior is described in figure 4.5.16.1c.

However, if the requested number of units of the attached semaphore is not available, it means that a new request has been executed. Therefore, the subroutine “Insert task into resource waiting list” is called to insert the task into the local resource waiting list. Also the AVAIL-bit on the RI-register is set to 0b so that the waiting tasks will not be checked for execution.

In figure 4.5.16.1c, the behavior of the subroutine “Sending Result” is described.

If a local task had sent the request, the subroutine “Sending Result To Local Task” is called for sending the result back. In addition, the subroutine “Wake Up” is called to wake up the task. Their behaviors are already
described in subsection 4.4.1 and in subsection 4.4.2 on page 32 respectively. Else, it means a remote task had sent the request. Therefore, the subroutine “Sending Result To Remote Task” is called for sending the result back to the remote task. Its behavior is already described in subsection 4.5.13.1 on page 57.

As last, in figure 4.5.16.1d, the behavior of the subroutine “Get Next Request” is described.

```
1. Temp_TN = RL(RN).First_TN;
2. IF (Temp_TN == local task) {
3.     IF (RL(RN).Last_Operation == 0)
4.         IF (RL(RN).Units ≥ TL(Temp_TN).Request.Parameter2)
5.             RI-register.Request = TL(Temp_TN).Request;
6.         ELSE
7.             RI-register.Avail = 0;
8.     ELSE
9.         IF (RD(RN) ≥ TL(Temp_TN).Request.Parameter2)
10.        RI-register.Request = TL(Temp_TN).Request;
11.        ELSE
12.       RI-register.Avail = 0;
13.   }
14. ELSE {
15.     IF (RL(RN).Last_Operation == 0)
16.       IF (RL(RN).Units ≥ RRTL(Temp_TN).Parameter)
17.         RI-register.Request = FIFO.Request;
18.     ELSE
19.       RI-register.Avail = 0;
20.   ELSE
21.     IF (RD(RN) ≥ RRTL(Temp_TN).Parameter)
22.       RI-register.Request = FIFO.Request;
23.     ELSE
24.       RI-register.Avail = 0;
25.   }
26. }
```

Figure 4.5.16.1d. Behavior description of “Get Next Request()”.

Firstly, the temporal register Temp_TN is used for making a copy of the pointer First_TN of the attached semaphore. The intention is to get short notation. Further, the code in line 3 up to line 12 will be executed, if the first waiting task is a local task. Else, it is a remote task. So, the code in line 15 up to line 24 is executed. However, they both behave the same. The difference is that the request will be fetched from the TL if the request is sent by a local task. Else, the request is fetched from the FIFO-buffer on the Network Manager, because the request is sent by a remote task.

If the attached semaphore is a short semaphore, the comparison is made from the field “Nr_Units” on the RL. Else, from the corresponding resource data block on the RD. Next, if the requested number of units is available, the request is fetched from the TL or from the FIFO-buffer and placed onto the RI-register. Because the AVAIL-bit is already high, it is not need to set anymore. However, if the requested number of units is not available, the AVAIL-bit is set to 0b so that the execution can be ended.

4.5.16.2 Behavior description of “Execute Writing Task”.

The subroutine “Execute Writing Task” behaves the same as the subroutine “Execute Reading Task”. There are two differences. Firstly, the sign ‘-’ in line 3 and in line 15 of figure 4.5.16.1b on page 67 must be replaced by the ‘+’. Secondly, the subroutine “Sending Result” of figure 4.5.16.1c on the previous page must replace by figure 4.5.16.2 as shown on the next page.
The Multi-Micro Processor

The Task Control Unit: Specification and Design

1. IF (local task) {  // TN ∈ [03FC0H ... 03FFEh]
   2. Sending_Result_To_Local_Task(DEST_TN);
   3. Wake_Up(DEST_TN);
   4. }
5. ELSIF (remote task)  // Remote TN ∈ [00000H ... 03FBFH]
6. Sending_Result_To_Remote_Task(DEST_TN);

Figure 4.5.16.2. Behavior description of “Sending Result()” for writing tasks.

Only the ELSE in line 5 of figure 4.5.16.1c must be replaced by ELSIF (remote task) as shown in figure 4.5.16.2. The reason of the change is that a write-operation from one of the four activity lines can be handled. Because the TN of this request is equal to 03FFFH, which is the NIL-task, no result is needed to send back. For further information, see chapter 6 on page 78.

4.6 Local Resource Request Handling

As described in section 4.4 on page 31, the block “Local Request Handling” that decodes the local request handles three types of local requests. The first type is executing of some special instructions, the second type is executing the requests for accessing to local resources and the third type is executing of requests for accessing to the remote resources. The first type is already described in section 4.5 on page 34 up to subsection 4.5.8 on page 41.

Before describing the second type, the principle of writing to a resource and reading from a resource are already described in subsection 4.5.11 on page 54 and in subsection 4.5.12 on page 54 respectively. Besides, the block “Execute Resource Request” is already described in subsection 4.5.13 on page 56. The intention of this block is executing the requests for accessing to local resources. Therefore, the requests must be stored on the RI-register. For placing the requests on the RI-register, it will be done by the block “Local Resource Request Handling” that is described in this section or by the block “Remote Request Handling” that is described in chapter 5 on page 76.

For the third type, the block “Remote Resource Request Handling” will handle all requests for accessing to remote resources. It will be described in section 4.7 on page 71.

4.6.1 The behavior description

The intention of the block “Local Resource Request Handling” is checking whether the requested resource is available. When not available, the task sending the request is placed onto the local resource waiting list. Else, it means the requested resource is available. Therefore, the request must be placed onto the RI-register for execution.

As inputs, the task number TN, the status word SW to be able to indicate what went wrong, the instruction code IC, the result return address RRA, Parameter 1 for holding the RN and Parameter 2 for holding DATA are needed. For a mailbox, DATA is equal to the message if it is a write-operation. For a pipe, DATA is equal to the pointer or the start address that refers to the main memory location where the block data is stored if it is a write-operation. In addition, for a semaphore, DATA is equal to the number of units. The behavior description for this subroutine is shown in figure 4.6.1 on the next page.
If the resource is not available yet, the subroutine “Insert task into resource waiting list” is called to insert the task into the local resource waiting list. Its behavior is described in subsection 4.6.2. When the requested resource is available, the subroutine “Placing local request onto RI-register” is called to execute the request. Its behavior is described in subsection 4.6.3.

4.6.2 Inserting task into resource waiting list

For the local resource waiting list, the new waiting task is always inserted at the end of the list according to the FIFO-structure. Its behavior description is shown in figure 4.6.2a.

Firstly, the whole request is stored on the TL. Next, the task’s state is changed to waiting-for-local-resource (tsWLR) and on the TaskQ its field “Runnable” is changed to 0b to block the task. If it is the first waiting task the pointer First_TN and the pointer Last_TN of the requested resource are set to refer TN. Else, if the last task on the waiting list is a remote task, RL(RN).Last_TN < 03FC0H, the next pointer Next_TN of this task on the RRTL is set to refer to the inserting task. For a local task, its next pointer on the TL is set to refer to the inserting task. The principle is shown in figure 4.6.2b on the next page.
4.6.3 Placing local request onto RI-register

When the requested resource is available, the subroutine "Placing local request onto RI-register" is called to place the request onto the RI-register of the block “Execute Resource Request”. The behavior of the block is already described in subsection 4.5.13 on page 56. The behavior description of the subroutine “Placing request onto RI-register” is shown in figure 4.6.3.

For execution, only the request is needed to place onto the RI-register. After the request has been placed on the RI-register, the AVAIL-bit has to set to 1b so that the request can be executed by the block “Execute Resource Request”.

4.7 Remote Resource Request Handling

The block “Remote Resource Request Handling” handles all requests from local tasks that want to access to remote resources. When the request has been executed completely the concerning MpP will send the result back. In addition, when a local task wants to exchange its data with a remote task, the local task can send a message including the data via the network to a remote mailbox. After that, the concerning remote task can read the message. Therefore, the intention of this block is sending the request onto the network and updating the task list TL.

Because it is not sure that in each cycle of the block “Request Handling”, also the controller of the TCU, the block “Remote Resource Request Handling” will be called, handling of all incoming messages will be handled by the block “Message Handling”. Its behavior is described in chapter 7 on page 79. In figure 4.7a on the next page, the communication between a local task’s request and the requested remote resource is shown.
When there is a local request present intended for accessing to the remote resource, the block “Remote Resource Request Handling” will send this request onto the Network Manager. In turn, the Network Manager will send the request to the MuP handling the resource after the request is encoded to a message. For further information, see chapter 10 “The Network Manager” on page 91. When the target MuP has received the message, it decodes the message to the request and sends this to the block “Message Handling” that will be described in chapter 7 on page 79. After the block “Message Handling” has received the request it will check the existence of the requested resource after it has stored the request onto the RRTL and updating the RL eventually. If the requested resource does not exist, the SW will be updated and sent back to the local task sending the request. When the local Network Manager has received the result, it will set the bit message-received (MR-bit) on the E-register indicating a new message is available. If the requested resource does exist and is available, the request will be executed. Therefore, the block “Remote Request Handling” will place the request onto the RI-register. After the request has been executed completely by the block “Execute Resource Request”, it will send the result back to the local task by sending the RWR-message onto the network, the Network Manager.

In figure 4.7b, the behavior of the block “Remote Resource Request Handling” is shown.

1. Send RWC-message onto network;
2. TL(TN).State = tsWCA;
3. TaskQ(TN).Runnable = 0;
4. TL(TN).TO = 0

Figure 4.7b. Behavior description of “Remote Resource Request Handling”.

For each new incoming request, the remote-wait-command (RWC) is sent onto the network towards the target resource, containing the whole request. After that, the task’s state on the TL is changed to wait-for-command-acknowledgement (tsWCA) while on the TaskQ its field “Runnable” is changed to 0b to block the task. As last, the TO-bit is set to 0b to start a new timeout for the sent RWC-message. The principle of the TO-bit is described in subsection 4.7.1.
4.7.1 Principle of message timeout

When a message has been sent onto the network towards the remote MiμP, the TO-bit can be used for keeping the message timeout time. Its principle is shown in figure 4.7.1.

![Figure 4.7.1. Principle of keeping the message timeout time.](image)

On the TCU, a message timeout timer runs continually. Suppose the length of the timeout time is set to $\frac{1}{2}$ second. Each time that the timer has been elapsed for $\frac{1}{2}$ second the timeout bit TO is set to 1b. For further information, see chapter 8 on page 87. When checking for the timeout occurrence periodically this bit can be checked. If it is equal to 1b, the timeout had been occurred and 0b when it had not been occurred.

When TO = 0b and the timeout time has occurred TO is set to 1b, which means that the timeout has been occurred for the first time. However, if TO = 1b it means the timeout time has occurred for the second time. Therefore, it can be concluded that the communication is lost. In this case, the communication link can be broken or the MiμP handling the task is down.

### 4.8 Delaying Task Handling

The intention of the block "Delaying Task Handling" is waking up one or more waiting local tasks that had requested for a certain timeout time. All of these waiting tasks are kept in the timeout waiting list. When inserting a task onto the timeout waiting list, its TN will be inserted into the list using its finish-time. So, a task with a greater finish-time will be inserted behind the other task with a smaller finish-time.

With the instruction DELAY(TN, DELAY_TIME), a local task can put itself or another task asleep for a certain timeout time. After the requested timeout time has been occurred, the task will be wake up. For administrating the tasks’ timeout time, the field “Finish-time” on the TL is used for keeping the finishing (awaking) time. For realizing the timeout waiting list, a double linked list is used with the pointer Prev_TO_TN that refers to the previous waiting task and the pointer Next_TO_TN that refers to the next waiting task. A double linked list is needed, because a waiting task can be removed from the list at any moment. Besides, the register First_TO_TN is needed to refer to the first waiting task.

For execution, it is assumed that the Timers Generator will set the TO-bit on the E-register when the timeout of the first blocked, which is referred by the pointer First_TO_TN, task is occurred.

The behavior of the subroutine “Timeout Task Handling” is described in figure 4.8 on the next page.
If the TO-bit on the E-register is equal 0, it means the timeout time of the first blocked task, which is referred by the pointer First_TO_TN, has not been occurred. Therefore, the block "Timeout Task Handling" is no need to handle any task. Else, the current time TO-time is fetched from the timeout timer. Next, a copy of the finish time of the first waiting task that is referred by the pointer First_TO_TN is made. It is just needed to get a short notation. If the TO-time is greater than or equal to the Finish-time, one or more waiting tasks can be wake up.

To wake up the current task that is referred by the register First_TO_TN, its state on the TL is set to tsRTR and on the TaskQ its field "Runnable" is set to 1b to it runnable. Next, the timeout waiting list must be updated. If the register First_TO_TN is equal to the register Last_TO_TN, it means there is only one waiting task present on the list. Therefore, the register First_TO_TN and the register Last_TO_TN are set to NIL indicating the beginning and the end of the list. Else, it means, there are waiting tasks. Therefore, the register First_TO_TN is set to refer to the next waiting task that is referred by the next pointer Next_TO_TN of the current task. As last, the previous pointer Prev_TO_TN of the next waiting task is set to NIL indicating the beginning of the list.

To wake up the next waiting task, the subroutine "Timeout Task Handling" is recalled.

### 4.9 Time slicing

Standard, the TCU has to administrate of 63 local tasks. For the moment, there are four local processing units (LPU's) available for executing four tasks simultaneously. On the TaskQ, all of these tasks are sorted automatically each time a waiting task is set running or when the priorities of the running and or of the runnable tasks are changed. The tasks are sorted according to their priorities from 0 (highest) up to 15 (lowest). When all LPU's are in use and there are runnable tasks that have the same priority as the running task(s), two or more tasks have to share the same LPU. Therefore, a mechanism called time-slicing is needed to arrange these tasks. This mechanism is also known as the Round-Robin Scheduling, which is often used in the standard operating system.
The TS-mechanism is already implemented on the TaskQ. Because there are four LPU's used, four 6-bits registers is used indicating which tasks are executing by holding the tasks' TN. For executing the TS-mechanism, only a copy of these registers has to be made. When using these TN's, the priorities of these tasks can be rewritten. After that, the TaskQ will sort the tasks automatically. As result, tasks with equal priority will be time-sliced.

Suppose, these registers are called LPU-register0 up to LPU-register3. Then the behavior of the block Time_Slicing can describe as shown in figure 4.9.

```plaintext
1. FOR (i = 0; i < 4; i++) {
2. Temp_TN = LPU-register(i);
3. Temp_PRIO = TaskQ(Temp_TN).PRIO;
4. TaskQ(Temp_TN).PRIO = Temp_PRIO;
5. }
```

Figure 4.9. Behavior description of the block "Time-slicing".

For handling the four running tasks, the for-loop of the programming language C is used. Firstly, a copy of TN of the running task is made and stored on the temporal register Temp_TN. Next, a copy of the priority of this task is made and stored on the temporal register Temp_PRIO. As last, the running task’s priority is rewritten by assigning the content of the register Temp_PRIO to the running task.
5. REMOTE REQUEST HANDLING

When a remote task wants access to a local resource, it will send a request to the local TCU via the Network Manager. The communication is shown in figure 5a.

![Figure 5a. The remote-task-to-local-resource communication.](image)

When a remote task wants access to a local resource, the block "Remote Resource Request Handling" of the remote MμP handling the remote task will send its request onto the Network Manager. In turn, the Network Manager will send its request to the local MμP handling the resource after the request is encoded to a packet message. For further information, see chapter II "The Network Manager" on page 93. When the target (local) MμP has received the packet, it decodes the packet to the message and sends this to the block "Message Handling" that will be described in chapter 7 on page 79. After the block "Message Handling" has received and decoded the message and it is an RWC-message, it will store the request partly onto the RRTL and set the RR-bit on the E-register if it was reset while the whole request is still held on the FIFO-buffer of the Network Manager.

When the controller, the block "Request Execution", of the TCU checks the RR-bit and it is high, the controller will trigger the block "Remote Request Handling". In turn, this block will check whether the requested local resource is available. If the resource is available, it fetches the resource from the FIFO-buffer and places the request onto the RI-register for execution. However, if the requested resource is not available, the task sending the request is placed onto the local resource waiting list.

In figure 5b, the behavior of the block "Remote Request Handling" is described.

![Figure 5b. Behavior description of "Remote Request Handling".](image)

The behavior for the block "Remote Request Handling" behaves the same as the block "Local Resource Request Handling" in figure 4.6.1 on page 70. The only difference is that the request must be fetched from the FIFO-buffer on the Network Manager. Therefore, the subroutine "Placing local request onto RI-register" of figure 4.6.1 is changed and is shown in figure 5c on the next page.
For execution, the first request from the FIFO-buffer on the Network Manager is fetched and placed onto the RI-register. After the AVAIL-bit on the RI-register is set to 1b, the request will be executed.
6. EVENTS HANDLING

Standard, there are four activity lines built-in on the MμP. When an external peripheral activates one or more of these lines, the TCU will trigger the attached semaphore(s).

When one or more of the activity lines is detected, the block “Activity Detection” will set the event bit E on the E-register and lets the “Execution Core” know which line(s) is/are activated by setting the concerning bit(s) on the activity register (A-register), which is 4 bits in width.

For execution, only the semaphore’s request is needed to place onto the RI-register of the block “Execute Resource Request”. Its behavior is already described in subsection 4.5.13 on page 56. Because a result can be sent to the task sending the request, the TN on the RI-register must be set to 03FFFH (NIL-task) so that result will be sent back neither waking up the task. The reason is that the activity lines are triggered by peripherals and not sent by a local or a remote task. Because the semaphore’s requests are initialized, it is assumed that setting the TN to 03FFFH is already done. As last, when there are tasks waiting for the attached semaphore, one or more waiting reading tasks can be wake up. In figure 6, the behavior of the block “Event Handling” is described.

```c
1. FOR(i = 0; i < 4; i++) {
2.   IF (A-register(i) == 1) {
3.     RI-register.Request = Semaphore-request(i);
4.     RI-register.Avail = 1;
5.     WHILE(RI-register.Avail);
6.     A-register(i) = 0;
7.   }
8. }
```

**Figure 6. Behavior description of the block “Event Handling”**.

For checking all bits on the A-register, the for-loop of the programming language C is used. If a bit on the A-register is high, the corresponding predefined semaphore’s request is placed onto the RI-register. The data structure of this request is the same as for the request sending by a writing task. The difference is that the TN of the semaphore’s request is equal to 03FFFH. In addition, the number of units is predefined. For initialization, this number can be set to a certain value. Further, the AVAIL-bit on the RI-register is set to 1b so that the request can be executed. After the request has been executed, the block “Execute Resource Request” will determine whether one or more waiting reading tasks can be wake up. As long as the AVAIL-bit on the RI-register is high, the next event will not be executed. For resetting the current bit on the A-register, it is set to 0b.
7. MESSAGE HANDLING

On the TCU the block “Message Handling” handles all the incoming messages received via the Network Manager. As described in section 4.7 “Remote Resource Request Handling” on page 71 and in chapter 5 “Remote Request Handling” different remote messages can be awaited. When a local task wants to access to a remote resource the block “Remote Resource Request Handling” will send an RWC-message including its request onto the network towards the target resource, after which the result is awaited. Especially waiting for the RWR-message in which the result word can be included. For the block “Remote Request Handling” the RWR-message can be sent onto the network after the request has been executed. In this case, the RA-message is awaited.

When a local task wants to access a remote resource its request will then be sent onto the network towards the target resource after which the result is expected. In the meantime, different things can occur, like time out of the sent message. Therefore, different messages can be received from the target MUP handling the resource.

When a remote task wants to access to a local resource, different messages can be received as shown in figure 7a.

When a local task’s request had been sent onto the network for accessing a remote resource, different messages can be received as shown in figure 7b.

In section 7.1 up to section 7.4, these messages are described comprehensively.

In section 7.5 up to section 7.7, these messages are described comprehensively.

In figure 7c on the next page, the behavior description of “Message Handling” is shown.

Suppose that only one message will be received for each cycle of the block “Request Handling”, which is the controller of the TCU. Therefore, the Network Manager will set the MR-bit on the E-register indicating a remote message is received. As shown in figure 7c the behavior of the block “Message Handling” consists of a case-
statement for selecting one of the possible messages. After the type of message is decoded the concerning subroutine is called for execution. In the next subsections, these subroutines will be described in more details.

### 7.1 Process RWC-message

When a remote task wants to access to a local resource it will send the remote-wait-command (RWC) message onto the network towards the local resource. In the message, the whole request is included. The intention is that when receiving this message, the block "Message Handling" will store the request onto the RRTL immediately and set the RR-bit on the E-register if it was reset. In figure 7.1 the behavior description of the subroutine "Process RWC-message" is shown.

```plaintext
1. IF (Free_Block == NIL) {
2.   SW.ERR = 1;
3.   SW.OOM = 1;
4.   Sending_Result_To_Remote_Task();
5. }
6. ELSE {
7.   RIN = Free_Block;
8.   Free_Block = RRTL(RIN).Next_TN;
9.   RRTL(RIN).Next_TN = NIL;
10.  RRTL(RIN).Prev_TN = NIL;
11.  RRTL(RIN).Block_State = rsOWLA;
12.  RRTL(RIN).Remote_TN = RWC.TN;
13.  RRTL(RIN).Remote_TN = RWC.Parameter1;
14.  RRTL(RIN).WR_SW = RWC.WR_SW;
15.  RRTL(RIN).IC = RWC.IC;
16.  RRTL(RIN).Parameter = RWC.Parameter2;
17.  IF (RWC.RD_SW == 1) RRTL(RIN).SW = RWC.SW;
18.  ELSE RRTL(RIN).SW = 00h;
19.  IF (E-register.RR == 0) E-register.RR = 1;
20. }
```

Figure 7.1. Behavior description for "Process RWC-message".

If the pointer Free_Block is equal to NIL, it means that all blocks on the RRTL are occupied. Therefore, the SW is updated indicating what went wrong before sending the result back to the remote task sending the request. In this case, the ERR-bit is set to 1b indicating an error has been occurred and the OOM-bit is set to 1b indicating out of memory. After that, the subroutine "Sending Result To Remote Task" is called for sending the result back. Its behavior is already described in subsection 4.5.13.1 on page 57.

If the pointer Free_Block is not equal to NIL, part of the new request can be stored onto the RRTL by making a copy of the request that is held on the FIFO-buffer. For storing the request, the block referred by the pointer Free_Block is used. Therefore, the RIN is loaded with the content of the pointer Free_Block. Free_Block is updated with the content of the pointer “Next_TN” that refers to the next free block on the RRTL. After that, the pointer “Next_TN” and the pointer “Prev_TN” are set to NIL indicating the beginning and or the end of the list. For occupying the block, the block’s state is set to rsOWLA indicating on-waiting-list-alive.

Further, the code in line 12 up to line 16 the needed information is stored by making a copy from the RWC-message. If the RD_SW-bit on the message is equal to 1b, it means the given SW is available. Therefore, a copy of the given SW is made. Else, the field “SW” on the RRTL is set 00h indicating a dummy SW. As last, if the RR-bit on the E-register is equal to 0b, it is set to 1b to indicating a new remote request is available.
7.2 Process CIA-message

When receiving the check-if-alive (CIA) message, also the remote index number (RIN) is included, it means a remote task sending the request wants to check the liveness of the requested local resource.

There are three things, which can happen. The RIN included in the CIA-message does not match with its block address on the RRTL, the requested local resource does not exist anymore or the requested resource is still alive. In the first situation, the RWR-message has to be sent back after the SW has been updated indicating incorrect TN. In the second situation, the RWR-message is sent back after the SW has been updated indicating the resource does not exist anymore. In addition, if the requested resource is still alive, the RWR-message is sent back after the SW has been updated indicating the resource is alive. In figure 7.2 the behavior description of subroutine “Process CIA-message” is shown.

If the RIN corresponding to the remote task sending the request does not match with the block address on the RRTL, the status word SW is updated to indicate what went wrong. In this case, the ITN-bit is set indicating the incorrect TN. Else, if the requested local resource is still exist, the RWR-message is sent onto the network after the SW has been updated indicating the resource is alive by setting the RIA-bit. If the requested resource had been removed, the RWR-message is sent onto the network after the SW has been updated indicating the resource does not exist by setting the RNE-bit. Because the EWA-message is awaited before clean up the concerning task’s block, the timeout bit TO on the RRTL is set to 0b to start the message timeout for the sent message. In subsection 4.7.1 on page 73, the principle is already described. Because the EWA-message instead of waiting for the RA-message must be awaited before cleaning up the task’s block, the block’s state is set to rsWEWA and not to rsRS as usually for sending an RWR-message.

7.3 Process CW-message

The remote task sending the request for accessing a local resource or another remote task can send a cancel-wait (CW) message to the local MµP handling the resource to cancel its request. When a CW-message is received, it means that a remote task wants to cancel its request that is intended for accessing a local resource. So, the task block for the concerning task on the RRTL has to be cleaned up after the end-wait-acknowledgement (EWA) message including the RIN is sent back to confirm the request. Because the remote task has requested for accessing a local resource and is therefore kept on the local resource waiting list, it has to be removed from the list. The behavior description is shown in figure 7.3 on the next page.
If the RIN corresponding to the remote task sending the request does not match with the block address on the RRTL, the status word SW is updated to indicate what went wrong. In this case, the ITN-bit is set indicating the incorrect TN. After that, the RWR-message including the remote index number is sent onto the network towards the remote task. Else, the EWA-message including the remote index number is sent onto the network towards the remote task to confirm the request. For chaining the free blocks, the field “Next_TN” is set to refer to the block that is referred by the pointer Free_Block. Next, the task block for the concerning task is set to rsTBF to clean up the block. As last, the subroutine “Removing Task From Resource Waiting List” is called to remove the task from the local resource waiting list. Its behavior is already described in subsection 4.5.3.1 on page 37.

### 7.4 Process RA-messages

A result-acknowledgement (RA) message including the RIN can only be received when an RWR-message was sent. It means that the request of a remote task had been executed successfully after which the RWR-message had been sent back. Therefore, the request of the remote task on the RRTL can be removed by setting its block’s state to task-block-free (rsTBF). In figure 7.4, the behavior description of “Process RA-message” is shown.
7.5 Process RWR-message

When a local task wants to access to a remote resource the block "Remote Resource Request Handling", as described in section 4.7 on page 71, will send a remote-wait-command (RWC) message onto the network towards to the target resource containing the whole request. If the request has been executed successfully or ended in error, but needs to be acknowledged, the target \( M\mu P \) handling the resource will send the RWR-message back including the RIN. Therefore, when an RWR-message is received, it can only be intended for a local task. The behavior description of "Process RWR-message" is shown in figure 7.5a.

```
1. IF (RIN known)
2.   IF (RIN matches to TL) {
3.     Send RA-message onto network (RIN);
4.     Process_Result(RIN);
5.   }
6.   ELSE
7.     Update_SW_Send_RWR (RIN);
8.   }
9. ELSE
10.   IF (received TN matches to TL) {
11.      Send RA-message onto network (TN);
12.      Process_Result (TN);
13.   }
14.   ELSE
15.     Update_SW_Send_RWR (RIN);
```

Figure 7.5a. Behavior description of "Process RWR-message".

When a local task had sent an RWC-message onto the network towards the remote \( M\mu P \) handling the resource, two things can happen. Firstly, the request can be executed at once after which the remote \( M\mu P \) will send the RWR-message back. In this case, the RIN is included in the RWR-message, but it is not known on the local TL. For checking the reception, the received TN of included in the message must be checked whether it matches with the TN on the TL in combination with the task's state. In this case, the task's state must be equal to tsWCA. Secondly, the request has been placed on the remote waiting list RRTL. On the local TL, the task's state is equal to tsWRR or to tsCRL. So, the RIN must be known on the TL. Therefore, it is kept on the field "TL(TN).Request.Parameter 2".

When receiving the RWR-message, the TN of the local task sending is included in the message. If the task's state is equal to tsWRR or to tsCRL, the received RIN must checked compared with the RIN on the TL, like "IF (TL(TN).Request.Parameter2 == RWR-message.RIN". If the RIN does match with the RIN on the TL, the result-acknowledgement (RA) message including the RIN is sent on the network to confirm the reception. After that, the subroutine "Process Result (RIN)" is called to send the result back to the local task and to wake up the task. Its behavior is described in figure 7.5b on the next page. This is done in line 3 and in line 4. If the RIN does not match, the subroutine "Update_SW_Send_RWR(RIN)" is called for sending the RWR-message back after the SW has been updated indicating what went wrong. This is done in line 7. Its behavior is described in figure 7.5c on the next page.

If the RIN is unknown on the TL, because the task's state is equal to tsWCA, the TN included in the received RWR-message must be compared with the TN on the TL. If they match, the subroutine "RA-message is sent onto the network to confirm the reception and the subroutine "Process_Result" is called to send the result back to the local task and waking up the task. If the received TN does not match on the TL, it means an error has been occurred. Therefore, the subroutine "Update_SW_Send_RWR" is called. For more information, see figure 7.5b and figure 7.5c on the next page.
1. IF (SW.ERR == 0) {
2.     Sending_Result_TO_Local_Task (TN);
3.     Wake_UP (TN);
4. }
5. ELSE {
6.     IF (SW.RIA == 1) {
7.         TL(TN).State = tsWRR;
8.         TL(TN).TO = 0;
9.     }
10. ELSE {
11.     Sending_Result_TO_Local_Task (TN);
12.     Wake_UP (TN);
13. }

Figure 7.5b. Behavior description of “Process Result”.

When the RWR-message is received, the SW included in the message must be checked. If the ERR-bit on the SW is equal to 0b, it means the request has been executed successfully. Therefore, the subroutine “Sending Result To Local Task” is called to send the result back to the local task sending the request. Its behavior is already described in subsection 4.4.1 on page 32. After that, the subroutine “Wake_Up” is called to wake up the task. Its behavior is already described in subsection 4.4.2 on page 32. Else, it means an error has been occurred. As shown in figure 4.3.2c on page 21, bit number 4 down to bit number 0 are used to specify the error. If the RIA-bit on the received SW is equal to 1b, it means the local Mf.lP had sent the CIA-message to check to the remote resource liveliness after the remote Mf.lP had sent the RWR-message back for acknowledgement. Therefore, if the RIA-bit is equal to 1b, it means the requested resource is still alive. Therefore, the task's state is set to tsWRR indicating waiting-for-remote-resource and the TO-bit is set to 0 to reset the message timeout time. When the RIA-bit is equal to 0b, a real error has been occurred. Whether the error is, the subroutine “Sending Result TO Local Task” is called to send the result back to the local task. Its behavior is already described in subsection 4.4.1 on page 32. In addition, the subroutine “Wake Up” is called to make the local task runnable, because it is blocked. Its behavior is already described in subsection 4.4.2 on page 32.

1. SW.ERR = 1;
2. SW.TAE = 1;
3. SW.ITN = 1;
4. Send RWR-message onto network (received_TN);

Figure 7.5c. Behavior description of “Update SW_Send_RWR”.

Because an error has been occurred, the SW is updated indicating what went wrong. In this case, the ITN-bit is set to 1b indicating the incorrect TN. This SW does not belong to any task, it is just used to send back to the remote Mf.lP that had sent the RWR-message to let it know what went wrong. After that, the RWR-message including the updated SW is sent onto the network towards the Mf.lP sending the RWR-message.

7.6 Process PWL-message

When the placed-on-waiting-list (PWL) message is received, it means the RWC-message local task had sent has been received successfully. However, because the requested remote resource is not available for the moment, its request is placed onto the remote RRTL. Therefore, the remote Mf.lP handling the resource had sent the PWL-message back telling the local Mf.lP handling the local task that its request is placed on RRTL waiting for execution. For updating the administration, the task’s state is changed to waiting-for-remote-resource (tsWRR) and the message timeout time must be cancelled. In figure 7.6 on the next page, the behavior description for “Process PWL-message” is shown.
In each message, the TN is included indicating for which task the message is intended. If this TN is not in use on the TL, it means the remote MµP had included a wrong TN or the TN of the target local task had been removed. Therefore, the RWR-message is sent onto the network including the SW, which has been updated to indicate what went wrong. In this case, the ITN-bit is set to 1b indicating the incorrect TN.

If the TN does in use, the task’s state is set to tsWRR indicating waiting for the remote resource. Because the message timeout time has been set after the RWC-message was sent, the message timeout time is now reset and or a new timeout time is set. When the request is placed on the remote waiting list RRTL, the remote MµP must include the RIN in the PWL-message for sending back to the local MµP handling the local task. Therefore, the received RIN is stored on the field “Parameter 2” on the TL.

### 7.7 Process EWA-message

When a local task had sent the RWC-message onto the network for accessing a remote, it can send the CW-message to cancel its request. After the remote MµP had cleaned up the task’s block on its RRTL, the MµP will send the EWA-message back to the local MµP to confirm the request. When the end-wait-acknowledgement (EWA) message is received, the local waiting task has to be made runnable and the result must be sent back to the local task sending the request. The behavior description is shown in figure 7.7.

In each message the TN is included indicating for which task the message is intended. If this TN is not in use on the TL, it means the remote MµP had included a wrong TN or the TN of the target local task had been removed.
Therefore, the RWR-message is sent onto the network including the SW, which has been updated to indicate what went wrong. In this case, the ITN-bit is set to 1b indicating the incorrect TN.

If the TN does in use, the message timeout time is cancelled by setting the TO-bit on the TL to 0. For sending the result back to the local task sending the request, the subroutine "Sending Result To Local Task" is called. Its behavior is already described in subsection 4.4.1 on page 32. In addition, for waking up the task, the subroutine "Wake Up" is called. Its behavior is already described in subsection 4.4.2 on page 32.
8. LOCAL TASKS' LIVELINESS CHECKING

When local tasks had requested for accessing to remote resources and the requested resources are not available temporarily, their requests are placed onto the remote RRTL. Because tasks can be removed at any moment, the TCU on each MμP must check the liveliness of the remote resources the local tasks had requested by sending the check-if-alive (CIA) message on the network. The intention is let the concerning remote MμP know whether the local tasks are still alive.

On each MμP, the TCU has to keep the administration for 63 local tasks. Because the liveliness check will be executed simultaneously with the controller "Execution" and because the TL is needed for accessing, the check will be executed for at most 16 local tasks at a time. The check will be done at a fixed interval, which is at least twice the time that the MμP will send the CIA-message. In figure 8, the communication between the local TCU and the remote TCU is shown.

For accessing to a remote resource, the block "Remote Resource Request Handling" will send the RWC-message including the local task's request onto the network towards the remote MμP handling the resource. If the request is executed immediately, the remote MμP will send the RWR-message including the result back. To confirm the reception, the local TCU sends the RA-message back indicating the result has been received successfully. When the requested resource is not available, the remote MμP sends the PWL-message back indicating the request is placed on the remote RRTL. When the local task has waited for a while, the local TCU sends the check-if-alive (CIA) message onto the network. If the remote resource is still existed, but still not available, the remote MμP sends the RWR-message message including the SW back. When the requested resource had been removed, the RNE-bit is set to 1b indicating the resource does not exist anymore.

8.1 Behavior description

It is assumed the timers generator will set the LC-bit on the E-register to 1b when the interval time for liveliness checking is expired. By checking the LC-bit, the block "Local Tasks' Liveliness Checking" can determine when it has to send the CIA-message.

However, before sending the CIA-message onto the network, the block's state must be checked. As shown in figure 4.3.4c on page 26, there are eight local task's states defined. The states tsRTR, tsWTO, tsSL and tsWLR are intended for local accesses. In the state tsWEWA, the task is waiting for the EWA-message. In the state tsCRL, it means the CIA-message is already sent for checking the remote resources liveliness. Only in the state tsWRR or in the state tsWCA, the task is waiting for the remote resources. Therefore, the CIA-message must be sent onto the network.

As written above, the TL keeps the administration for 63 local tasks. To keep the check short and because of the length of the outgoing message buffer on the Network Manager, the TL is divided into sixteen sections each
consists of 4 tasks. Only the sixteenth section consists of only 3 tasks. It means, each time the timeout time has been expired, only 4 tasks on the TL will be checked to determine whether a local task is waiting for a remote resource. If so, the CIA-message will be sent onto the network to check the resource liveliness. To be able to keep which section has been checked, a 4-bits register called LC_Block is used. In figure 8.1, the behavior of the block “Local Tasks’ Liveliness Handling” is shown.

```c
1.    WHILE (!E-register.LC);
2.    FOR (index_TN = 0; i < 4; i++) {
3.        next_TN = LC_Block * 4 + index_TN
4.        IF ((TL(next_TN).State == tsWRR) OR (TL(next_TN).State == tsWCA)) {
5.            IF (NOT((index_TN == 3) AND (LC_Block == 15))) {
6.                Send CIA-message onto network (next_TN);
7.                TL(next_TN).State = tsCRL;
8.                TL(next_TN).TO = 0;
9.            }
10.        }
11.    }
12.    LC_Block++;
13.    E-register.LC = 0;
```

Figure 8.1. Behavior description of “Local Tasks’ Liveliness Handling”.

As long as the LC-bit on the E-register is low, the block “Local Tasks’ Liveliness Handling” will not be executed. Because each LC_Block consists of 4 tasks, except in the sixteenth LC_Block, the for-loop is used. Further, to get a short notation, the temporal register next_TN is used.

IF LC_Block is equal 0, the register next_TN in line 3 is equal index_TN. For LC_Block = 1, the register next_TN is equal to 4 + index_TN. For LC_Block = 2, the register next_TN is equal to 2*4 + index_TN. In addition, for LC_Block = 3, the register next_TN is equal to 3*4 + next_TN. Because the sixteenth LC_Block consists of only 3 tasks, the last task may not be handled according to the condition in line 5.

For execution, the code in line number 6 up to line number 8 is executed. So, the CIA-message is sent onto the network towards the remote MuP handling the resource. For keeping the administration, the task’s state is set to tsCRL indicating checking for resource liveliness. For setting a new timeout time for the sent message, the TO-bit is set to 0b. The principle of the TO-bit is already described in subsection 4.7.1 on page 73.

For updating the register LC_Block, this register is increased by one after the for-loop is executed completely. As last, the LC-bit on the E-register is set to 0b so that a new message timeout time can be set.
9. LOCAL TASKS' MESSAGE TIMEOUT HANDLING

On a multi-MµP system, the TCU on each MµP can send different messages onto the network to communicate with other MµP's. Especially when local tasks want accessing to remote resources and when remote tasks want accessing to local resources, different messages have to be sent onto the network. Because the communication can get lost somehow, the timeout is used to detect the communication losses. For handling the remote tasks’ message timeout, it is described in chapter 10 “Remote Tasks’ Message Timeout Handling” on page 91. For handling the local tasks’ message timeout, it is described in this chapter.

For the moment, the TCU keeps the administration for 63 local tasks on the TL. Avoiding handling the local tasks’ message timeout will last too long in each cycle of the controller “Request Execution” and because of the length of the outgoing message buffer on the Network Manager, the TL is divided into 63 / 4 = 16 blocks each consists of 4 tasks for handling the local tasks’ message timeout. To be able to keep which block has been handled, a 4-bits register called local-message-timeout-block (LMT_Block) is needed. For initialization, this register is set to 0.

For handling the message timeout, it is assumed that the timers generator will set the LMT-bit on the E-register when the timeout time is expired. This timeout time can be initialized to a certain value. When the timeout time is expired, the TO-bit on the TL will be checked before updating this bit. Therefore, a message can be resent or the local task's request can be removed from the TL. In figure 9, the behavior of this block is described.

1. FOR (index_TN = 0; index_TN < 4; index_TN++) {
2.     next_TN = 4 * LMT_Block + index_TN;
3.     IF ((TL(next_TN).Block_State != tsIDLE) OR
4.             (NOT((next_TN == 3) AND (LMT_Block == 15)))
5.             IF (TL(next_TN).TO == 0) // First timeout time occurrence
6.                 TL(next_TN).TO = 1;
7.             ELSE {
8.                 TL(next_TN).TO = 0;
9.                 switch (TL(next_TN).State) {
10.                    case tsWRR: Send CIA(next _TN) onto network;
11.                        TL(next _TN).State = tsCRL;
12.                        break;
13.                    case tsWCA: Sending_Result_To_Local_Task(next _TN);
14.                        Wake_Up(next_TN);
15.                        break;
16.                    case tsCRL: Re-send CIA(next _TN) onto network;
17.                        TL(next _TN).State = tsCRLC;
18.                        break;
19.                    case tsCRLC: Sending_Result_To_Local_Task(next _TN);
20.                        Wake_Up(next_TN);
21.                        break;
22.                    case tsWEWA: Send CW(next _TN) onto network;
23.                        Sending_Result_To_Local_Task(next _TN);
24.                        Wake_Up(next_TN);
25.                        break;
26.                    default: break;
27.                 }
28.             }
29. }
30. E-register.LMT = 0;

Figure 9. Behavior description of “Local tasks' message timeout handling”.

Because the TL is divided into 16 blocks each with 4 blocks as done in chapter 8 for checking the local tasks' liveliness, the same principle is used for handling the local tasks' message timeout.

For handling the 4 tasks of each block, the for-loop is used. In line 2, the temporal register next_TN is used to get a short notation. Further, only occupied tasks' blocks will be handled, which block's state is not equal to tsIDLE or when the register next_TN is not equal to the fourth task of the sixteen block that is referred by the register LMT_Block.
If the TO-bit is equal to 0b, the timeout time for the current task has been occurred for the first time. So, only its TO-bit is needed to set to 1b. This is done in line 5. Else, it means the timeout time for the current task has been occurred for the second time. Therefore, its TO-bit is set to 0b to start a new timeout time for this task. Next, the block’s state is checked before taking any action. This is done in line 9 up to line 24. For checking the task’s state, the case-statement of the C-language is used so that only one state can be selected.

In the state tsWRR, it means the local task had waited for a while for the remote resource. Therefore, the CIA-message is sent onto the network to check the liveliness of the remote resource and the task’s state is set to tsCRL. If the task’s state is equal tsWCA, it means the RWC-message had been sent for a while, but the response is still not received. It can be concluded that the MuP handling the remote task is down or the communication link is broken. Therefore, the subroutine “Sending Result To Local Task” is called to send the result back to the local task after the SW has been updated to indicate what went wrong. In this case, the NR-bit on the SW is set to 1b indicating no response. Also the ERR-bit has to set 1b indicating an error has been occurred. Its behavior is already described in subsection 4.4.1 on page 32. After that, the subroutine “Wake Up” is called to make the task runnable. Its behavior is already described in subsection 4.4.2 on page 32.

In the state tsCRL, it means the CIA-message had been sent to check the remote resource liveliness. Therefore, the CIA-message is re-sent onto the network after which the task’s state is changed to tsCRLC indicating checking-resource-liveliness-checked. When in the state tsCRLC, it can be concluded that the requested resource does not exist anymore. Therefore, the subroutine “Sending Result To Local Task” is called to send the result back to the local task after the SW has been updated to indicate what went wrong. In this case, the RNE-bit on the SW is set to 1b indicating the resource does not exist. Also, the ERR-bit and the RAE-bit have to set to 1b. After that, the subroutine “Wake Up” is called to make the task runnable.

As last, in the state tsWEWA, it means the CW-message had been sent onto the network towards the remote MuP handling the resource requesting to cancel its request. Because no response had been received so far, the CW-message is re-sent onto the network. However, its response is not awaited anymore. Therefore, the subroutine “Sending Result To Local Task” is called to send the result back to the local task after the SW has been updated to indicate what went wrong. In this case, the NR-bit on the SW is set to 1b indicating no response. Also the ERR-bit has to set 1b indicating an error has been occurred. After that, the subroutine “Wake Up” is called to make the task runnable.
10. REMOTE TASKS’ MESSAGE TIMEOUT HANDLING

For communication with remote MIP’s, there are different messages sent onto the network. Because the communication can get lost somehow, the timeout time is used to detect the communication losses. On each block of the RRTL, there is a field named “TO”. It is one bit in width and is used to indicate the number of timeout time that has been occurred when the message for a remote task had been sent onto the network. The principle of the TO-bit is already described in subsection 4.7.1 on page 73. Further, the field “Block State” and the bit remote-message-timeout (RMT) on the E-register are needed to be able to process the message timeout.

For the moment, the TCU keeps the administration of up to 16K – 64 remote tasks. As for handling the local tasks’ message timeout that is already described in chapter 9 on page 89, the RRTL will be divided into ((16K – 64) / 4 =) 4K blocks each consists of 4 tasks for handling the remote tasks’ message timeout. To be able to keep which block has been handled, a 12-bits register called RMT_Block is needed. For initialization, this register is set to 0.

For handling the message timeout, it is assumed that the timers generator will set the RMT-bit on the E-register when the timeout time is occurred. This timeout time can be initialized to a certain value. When the timeout time is occurred, the TO-bit on the RRTL will be checked. Therefore, a message can be resent or the remote task’s request can be removed from the RRTL. In figure 10, the behavior of this block is described.

```
FOR (index_TN = 0; index_TN < 4; index_TN++) {
    next_TN = 4 * RMT_Block + index_TN;
    IF (RRTL(next_TN).Block_State != rsTBF) {
        IF (RRTL(next_TN).TO == 0) // First timeout time occurrence
            RRTL(next_TN).TO = 1;
        ELSE { // Second timeout time occurrence
            RRTL(next_TN).TO = 0;
            switch (RRTL(next_TN).Block_State) {
                case rsRS: Re-send RWR(next_TN) onto network;
                    RRTL(next_TN).Block_State = rsRSC;
                    break;
                case rsRSC: Re-send RWR(next_TN) onto network;
                    Removing_task_from_RRTL(next_TN);
                    break;
                case rsOWLA: RRTL(next_TN).Block_State = rsOWLC;
                    break;
                case rsOWLC: Send RWR(next_TN) onto network;
                    RRTL(next_TN).Block_State = rsOWLS;
                    break;
                case rsOWLS: Removing_task_from_RRTL(next_TN);
                default: break;
            }
        }
    }
}
E-register.RMT = 0;
```

Figure 10. Behavior description of "Remote tasks' message timeout handling".

Because the RRTL is divided into up to 4K blocks each with 4 blocks as done in chapter 8 for checking the local tasks’ liveliness, the same principle is used for handling the message timeout.

For handling the 4 tasks of each block, the for-loop is used. In line 2, the temporal register next_TN is used to get a short notation. Further, only occupied tasks’ blocks will be handled, which block’s state is not equal to rsTBF.

If the TO-bit is equal to 0b, the timeout time of the current has been occurred for the first time. So, only its TO-bit is needed to set to 1b. This is done in line 5. Else, it means the timeout time for the current task has been occurred for the second time. Therefore, its field “TO” is set to 0b to start a new timeout time for this task. Next,
the block's state is checked before taking any action. This is done in line 9 up to line 21. For checking the block's state, the case-statement of the C-language is used so that only one state can be selected.

In the state rsRS, it means the RWR-message had been sent. Therefore, the RWR-message is re-sent onto the network and the block's state is set to rsRSC. If the block's state is equal rsRSC, it means the RWR-message had been re-sent, but the RA-message is still not received. It can be concluded that the MiP handling the remote task is down or the communication link is broken. Therefore, the RWR is re-sent onto the network for the third time, but its response is not awaited. Therefore, the subroutine “Removing task from RRTL” is called to remove the remote task’s request from the RRTL. Its behavior is described in section 10.1.

When a remote task is waiting for local resource, its block's state is equal to rsOWLA, rsOWLC or equal to rsOWLS. When in the state rsOWLA and the timeout time is occurred, only the block's state is needed to change to rsOWLe. When in the state rsOWLC, the RWR-message is sent onto the network towards the MiP handling the remote task after the SW has been updated indicating the requested local resource is still alive. In this case, the TIA-bit on the SW is set to 1b. Of course, the ERR-bit and the TAE-bit on the SW are set to 1b. After that, the block’s state is changed to rsOWLS. When in the rsOWLS, it can be concluded that the MiP handling the remote task is down or the communication link is broken. Therefore, the subroutine “Removing task from RRTL” is called to remove the remote task’s request from the RRTL. Its behavior is described in section 10.1.

### 10.1 Removing a task from the RRTL

When remote task’s request must be removed from the RRTL, the corresponding block will be cleaned up by calling the subroutine "Removing task from RRTL". Its behavior is described in figure 10.1.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. RRTL(RIN).Block_State</td>
<td>= rsTBF;</td>
</tr>
<tr>
<td>2. RRTL(RIN).Prev_TN</td>
<td>= NIL;</td>
</tr>
<tr>
<td>3. RRTL(RIN).Next_TN</td>
<td>= Free_Block;</td>
</tr>
<tr>
<td>4. Free_Block</td>
<td>= RIN;</td>
</tr>
</tbody>
</table>

*Figure 10.1. Behavior description of “Removing task from RRTL”.*

Because the RRTL is indexed by the pointer Free_Block, from which the RIN is determined, the RIN is used.

When cleaning up a task's block, the task's block is set to rsTBF indicating a free block. For keeping the administration, the field “Prev_TN” is set to NIL. Maybe it is not needed to set to NIL, because when removing a task from the local resource waiting list, it is already done. As last, the field “Next_TN” is set to Free_Block for chaining the free blocks and the pointer Free_Block is set to the RIN of the removed block. On this way, the upper part of the RRTL can be used often. The advantage is that when searching for a certain request on the RRTL, the search can be kept short.
11. THE NETWORK MANAGER

As the third block on the TCU, the block Network Manager is needed using for external communication in a multi-MuP's system. In a single MuP system this block not needed. In figure 10, the global architecture of the Network Manager is shown using a communication protocol similar to that of the Transputer T9000.

When a local task wants to access a remote resource, it can send the RWC-message including its request to the Network Manager. When the message has been received by the block “Encode & Decode Message”, it will be encoded into a packet. For more information about packets, see the next section “Encoding & Decoding Message”. Next, the block “Link Manager” will send the packet (encoded message) to the target MuP handling the resource. Therefore, the Link Manager will use the target address that is included in the message. For sending, the Link Manager can use one of the four links, numbered from one to four. The intention of these is realizing a network so that all MuP’s can communicate with each other.

The four links are already implemented using IDaSS. Because of time shortage, the block “Encode & Decode Message” and the block “Link Manager” are still not designed and or implemented.
12. CONCLUSIONS & RECOMMENDATIONS

After the nucleus is analyzed, a conceptual specification for the Task Control Unit (TCU) is formed. This is described in chapter 3 on page 9. As shown in figure 4a on page 15, the TCU is divided into the TaskQ for handling of runnable tasks, the Network Manager for handling of all off-core communications and the Core Unit for handling and executing of all local and remote requests. Because all blocks are already implemented using IDaSS, except the block “Execution Core” of the Core Unit and the blocks “Encode & Decode Message” and “Link Manager” of the Network Manager, it is concentrated to design the Execution Core. This is the most important block of the Core Unit. Therefore, a conceptual architecture for the Execution Core is made, as shown in figure 4.2b on page 18.

A functional design for the Execution Core has been completed. Only the behavior of all blocks is described using flowchart diagrams and or pseudo-code. The intention is that the Execution Core can be implemented easily in the future by mapping the behavior description directly to the programming code of a certain language or in hardware.

For future works, the block “Encode & Decode Message” and the block “Link Manager” of the Network Manager have to be designed. The block “Encode & Decode Message” has to encode every outgoing message to the defined data packet, similar to the communication protocol of the Transputer T9000. After that, the block “Link Manager” will determine via which link the message will be sent. For incoming messages, the block “Encode & Decode Message” has to decode the received data packet into a message format. By sending the message to the block “Message Handling”, the message will be processed.

When designing the Execution Core, it is assumed that some blocks are implemented and will operate according to a certain behavior. Especially, the block “Command Receiver”, the block “Activity Detection”, the block “Result transmitter”, the block TaskQ and the whole Network Manager. For implementing the TCU completely, some compromises must be made.

For the moment, only short semaphores, long semaphores, mailboxes and pipes used. Another communication resource is the long mailbox. The standard mailbox contains up to 16 messages each is 32 bits in width. When a longer mailbox is needed, the long mailbox can be used. Like the pipe, the long mailbox will store its messages in the main memory. On the resource list, the corresponding resource data block will be used to store the pointer that refers to the location in the main memory. For indicating the resource type long mailbox, the value 11b of the field “RT” can be used. For the moment, this value is not in use.

Because of the complexity and the flexibility of the pipe, a complete protocol for the pipe must be written in the future.

For the moment, all instructions except for accessing to communication resources, as shown in figure 3.7 on page 14, are send sent by local tasks. In the future work, it is recommended to make all instructions transparently. It means that all instructions can be sent by both the local and remote tasks.
REFERENCES

[Verschueren-1]
A.C. Verschueren
(http://www.ics.ele.tue.nl/~ad/isass.html)

[Verschueren-2]
A.C. Verschueren
Multimicroprocessor task control unit, the networked multiprocessor option, Internal note, Eindhoven University of Technology, ICS-section August, 1999

[Geurts]
A.G.M. Geurts

[Mitchell]
D.A.P. Mitchell, J.A. Thompson, G.A. Manson, and G.R. Brookes

[Theelen]
B.D. Theelen

[Nedea]
S. Nedea

[Loo]
T. van Loo
The Task Control Unit of the Multi Micro Processor, Concept and improvements, Report on a period of practical training, Eindhoven University of Technology, ICS-section May, 1999