OS.8MT ORIENTATION

Anders Wollbeck 831220

****
THE COMPUTER HARDWARE
****

This chapter will take a look at computer hardware in general and then make references to the DataBoard system.

THE BASIC COMPUTER HARDWARE

Most computers consist of some basic parts:

- A CENTRAL PROCESS UNIT (CPU).

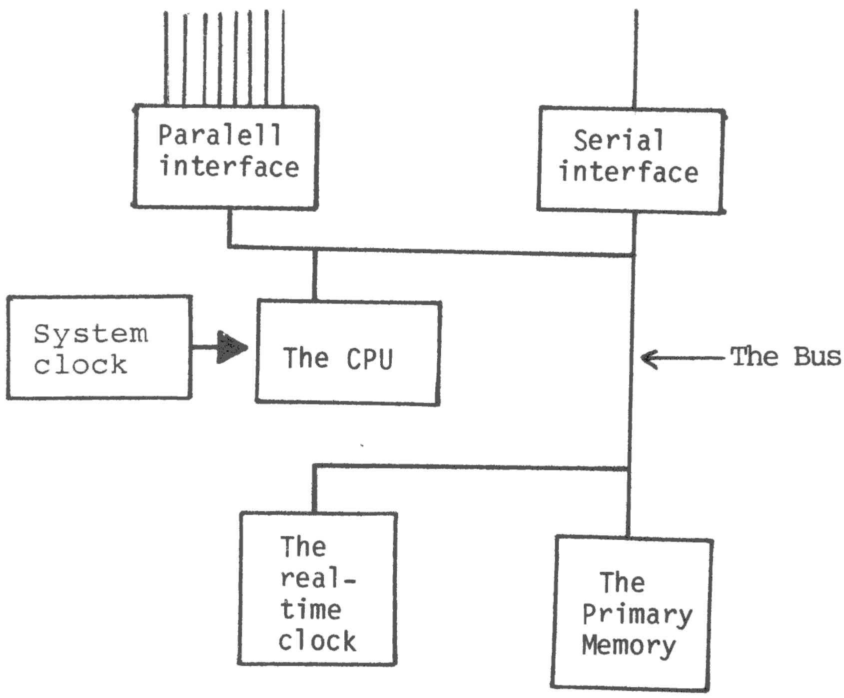
- A SYSTEM CLOCK.

- A PRIMARY MEMORY.

- A REAL TIME CLOCK.

- Various INTERFACES to external equipment.

- A system of lines which connect the different parts of the computer. This is often called the BUS of the computer.



Pic 1.1 The basic hardware of a computer.

THE CPU

The CENTRAL PROCESS UNIT (CPU) is the heart and controller of the computer. It consists physically of registers and logic gates.
The CPU recognizes a number of INSTRUCTIONS. An instruction is a combination of 8, 16 or 32 bits. (The number is dependent on the complexity of the instruction). Every instruction means something special to the CPU, and makes it perform certain actions.
The execution time of an instruction increases by the complexity of it.

00111100	"Add one to the contence of regester A"
10000110 00000011	"Add to register A.... ....the binary value 00000011 (3 dec)"
11000011 00000011 01010100	"Jump to the address..... ....00000011 01010100 in the primary memory"

Pic 1.2 Some examples of instructions, and their meaning to the CPU which in this case is the Z80 used in the DataBoard system.

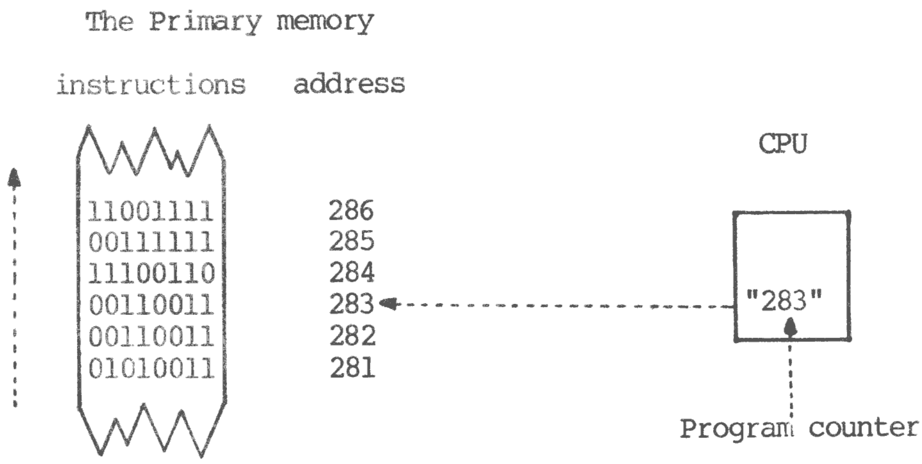
If we have a sequence of instructions which performs a sequence of actions we have a PROGRAM.

THE SYSTEM CLOCK

The CPU executes instructions at a certain pace. This pace is determined by the SYSTEM CLOCK. The pulses from the clock are also used in several other parts of the computer in order to synchronise all actions.
The clock frequence in DataBoard systems are 2 - 4 MHz.

THE PRIMARY MEMORY

The program is stored in a PRIMARY MEMORY, which can be looked upon as a long row of spaces where instructions are held. Each space can be reached by means of an address.
The CPU takes one instruction at a time and executes it, the program is running. Programs often consist of data as well as instructions. The part of the program where data are stored is called the WORK AREA of the program.
The CPU knows where to read instructions in the primary memory by keeping a PROGRAM COUNTER which holds the address of the next instruction in the program.



Pic 1.3 A part of the primary memory. The Program counter shows the next instruction the computer will execute.

When we in the following text discuss the primary memory it is often more convenient to look upon it as an "area" of instructions instead of a "row" of instructions.

THE INTERVAL CLOCK

The interval clock is mainly used in two ways by the computer.

- Updating a software REAL-TIME clock acting as a "stop watch".

- The interval clock may also hold the real-time (Year, Month, Day, Hour, Minute, Second), acting as a referance to the software real-time clock when the computer is started.

INTERFACES

In order to communicate with the world outside, the computer uses interfaces which can be of different types:

- Digital inputs and outputs.

- Analogue to digital (A/D) converters which convert a referance current to a digital value which the computer can handle.

- Digital to analogue (D/A) converters which does the reverse function as an A/D.

- Serial inputs and outputs. Data is transferred "one bit after another" according to different rules.

- Parallel inputs and outputs. Data is transferred several bits at the same time. The number of bits is dependent on the number of lines.

THE BUS OF THE COMPUTER

The bus is used to transfer instructions and data between the different components in the computer. You can make a distinction between the DATA BUS which transfers data and the ADDRESS bus which gives the address of the component in the computer to which the data is transferred.
The CONTROL BUS is a number of lines, each transferring control information between the components.

BUS STANDARDS

There exist several different standard buses on the market today, like Multibus, S100 bus and the DataBoard bus which is used by ABC computers, Facit computers, Monroe computers and, naturally, DataBoard computers.

PERIPHERALS

By using interfaces different peripherals can be connected to the computer.
Some common peripherals are:

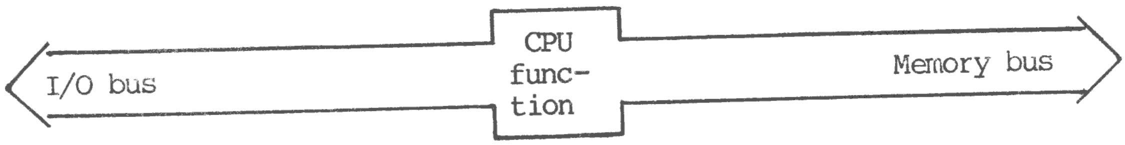
- One or more KEYBOARDS, DISPLAYS and TERMINAL DEVICES with which the operator can communicate with the computer.
- EXTERNAL MEMORY by means of floppy discs, hard discs, magnetic tape etc.
- PRINTERS of different kinds.

THE PHILOSOPHY OF THE DATABOARD HARDWARE

The DataBoard hardware is completely modular. The modules consist of eurocards, each having a specific function like CPU, memory, digital output, etc. The eurocards are connected by means of the DataBoard bus. Only the modules necessary for the application are needed.

THE DATABOARD BUS

The DataBoard bus is divided into different CPU, memory and I/O parts.



Serial and analogue interfaces, Digital and analogue I/O, cards for special purposes etc.	Z80 CPU board with control board, one board comp- uter etc.	RAM memory, CMOS RAM mem- ory, EPROM memory, EEROM memory, DMA and MAC function (see below) etc.

Pic 1.5 The DataBoard bus and some examples of the available functions. All fuctions are modular and can be changed to taylor the system according to the requirements of the application.

ESSENTIAL HARDWARE

The minimum environment for OS.8MT is:

- Z80 CPU board with control board or single board computer.
- A minimum of 48 kb memory.
- A real time interval clock.

This is the minimum system. In this form it can for example be used as a dedicated system.

THE PRIMARY MEMORY

The primary memory can consist of a mixture of RAM (regular and CMOS with battery backup) and PROM memory boards.
The Z80 CPU has 16 address lines and can thus address only 64 Kbytes of memory. To overcome this shortage a MEMORY ACCESS CONTROLLER (MAC), can be added, which along with memory boards expand the primary memory to max 256 Kbytes.
The amount of memory needed for a certain application depends primary on how many OS modules you have to include and the number and size of the tasks you use.
The DataBoard System manual includes information about selecting the right memory boards.

PERIPHERALS

Some examples of peripherals which can be used with the DataBoard system are:

- One or up to 6 TERMINAL DEVICES using V24 (RS232) interface boards (Up to 19 600 asynchronous protocols). You can actually use almost any amount of terminal devices. The limit is mainly set by the activity at each terminal.
- HARD DISC drives, WINCHESTER drives and 5" as well as 8" floppy disc drives may be combined in steps from 80 Kb to 400 Mb.
  A DIRECT MEMORY ACCESS-board (DMA) is available for use with most types of mass memory.
- MAGNETIC TAPE and CASSETTE stations.
- High speed LINE PRINTERS (spooling is available in the system.
- High speed PAPER TAPE READER and PUNCH.
- CARD READER.
- MODEMS and DIAL-UP UNITS.
- DIGITAL and ANALOG INPUTS and OUTPUTS.
- Synchronous and asynchronous COMMUNICTION INTERFACES.

OS.8MT IS PROMMABLE

It is possible to place OS.8MT in EPROM, thereby eliminating the need of mass storage memory. As Assembler, Basic, Fortran and Pascal code also can reside in EPROM you have many options when it comes to stand-alone systems.

SUMMARY

OS.8MT is equal at home in a dedicated single board computer as in a large mini-computer resembling development system.

****
THE SOFTWARE OF THE COMPUTER
****

This chapter will include a brief discussion of computer software.

THE PROGRAM

A computer program consists of instructions and data. The instructions make the CPU perform certain actions like: "Fetch the contents of memory location 67898 and put it into the A register"."Add the contents of memory location 67898 to the contents of the A register". "Put the result in memory location 67898", etc.
The data can be of several types like numerical variables and pointers which indicate the location of data.

PROGRAM DEVELOPMENT

Writing programs working with binary numbers would be rather tedius work. There exist for this reason programming languages. Some common languages are Assembler, BASIC, Pascal and Fortran. We will here describe these languages very briefly.

ASSEMBLER

Assembler is a low level, machine oriented language. The programmer writes the program in mnemonics which is kind of a shorthand for binary instructions. From the mnemonics the program is assembled into binary code.
The programmer can also give instructions, making the assembly process perform in different ways.

```
Source code          Assembly process          Executable code

 .                                                   .
 .                                                   .
EQU  *                                              00011011
JFCS WRITE    -------------------------->          11011000
LDI  HL,0                                           01010011
STD  HL,ANTREC                                       .
 .                                                   .
 .
```

Pic 2.1 Assembler.

By programming in assembler you can totaly optimize the program and the result is a very fast execution. The programming time can, however, be long.

PASCAL

Pascal is a relativly new language with the aim to promote structured programming. The source code is compiled by a Pascal compiler to executable object code. Each instruction results in a number of binary instructions for the CPU.

```
Source code              Compilation              Executable code
 .                                                      .
 .                                                      .
FOR I:=1 TO 400 DO                                  00001001
  BEGIN                ---------------------->      01010011
    J:=J+1                                          01001001
 .                                                  01010010
 .                                                      .
                                                        .
                                                        .
```

Pic 2.2 Pascal.

FORTRAN

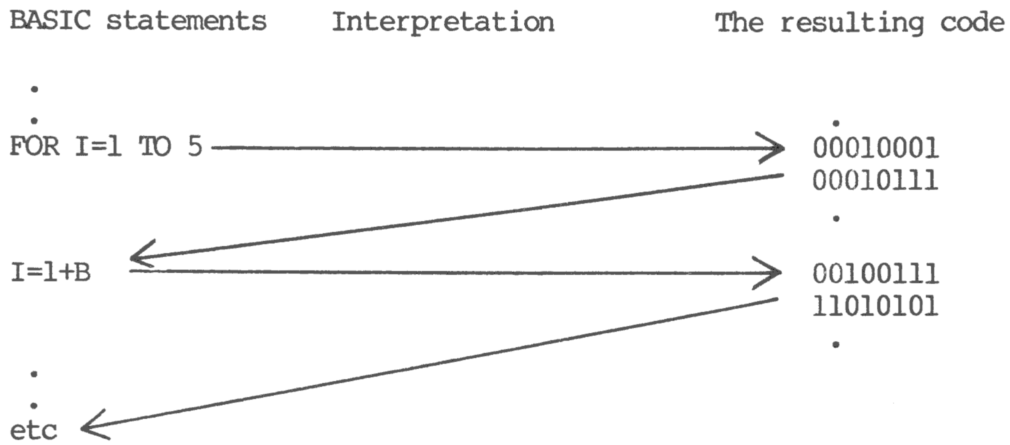
Fortran is an older compiling language. It is mainly used for numeric calculations. It uses the same compilation principle as Pascal.

```
Source code              Compilation              Executable code
 .                                                      .
 .                                                      .
     WRITE(6,100) TAL                               00010001
100  FORMAT(1H ,2F5.1)  -------------------->       11011011
 .                                                  11110101
 .                                                      .
 .                                                      .
```

Pic 2.3 Fortran.

BASIC

Basic is a easy to learn interactive language. Interactive means each statement is interpreted immediatly checking for syntax errors. This interpretation is also done while the program is running, requiring a BASIC interpretator.



Pic 2.4 BASIC.

While BASIC programs are easy to and fast to write the execution is generally slower than programs written in assembler or compiling languages.

LINKING PROGRAM MODULES

A program is often linked together from a number of different program modules. The linking process can be controlled in different ways.

```
               compilation
Source code ---------------> object module
               compilation                           executable
Source code ---------------> object module      linker -> program
               compilation
source code ---------------> object module
```

Pic 2.5 This program is linked together from three different modules.

PROGRAM CODE - A DEFINITION

Both instructions and data are often referred to as program code.

STACK

The stack is a row of memory locations where code can be saved by means of simple fast instructions, like "PUSH" (saving code) and "POP" fetching the last "PUSHED" code.
A STACK POINTER in the CPU shows the location of the stack.

DATA STRUCTURES

It is often very convenient to group data into structures of different kinds. This is also called DATA TABLES, DATA BLOCKS and in some cases NODES.
These structures can contain pointers which show the location of other data structures. Two very common data structures are LIST STRUCTURES and TREE STRUCTURES.

LIST STRUCTURES

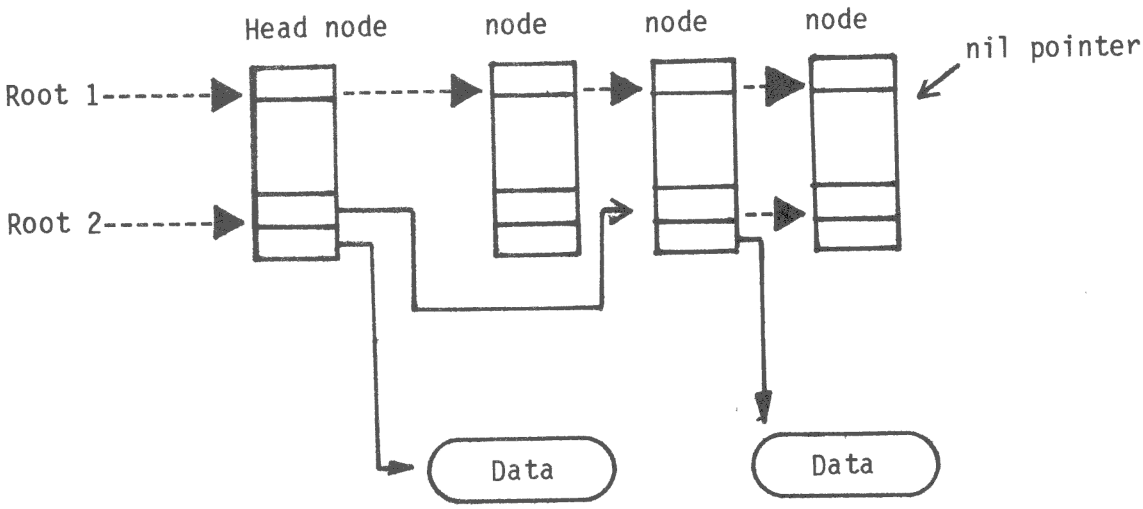
The list structure is a number of blocks which are tied together by means of pointers. The list structure consists of:

- A ROOT which points at the first block.

- BLOCKS the areas where data are stored and pointed to from. These blocks are often called NODES. The first node in a list structure is called the HEAD NODE.

- LINK FIELD, a pointer to another node in the same list structure. More then one link field can exist in a node.

- NIL POINTER, a pointer whose value indicates that nothing is pointed at.



Pic 2.6 A list structure which includes two roots and pointers to data.

A QUEUE is a list structure where the nodes are ordered in some way. Sometimes a list structure with no special order is also referred to as a queue.
From the block there can go pointers to other data structures. It is easy to reach data in the blocks and data pointed to from the blocks. Adding and removing blocks present no problem. If we want to use the blocks for another list structure which does not include all the blocks, we simply have to add another root and another link field in the blocks as shown in the picture above.

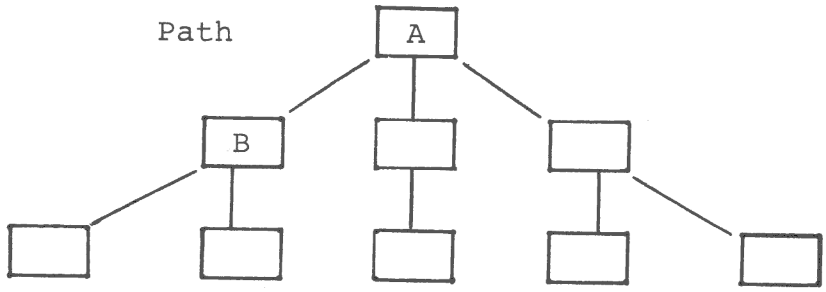
TREE STRUCTURES

The nodes can be linked in more ways than into a list structure. One important way is the TREE STRUCTURE. It means you have a top node which points at other nodes on a lower level, which in turn can point at nodes on an even lower level and so on. Some terms are:

- PARENT, the node which is on the level above the one you are viewing. The top node is the only node which has no parent.

- DESCENDENT, a node on the nearest level below the one you are viewing. The bottom nodes have no descendent.

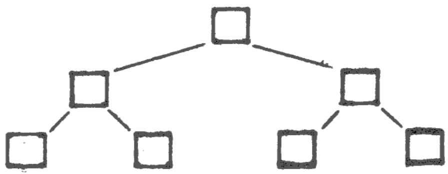
- PATH, the line made if you go from the top node to one of the bottom nodes. There exist as many paths as nodes that have no descendent.



Pic 2.7 An example of a tree structure. Node A is parent to node B. Node B is decendant to node A.

THE BINARY TREE

One common tree structure is the BINARY TREE, where each node can have only two descendents. It is often used when you want to structure data in some sort of order.



Pic 2.8 A binary tree.

ROUTINE - A DEFINITION

A number of instructions in program which does something special is often referred to as a routine.

SUBROUTINE - A DEFINITION

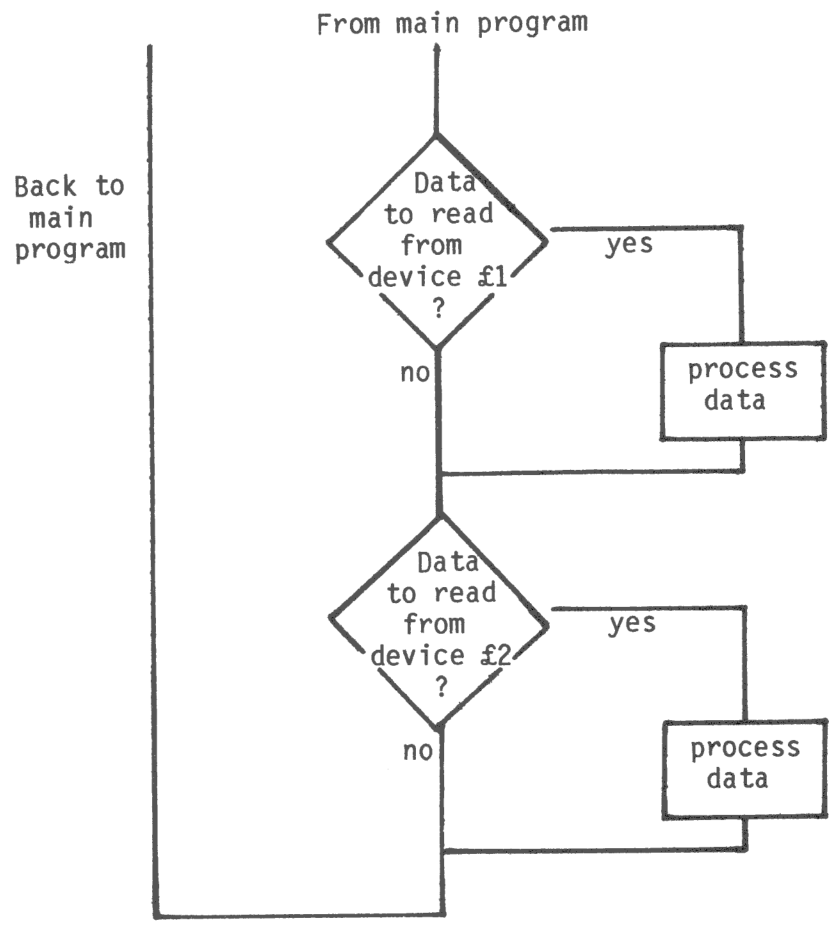
If a routine is used frequently in a program it is a good idea to collect them into a subroutine which can be entered from different points in the program.
Some of the advantages of using subroutines are that you do not have to duplicate the same code in a program. The program will also be better structured and easy to read.

HANDLER - A DEFINITION

A routine which does a specific thing is sometimes referred to as a HANDLER.

POLLING

When a peripheral needs to communicate with the computer it must have a method of telling the computer about it. One way is to scan the peripherals frequently and look if they possibly need attention. The scanning is done by a piece of code. This is called POLLING and is a simple but cumbersome method.



Pic 2.9 A polling routine.

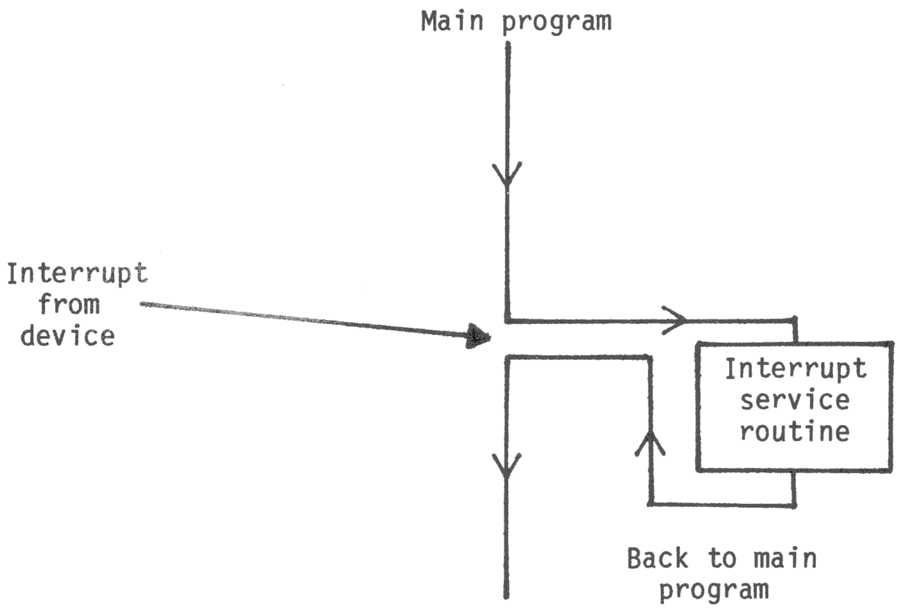
INTERRUPTS

A more sophisticated method is when a peripheral issues an INTERRUPT when it needs attention. The actual physical process of an interrupt going from a peripheral to the CPU differs on different machines, but the resorts taken after it has been received are the same.
What actually happens when an interrupt has been detected is:

1. The program executing instructions is suspended temporarily.

2. A jump is made to an INTERRUPT SERVICE ROUTINE which does the things necessary to service the interrupt.

3. When finished, the program continues.



Some interrupts are more important than others so we associate the interrupts to different INTERRUPT LEVELS. An interrupt on a higher level may interrupt a lower one, but not the opposite.

Pic 2.10 An interrupt

****
THE SMALL OPERATING SYSTEM
****

In this chapter we will discuss the motivation for having an operating system in a computer.

THE NEED OF AN OPERATING SYSTEM

There are a lot of routines, especially for the control of files and mass storage units, which are used by almost every program running on the computer.
If every program which runs on the computer should supply all these subroutines, a lot of programmer's time would be wasted. There is also a chance of errors in the code, possibly causing damage.
A better idea is to collect all these subroutines into a package which can be requested by the programs through standardised calls. This is the beginning of an operating system.
We will now take a look at the facilities of a small computer: The Monitor, the Error handler, the File management system and The Utilities.

THE FILE MANAGEMENT SYSTEM

In many computer applications you need to store data and programs on external memory. The normal way to do this is to use FILES. Each file can be reached by means of a DIRECTORY which also can hold information about:

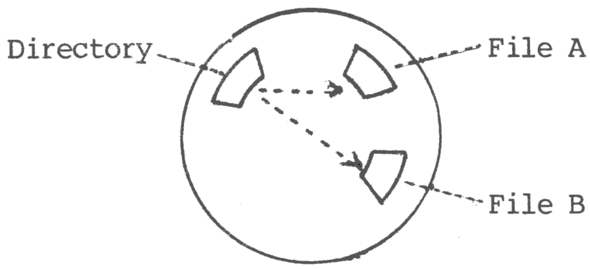
- The name of the file.

- Creation date and time for the file.

- When the file last was updated.

- What kind of data the file containes (ASCII, Binary, etc)

- Etc.



Pic 3.1 Every file can be reached from the directory.

The computer uses a number of routines to handle the files on external memory. This is called the FILE MANAGEMENT SYSTEM (FMS).
The FMS gives commands to the hardware which controls the external memory, so data can be written on and read from the right place on the disc.

THE MONITOR

The terminal device is mandatory for communicating with the machine. It needs routines to take care of the data going to and from it. Furthermore, it is convenient if the operator can give the computer COMMANDS to guide its actions. The program capable of decoding the commands and delegating what should be done is called the MONITOR.

ERRORS

The operator is like all humans bound to make errors while working with the machine. He can for instance give a bad command. Errors can also occur in the computer hardware or software which must be taken care of.
This is done by an ERROR HANDLER which gives the operator information about the error on the terminal device, often accompanied by an audible signal.

CRASHES

If the error is so severe that
there is a risk of data being destroyed, the computer CRASHES leaving a CRASH CODE on the terminal. Being crashed means that the CPU stops. The crash code contains information about the reason that caused the crash.

UTILITIES

Most of the routines mentioned above are used quite frequently and therefore exist all the time in the primary memory.
A lot of routine work is however to be done much less frequently like: formatting discs, edit files, copy files, compile and interpret programs. Programs which take care of this, called UTILITY PROGRAMS, can be provided and are sometimes considered as a part of the OS.
These programs are normally held in an external memory and are only called when needed.

THE SMALL OS

Small microcomputers are usually configurated with the things described above. It has sometimes been debated if this really is an operating system but many manufacturers refer to systems like this as one.
Some examples of operating systems which can be looked upon as "subroutine packages" are CP/M, MS-DOS, DOS 6, etc.

****
THE MULTITASKING OPERATING SYSTEM
****

This chapter will explain multitasking and the demands it put on an operating system.

MULTITASKING

From the computer's point of view man is hopelessly slow. When the computer has finished something to, for instance, ask a question, it takes several seconds, perhaps minutes for the operator to answer it. During this time the computer could have done something useful.
The logical thing would be to have another program running during this time. The technique of having several programs running "at the same time" is called MULTITASKING (multiprogramming) and is fundamental in the computer world. With multitasking we can take full advantage of the CPU's and the peripheral's time.

PROBLEMS RELATED TO MULTITASKING

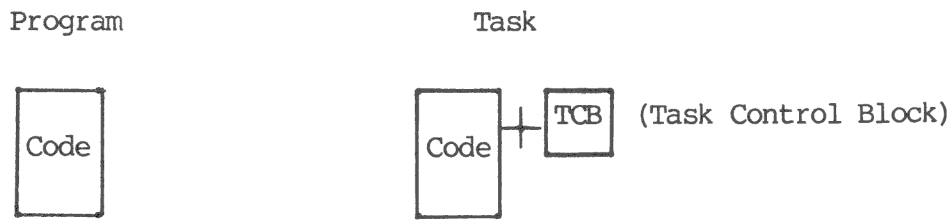
There are, however, several problems related to multitasking like:

- Which program should be running on the CPU?
- Two programs can demand access to the same peripheral simultaneously.
- Where in the memory should each program be placed?
- Etc.

Without strict rules we would have instant anarchy in the computer.

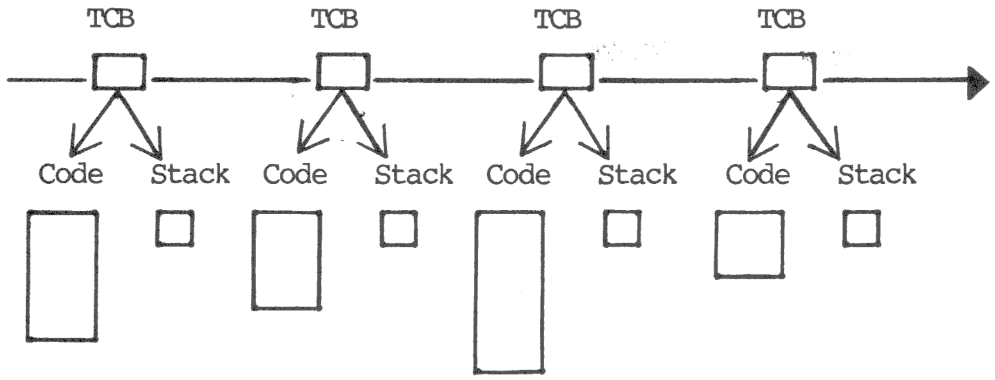
TASK - A DEFINITION

A program which runs in a multitasking environment needs additional control information so that the operating system can manage it. A program complete with such information is called a TASK (process).
Part of this information exists on the task file, and part of it is created by the OS (as a reference) when the task is loaded into the primary memory.
This reference point is called a TASK CONTROL BLOCK (TCB) and includes the size of the task, where the task resides in the primary memory, the status of the task etc.



Pic 4.1 The differance between a program and a task in the primary memory.

In order to have easy administration of the tasks, the Task control blocks are linked into lists and queues.
Every task has its own stack which is selected when the task is executing instructions.



Pic 4.2 The TCBs point at the code and the stacks of the tasks.

TASK STATES

Only one task can execute instructions at any given time. This means that all tasks are not active all the time. We refer to tasks as being in different STATES. The states defined in OS.8MT are:

- CURRENT STATE. This is the state of the task currently executing instructions. Only one task may be in current state at any given time.

- READY STATE. The task is aspiring to be the current task.

- WAIT STATE. The task waits for something to happen, an event, before it may reach ready state and current state.

- PAUSE STATE. The task has been paused by the operator or another task (even itself).

- DORMANT STATE. The task has not been started.

At any given time a task is in one of these five states.

RESIDENT/NON RESIDENT, ABOARTABLE/NON ABORTABLE TASKS

Tasks are also referred to as being:

- RESIDENT, it remains in memory when it has terminated (come to an end).
- NON RESIDENT, it is thrown out of memory when terminated. The memory space previously occupied by the task can be used again.
- ABORTABLE. The task may be canceled by another task
- NON ABORTABLE. The task can not be canceled by another task.

These states can be changed by other tasks or the operator.

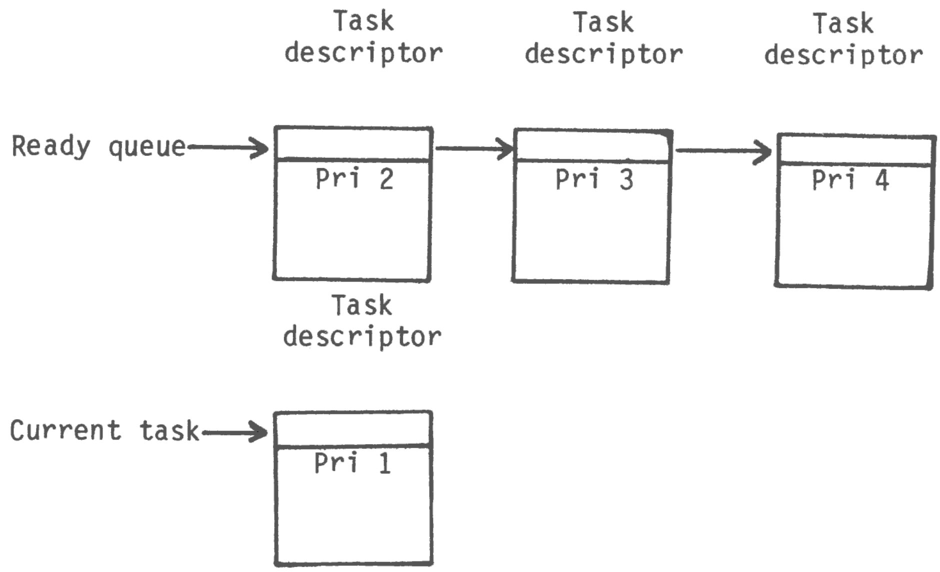
TASK PRIORITY

All tasks are not equally important. The tasks can therefore be associated with different PRIORITIES. A task with higher priority can "take over" the CPU from a task with lower priority, but not the opposite.

THE READY QUEUE

Only one task at a time may execute instructions on the CPU. Therefore there exists only one current task at any given time. All the ready tasks are kept in a READY QUEUE which is ordered in priority fashion, the task having the highest priority first. The current task will continue to execute instructions until:

- The task has nothing more to do and terminates.
- The task is paused by the operator or another task.
- The task is put into wait state for some reason.
- Another task with higher priority becomes ready.
- The task suspends itself, i.e. puts itself in the ready queue after the other tasks.

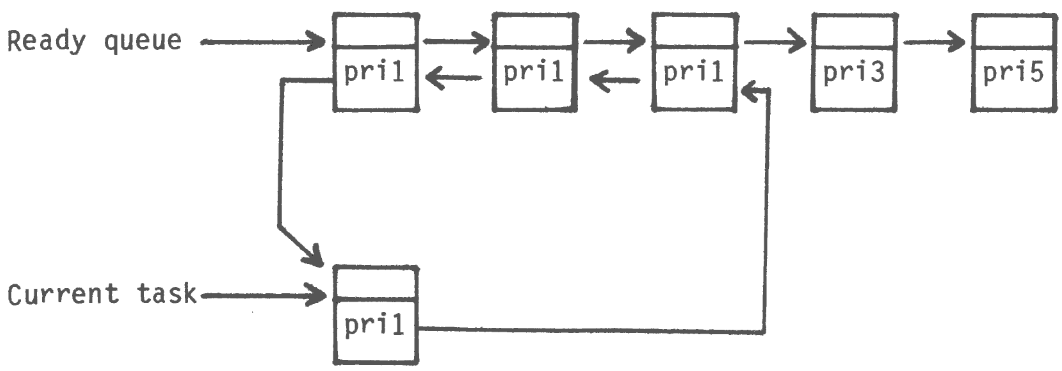


Pic 4.3 The current task and the ready tasks.

While this way of running tasks "one after another" sometimes is sufficient it would be nice if several tasks could run seemingly "at the same time".
This is solved with a technique called TIME SHARING.

TIME SHARING

Time sharing means that every task on the same priority level gets a slice of time, the time it may be the current task. When this time has come to an end, the task is put in the ready queue again, behind the other ready tasks on the same priority level. The first task in the ready queue is then picked to be the current task. This way of queuing is called ROUND ROBIN. The time slice is normally 0.1-1 second.
If a task of a higher priority level becomes ready it becomes the current task and interrupts the time-shared tasks on a lower level.



Pic 4.4 Time sharing.

Some computer systems which offer multitasking do not support time-sharing which makes them less usable for many applications.

THE ACCESS OF PERIPHERALS

As more than one task can demand access to a peripheral simultaneously this access has to be controlled by the operating system in some way.
The first thing to do is to define a RESOURCE.

RESOURCE - A DEFINITION

A RESOURCE is anything offering something to a task. It can be:

- A DEVICE like a disk drive, printer, terminal, I/O board, etc.
- Another TASK.
- A VOLUME, like a floppy diskette, disc-pack, etc.
- An AREA in the memory.

You can make a distinction between:

- A SHARABLE (reentrant) resource, which can be used by many tasks at the same time.
- An EXCLUSIVE (sequential) resource, which only one task at a time can use.

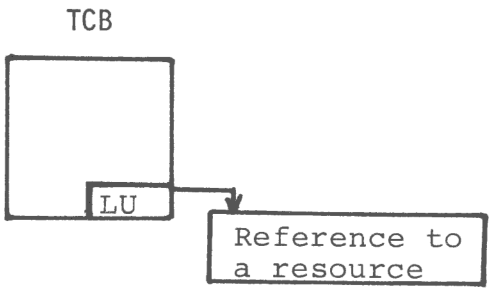
When a piece of code is a sharable resource, you talk about reentrant code. This will be explained later.

ADMINISTRATION OF THE RESOURCES

The OS keeps information about all resources present in the system. This information is, like the control information for the tasks, held in tables which hold information of the type of the resource, if a task is using it, if it is sharable or exclusive etc.

LOGICAL UNIT

Before a task may demand access to a resource it must make a reference to it. This is called to establish a LOGICAL UNIT. The logical unit has the form of a number which is given at each request of a resource.
The assignment can be pictured as a bridge between the task and the resource. If the bridge is closed no traffic is possible. The command OPEN in BASIC performs this function.



Pic 4.5 Logical unit. All referances from tasks to resources have different numbers.

SUPERVISOR CALLS

The tasks may not themselves have access to the resources, they have to request the operating system to access the resource for them by making standardised requests.
Such a request is called a SUPERVISOR CALL (SVC). A SVC is the only way a task can request a resource, and service by the OS.

THE SUPERVISOR CALLS OF OS.8MT

A Supervisor Call (SVC) consists of:

- The characters: "SVC".
- A group number.(1-8)
- A parameter block

There exist 8 different SVC types in OS.8MT. Each SVC type can in turn perform a great number of different functions. The function is given in the parameter block of the SVC. You also have to specify how the function should be made.

This listing shows some examples of the most used SVC functions in OS.8MT. For a complete listing of the available SVC functions see the OS.8MT PM.

FUNCTION	OPTIONS	PARAMETERS	SVC
The assignment of a task to a resource. (Compare OPEN in BASIC)	The assignment of - A Device. - A File. - A Task.	The LU number the resource will be asso- ciated with.	SVC 7
An I/O call to an assig- ned resource (Compare INPUT and PRINT in BASIC)	- Read call - Write call	ASCII or Binary data. Access mode. The LU number.	SVC 1
The allocation of a file. (Compare PREPARE in BASIC)		File name. Access mode.	SVC 7
The opening of a device. (Puting it on- line).This is NOT the same as OPEN in BASIC!	The opening of the device: - Write protected - Non-file structured	The name of the device.	SVC 2.12
The closing of a device. (Putting it off-line).		The name of the device	SVC 2.12
An I/O call to an assigned resource	Read Write	Data Format Access mode	SVC 1
The starting of one task from another		Name of the task	SVC 6
The allocation of a file.	File type	Name of the file	SVC 7

MODES

When the computer executes different types of code and data it is referred to as going through different MODES:

- USER MODE (problem mode, slave mode), which is the mode the system is in when task code is executed.

- SUPERVISOR MODE (master mode), when the OS executes code.

The reason for having different modes is that certain instructions are reserved for use by the OS. The OS knows which mode the system presently is in and can prevent illegal instructions from the tasks.

PRIORITY LEVELS OF THE PERIPHERALS

There exist two ways of entering the OS:

- A task issues an SVC.
- An interrupt is detected.

```
            SVC                 Interrupt
Task ------------>  OS  <------------ Device
```

As mentioned earlier peripherals issue interrupts when they need attention. The physical devices have different priority levels depending on how important they are.
For example it is more important that the real time clock gets serviced than the printer. Many important things may happen because of a clock interrupt, while the printer can halt a short time with no bad effects. Thus the clock has a higher level than the printer.

PRIORITY LEVELS DURING THE WORK OF THE OS.

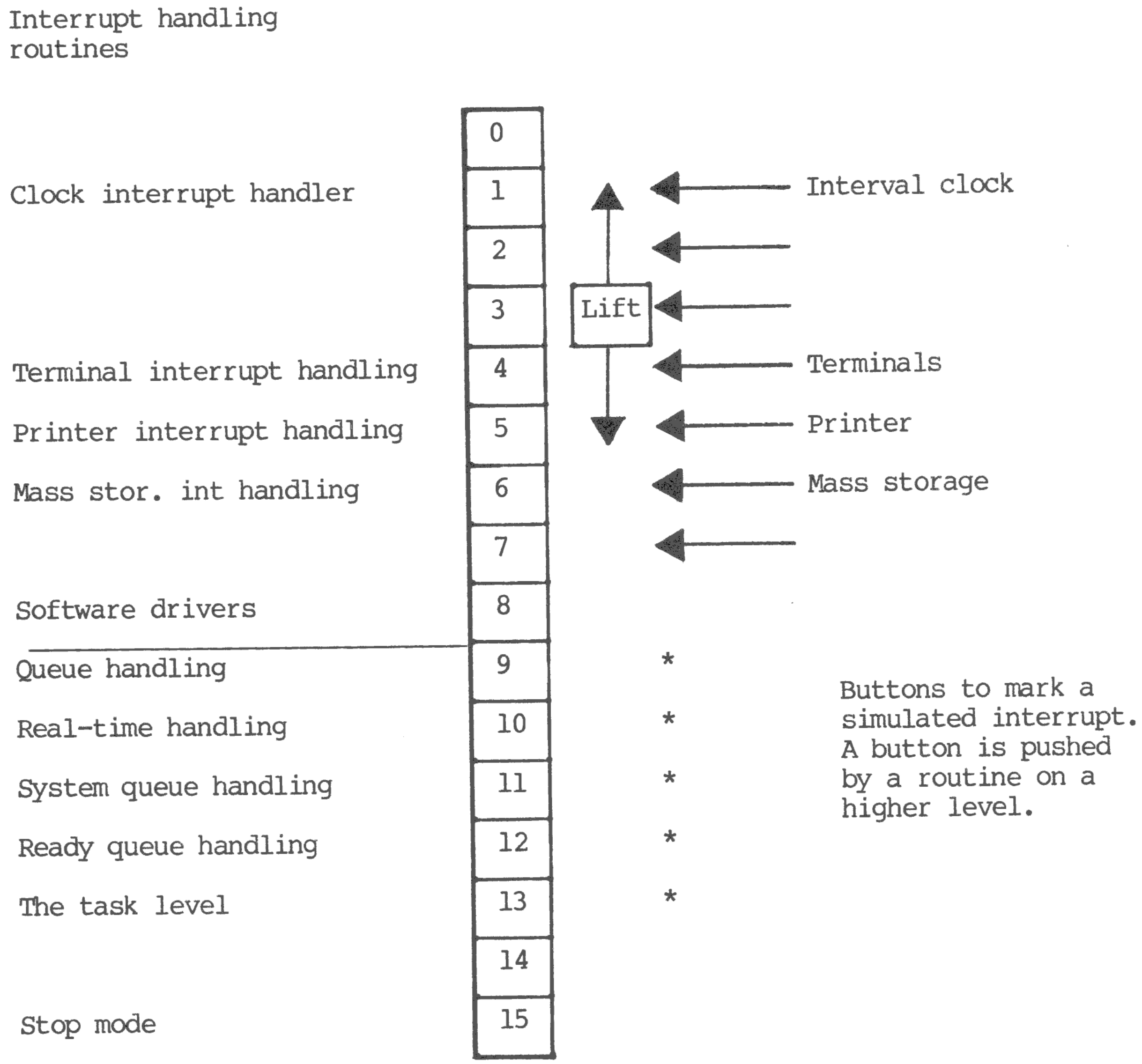
OS.8MT uses interrupts not only for the peripherals but for several purposes. There are a for instance a number of routines hich should not be interrupted by other routines and has therefore a higher priority.

We will try to make a visual model OS.8MT's structure. Note that it is not necessary to learn how the OS works in detail, but it adds to the understanding.

As the OS executes on different levels, a close analogy is a "house" with 16 levels. The only way to reach the different levels is by the system interrupt handler.

THE SYSTEM INTERRUPT HANDLER

The SYSTEM INTERRUPT HANDLER which can be looked upon as a "lift" which travels up and down according to the rules given in pic 4.6.



Pic x.x A symbolic picture of the interrupt system in OS.8MT. The system interrupt handler is here in the shape of a "lift".

- The "lift" travel upwards if a hardware interrupt occures on a level HIGHER than the present level. The lift always travels to the HIGHEST interrupted level.

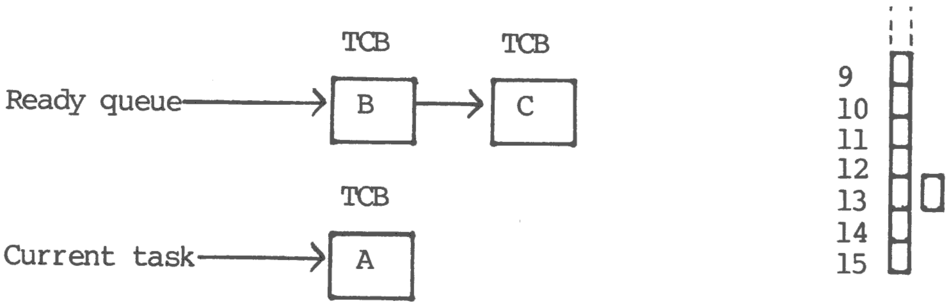
- The "lift travels downward if all work has been done at the present level.

- The "lift" halts at a level if a routine on a higher level has marked a software interrupt on that level. I.e. "pushed the button" on that level

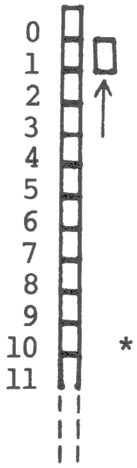
AN EXAMPLE OF THE WORK OF THE OS

We will by a simple example show how the OS works.

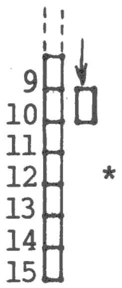
1. The computer executes task A one of three time shared tasks. The priority level is 13 - task level.



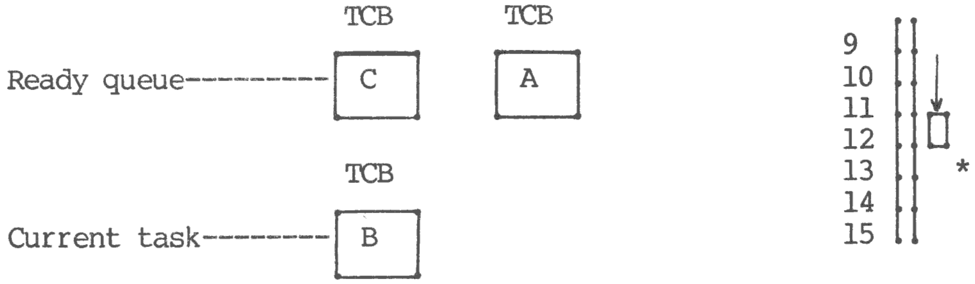
2. The interval clock issues an interrupt on level 1. The "lift" travels up to level 1 and the interrupt handler updates the real-time clock in the OS. A certain time interval has passed and the interrupt handler "pushes the button" at level 10.



3. When the "lift" travels downward it halts at level 10 (the "button is pushed). The real time handler finds out the reason for the interrupt. A new task is about to execute instructions. The real-time handler "pushes the button" on level 12.



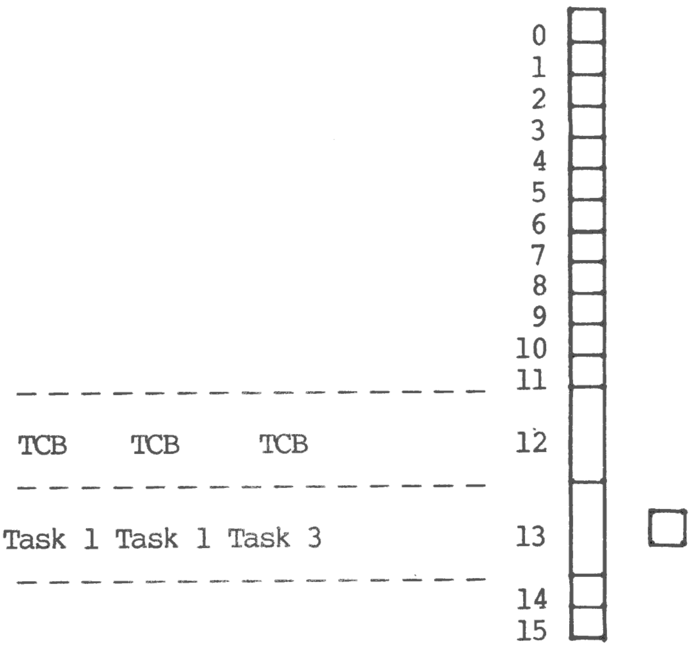
4. The "lift" halts at level 12 where the ready queue handler is entered. It changes task B to be the current task and puts task A at the end of the ready queue.



5. The task level is reached and task B executes instructions.

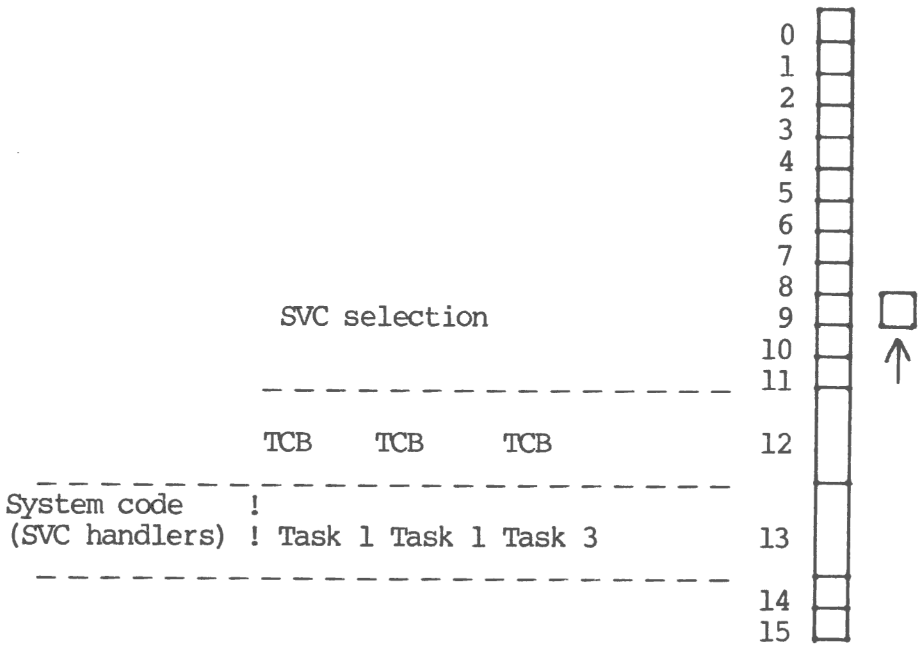
SVC HANDLING

We said earlier that the only way to reach the OS is through an interrupt or a task issuing an SVC. Actually, when an SVC request is made, an interrupt is simulated on level 9.



Pic 4.7 Three tasks are in the computer. Note the code which is executed at level 13 - task level, while the control information (TCBs) only is reached on level 12 - ready queue handling.

If a task issues a SVC an interrupt is simulated on level 9 where the SVC handler is selected. The SVC handler checks the SVC call and choses an appropriate SVC code which is executed on task level

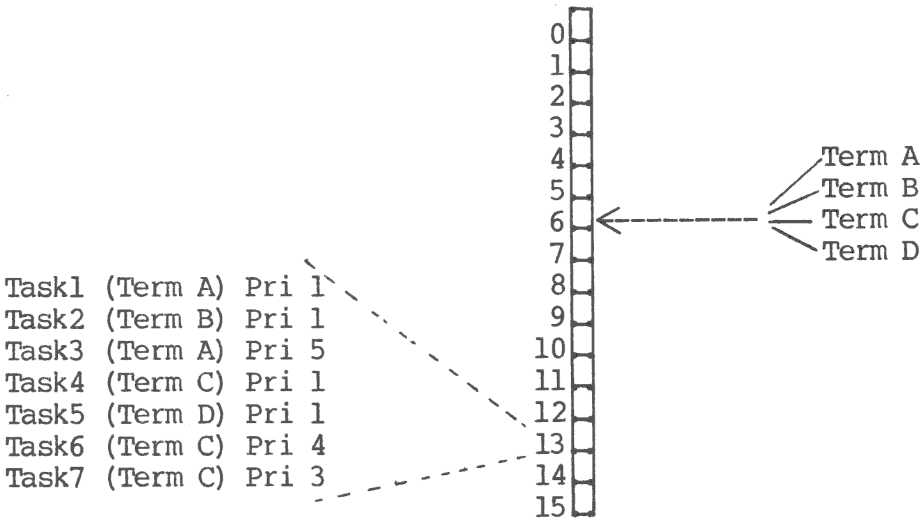


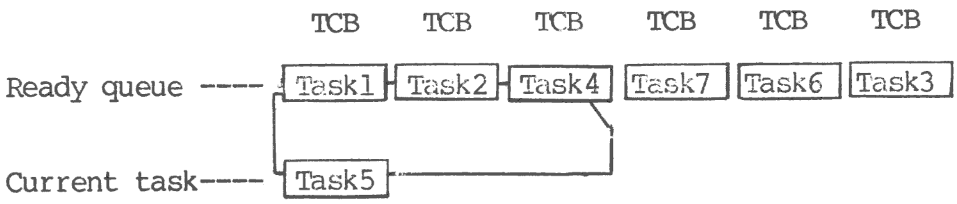
Pic 4.8 The appropriate SVC handler is selected on level 9. The handler is executed when the level is allowed to drop to 13.

The system code is executed like a subroutine to the calling task

THE MONITOR IN MULTITASKING OS

In a multitasking computer the monitor is not only responsible for interpreting and executing commands. It also has to direct terminal I/O to the correct terminal (if more than one terminal is used).



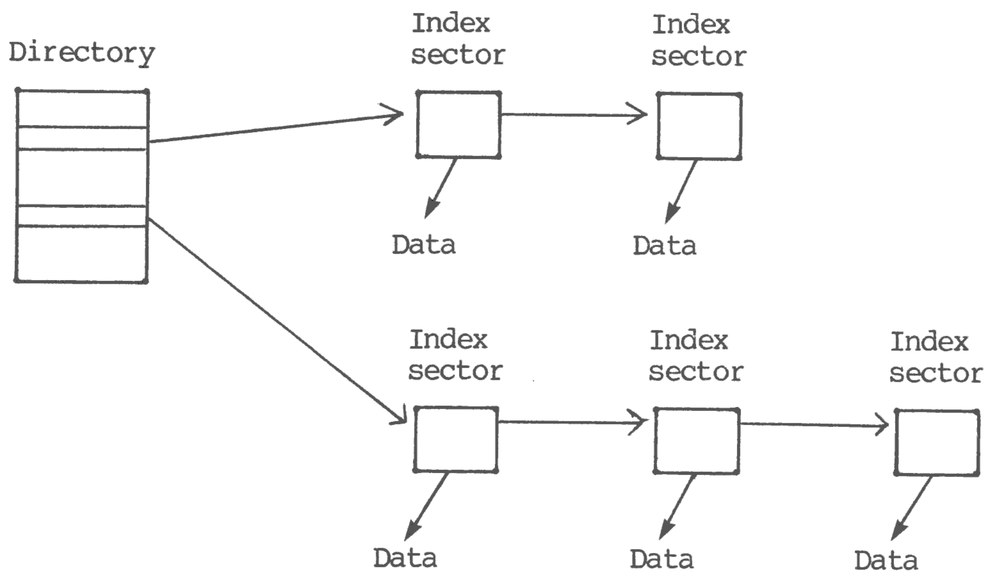


Pic 4.9 Four terminals (Term A-D) have started the execution of seven tasks (Task 1-7). Task 1, 2, 4, 5 are time shared on equal priority.

THE FILE HANDLING IN A MULTITASKING COMPUTER

The main differance between the single user- and the mulititasking system is the protection needed in the latter. This is acomplished in OS.8MT by regarding files as exclusive resources when it comes to the reading of them. Each volume and file has added control information similar to tasks and other resources.
Furthermore the File management system in OS.8MT offers a number of other important features not found in most single user systems.

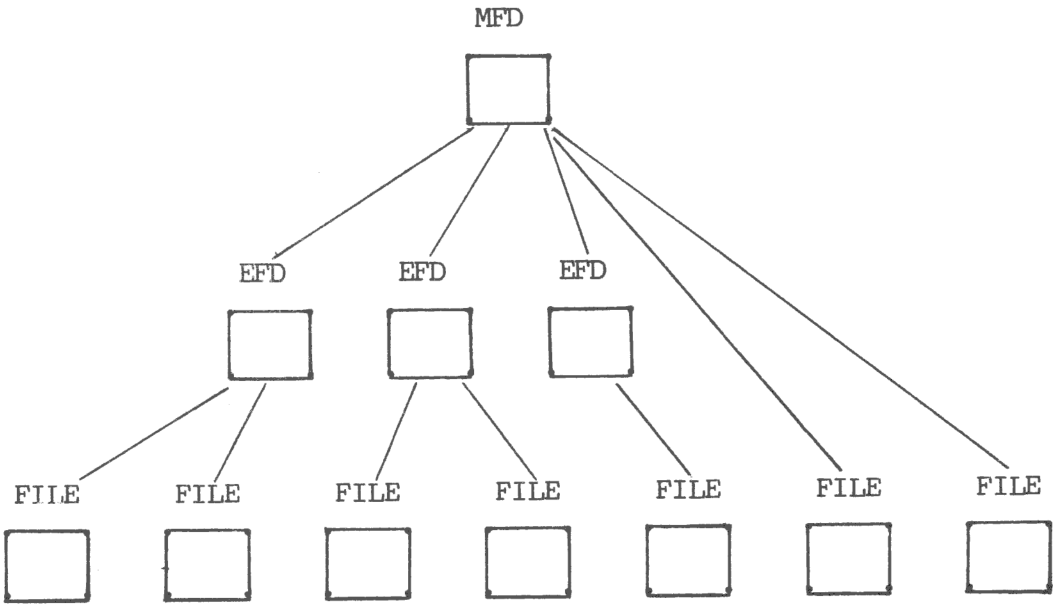
THE LOGICAL LAYOUT OF A VOLUME



Pic 4.10 A (simplified) logical layout on a volume.

Data is reached by using a directory and index sectors. This makes it possible to add and return space to a file in a dynamic way.
It is also possible to have non-indexed files which are called contignous files.
The content of a volume is displayed by giving the command LIBRARY volume name. LIBRARY ABCD: gives for example the content of the volume ABCD.

ELEMENT FILE DIRECTORIES

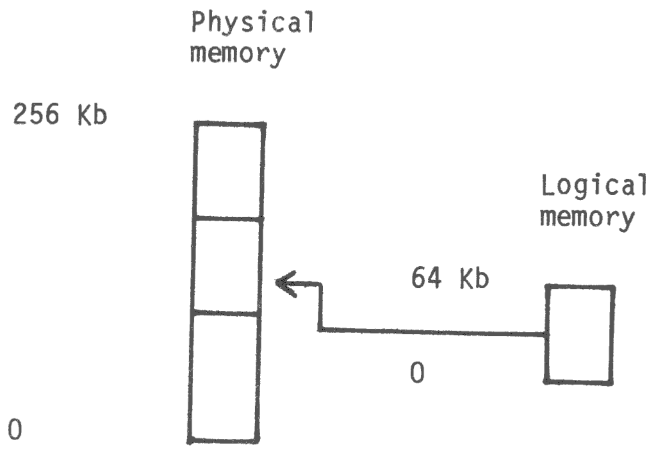


Pic 4.11 Element file directories (EFD) are on a level below the Master file directory (MFD).

By using element file directories, element files can be used. This has many advantages as for example every user can have his own directory and files of a certain kinds can be kept reached from the same EFD.

MEMORY MANAGEMENT

The addressing range of a CPU is limited. If the CPU has 16 address lines like the Z80, it can address 64 Kbytes. This is called the LOGICAL adress range.
If we want to expand the memory, we need a hardware device called MEMORY ACCESS CONTROLLER (MAC). By giving the MAC instructions the OS can determine what part of the PHYSICAL memory the logical memory can "see".



Pic 4.12 The physical and logical memory.

THE PRIMARY MEMORY OF A MULTITASKING COMPUTER

In a multitasking computer the primary memory is divided into two parts:

- A PURE segment which includes instructions, but no data.
- An IMPURE segment, which may include both code and data.

The reason for this is that if two or more tasks are using the same piece of code they must have separate data areas. Otherwise the data of the tasks would get mixed up. Code that, in this way, can be used by many tasks is called REENTRANT code. Reentrant code must be placed in the memory's pure segment. An example of a task written in reenterant code is the BASIC interpreter which only exists in one copy no matter the number of tasks using it. Every task has a separate data area though.

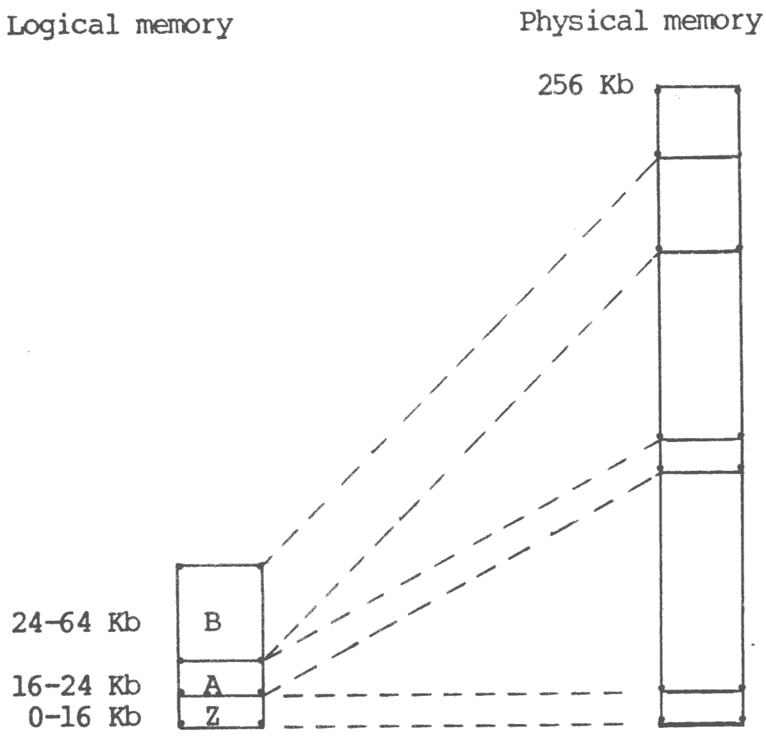
THE A, B AND Z SEGMENTS

The logigal memory of OS.8MT is divided into three parts:

A Z-segment which always "see" the bottom 16 Kb of the physical memory.

A pure code A-segment of 8 Kbyte which can be moved to "see" different parts of the physicl memory.

An impure code B-segment of 40 Kbyte which also can be moved to "see" different parts of the primary memory.

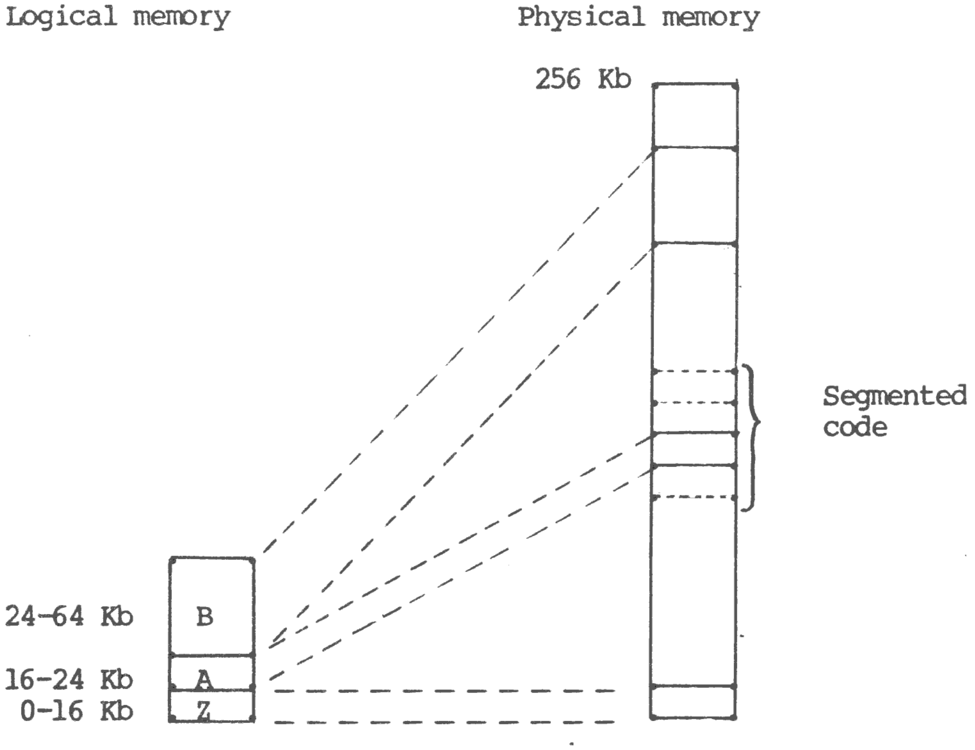


Pic 4.13 An example of the operation of the MAC. The A and B segments in the physical memory are moved to the location of the executing task. The Task Control Blocks hold information about where each task is placed.

Several tasks can use reenrant code in the same A segment while they have different B segments.

SEGMENTED TASKS

Large tasks can be segmented using several A segments in the physical memory. The different segments are tuned-in when needed. The BASIC interpretator is an example of a task using this technique



Pic 4.14 An example of a segmented task. The physical segments are tuned in when needed.

LOADING TASKS IN THE PRIMARY MEMORY

When a task is loaded into the primary memory from an external memory, it must be placed in a free memory part.
This is done by a LOADER. The loader uses the information supplied by the task to determine the space needed for it and whether it should be put in the pure or impure part of the memory. The OS contains information about the free parts of the primary memory.
During the time of execution, many tasks can be loaded and cancelled. (A cancelled non-resident task no longer exists in the primary memory.) The cancelled tasks leave "holes" where new tasks can be loaded.

DRIVER ROUTINES

The peripherals are controlled by means of special device dependant instructions. When collected into a separate routine including all the necessary instructions for the control of a peripheral you talk about a DRIVER ROUTINE.
By having driver routines you gain device independency of the programs as you can change the driver routine instead of all the programs using the peripheral.
A device driver may be loaded into the primary memory with the system on-line, just like a task.

SUMMARY

Multitasking means that more than one program is using the computer, running concurrently "at the same time". A task is a program with additional control information so that the OS can manage it in a multitasking environment.
All things in the computer which can be used by the tasks are called resources. The resources can either be sharable or exclusive.
The tasks gain access to the resources by issuing Supervisor Calls (SVC), which is the only way to enter the OS except for an interrupt. The interrupts can have different levels.
The primary memory is controlled by means of a Memory Access Controller (MAC) which permits the logical address range to be expanded.

****
THE SOFTWARE MODULES OF OS.8MT
****

The flexiblility of the DataBoard hardware demands an equally flexible operating system. The solution has been to group OS.8MT into modules. Only the essential modules need to be included at system generation time. This minimizes the space the OS occupies in the primary memory. The main parts are:
The Kernel, the File manager, the device drivers, the Monitor and the Utilities.

THE KERNEL

The kernel includes the essential parts of the system like:

- The READY QUEUE HANDLER which keeps order in the ready queue and determiens which task is to execute instructions.

- The INTERRUPT HANDLER which delegates the work to be done as the result of an interrupt.
  The interrupts from the interval clock are handled by a special CLOCK INTERRUPT HANDLER in order to minimize the overhead.

- The SVC HANDLER and SVC FUNCTIONS, managing task requests for system resources.

- Handlers which manage the control of resources. Included here is the CONNECTION HANDLER administrating the connection and queueing of a task request to a resource. The reverse function is made by the DISCONNECTION HANDLER which releases a task request from a resource aided by the SYSTEM QUEUE HANDLER:

- The REAL TIME HANDLER which manages the system clock and calender. Task requests made through Supervisor Calls are also handled as well as when a device has reached a time-out, a condition described in chapter 7.

- The CRASH HANDLER which is called when OS.8MT determiens that there is a risk of data being damaged.

- The MEMORY MANAGER which administrates the primary memory of the computer, that is: the free parts of the memory, where tasks are located etc.

Although the kernel is necessary in every configuration of OS.8MT many system tuning constants can be changed in order to optimize the performance of the system for different applications.

The kernel needs about 8 Kbytes of the primary memory.

THE FILE MANAGER

The File Manager consists of routines which manage files on mass storage devices. Included here are a directory manager which can change the information held in the directories on the mass storage units and a bit-map manager which provides a method for allocating and deleting space on files.
The file manager needs about 8 Kbytes of primary memory.

DEVICE DRIVER ROUTINES

These routines are able to control the physical devices present with the system. You can either chose from a library of pre-prepared drivers for different devices or write your own drivers. By a matter of choice the device driver routines can be resident in the system or loaded from mass storage when needed. By loading device drivers on-line you can minimize the amount of needed memory space.
By keeping the device dependent instruction in driver routines you gain device independency of thè tasks.

TERMINAL MANAGER (MONITOR)

This task is responsible for interpreting and executing commands. It peforms all I/O requests to the terminals.
When OS.8MT is used with more than one terminal, the terminal manager directs terminal I/O to the correct terminal.
The terminal manager needs about 8 Kbytes of the primary memory.

UTILITIES

The utilities perform things like:

- Initializing and formatting disks.
- Copying from one disk or device to another.
- Organising the OS.8MT data base system (ISAM).
- Controlling the state of a disk.

The utilities are just like user tasks and must use SVCs to gain access to the facilities of the OS. While the utilities normally reside on mass storage they are loaded into the primary memory when needed.

SYSTEM GENERATION

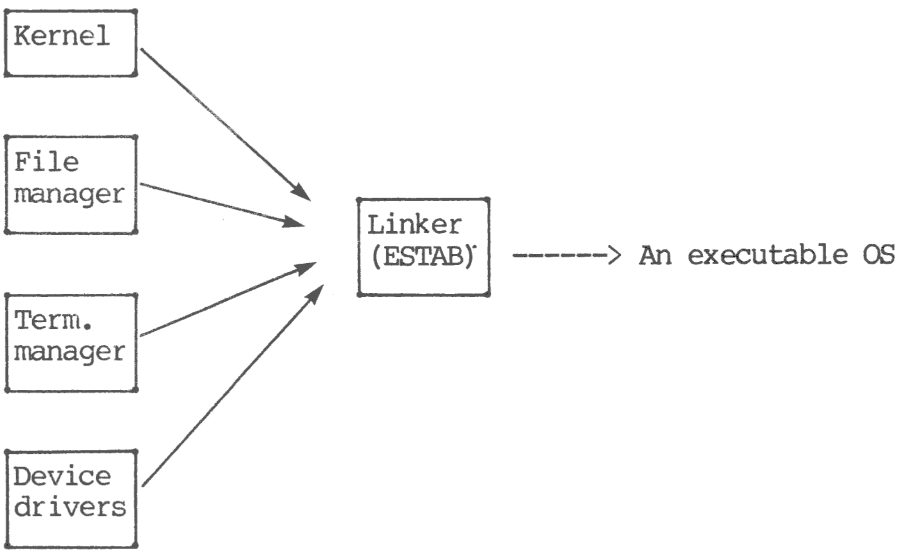
OS.8MT is generated in the following way:

- Edit the selectfile which the linker uses to know which modules to include.

- Omitt the modules you don't need and change the tuning constants according to your needs.

- Initiate the linking process.

An example of a selectfile and a linking process is found in appendix D.



Pic 5.1 OS.8MT is generated by linking the appropriate modules.

SUMMARY

The main modules of OS.8MT are:

- The kernel

- The file manager

- The monitor

- Device drivers

The modules are linked together to form a complete operating system. Several system constants can also be manipulated in order to optimize the OS for a certain application.

****
INFORMATION STRUCTURES
****

THE NEED OF CONTROL INFORMATION

As mentioned, it is very important to have a strict order in the OS. Every task and resource must be accompanied by control information so that the OS can manage them.
As tasks and resources can be created, loaded, and cancelled while the system is running, the control information must be dynamic, reflecting the state of the tasks and resources. A suitable method is to have the information in tables, which are linked together into a list structure.

SYSTEM POINTER TABLE

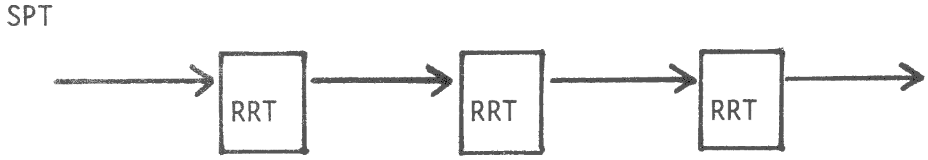
All information has to be reached from somewhere, we need a static reference point in the system. This reference point in OS.8MT is called the SYSTEM POINTER TABLE (SPT).
The SPT is located in the bottom 16 kb of the memory. It consists not only of reference roots to list structures and queues, but also of system constants and interrupt vectors.

THE RESOURCE QUEUES

There are three classes of resources in OS.8MT, tasks, devices and volumes. It may seem confusing that all tasks are also resources, but it is necessary as one task can request the service of another task.
In the SPT there are roots to three list structures, each describing one type of resource. The three list structures are similar in many ways.

THE RESOURCE REFERENCE TABLES

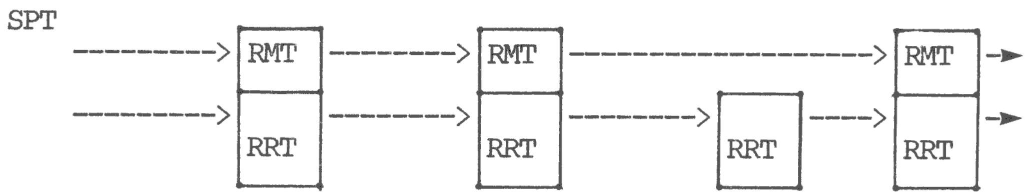
Every resource has a unique NUMBER. It can be reached simply by giving that number. Every resource must therefore be described in a numeric way. This is done by the RESOURCE REFERENCE TABLE (RRT).
In addition to the resource number it contains information about the TYPE of the resource i.e. if it is sharable or exclusive. (A sharable resource may be used by more than one task at a time.)
All RRTs for each resource class are linked together. A pointer from the SPT points at the head of the queue.



Pic 6.1 Resource referance tables

THE RESOURCE MNEMONIC TABLES

If we want to reach a resource by giving a symbolic name, we need another table called RESOURCE MNEMONIC TABLE (RMT). The RMT consists only of a name of four characters and a pointer to the corresponding RRT.
Like the RRTs, the RMTs are linked together.



Pic 6.2 Resource Mnemonic- and resource referance tables. Note that some resources only can be reached by the number and therefore misses the RMT.

THE DIFFERENCE BETWEEN SHARABLE AND EXCLUSIVE RESOURCES

If the resource is SHARABLE we find a pointer in the RRT to either the entry address to the code.
If the resource is EXCLUSIVE, we must have a way of controlling the access to the resource. We have to introduce a "bouncer" (see below) which only permits one request at a time to have access to the resource. The other requests are queued while waiting for their turn.

THE RESOURCE CONTROL BLOCKS

This "bouncer" takes the form of a block called the RESOURCE CONTROL BLOCK (RCB). The pointer which pointed at the code in the sharable resource is pointed at the RCB in exclusive resources.
The RCB contains information about:

- The type of resource the RCB controls (Task, Device, Volume, File etc.
- What kind of calls the resource supports.

- The status of the resource. (Active, off-line, etc.)

- If the resource is free to use, or not.

It also contains pointers to the request which currently is using the resource and a pointer to a request queue. A pointer to an optional parent also exists (more about this later).

THE TASK/VOLUME/DEVICE CONTROL BLOCKS

So far the three resource queues have been very similar but now we need more specific information about each resource. This information is found in the TASK CONTROL BLOCK, DEVICE CONTROL BLOCK and the VOLUME CONTROL BLOCK which all are extensions of the Resource Control Blocks (RCB).

THE TASK CONTROL BLOCKS

In the introduction we mentioned that what makes a task different from a program is the control information added to the task. The TCB holds this information which include:

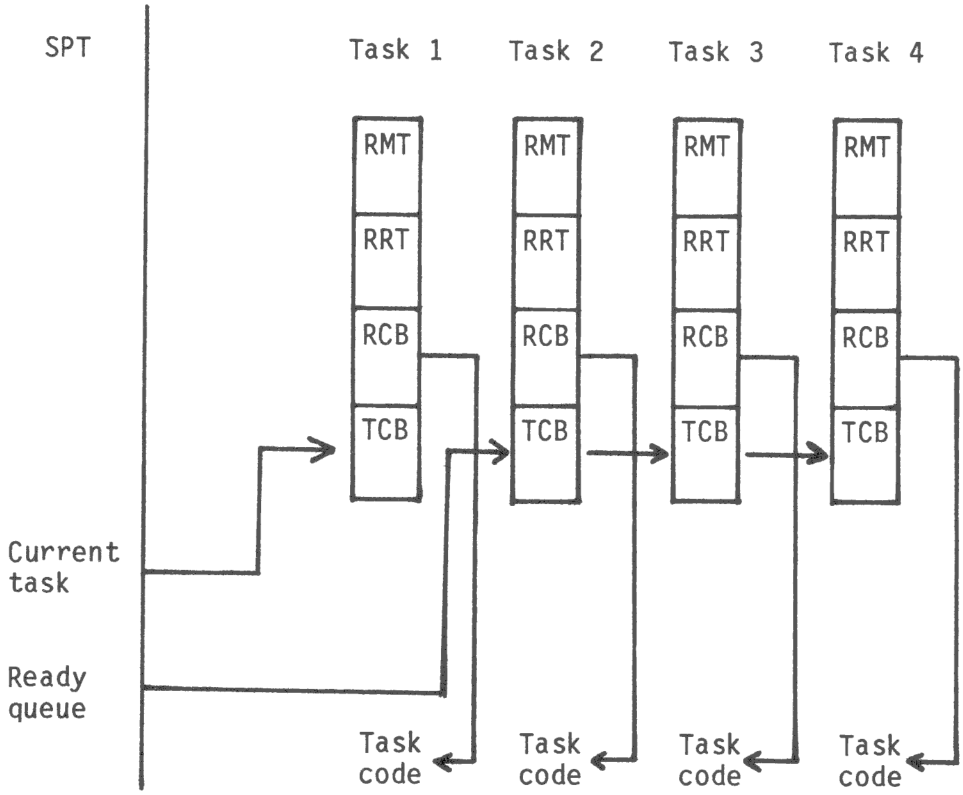
- Where in memory the task segments are placed.

- Address to the tasks stack.

- The task's priority.

- The task's TYPE (resident/non-resident, abortable/non-abortable etc).

- The task's status (current, ready, waiting, paused etc).



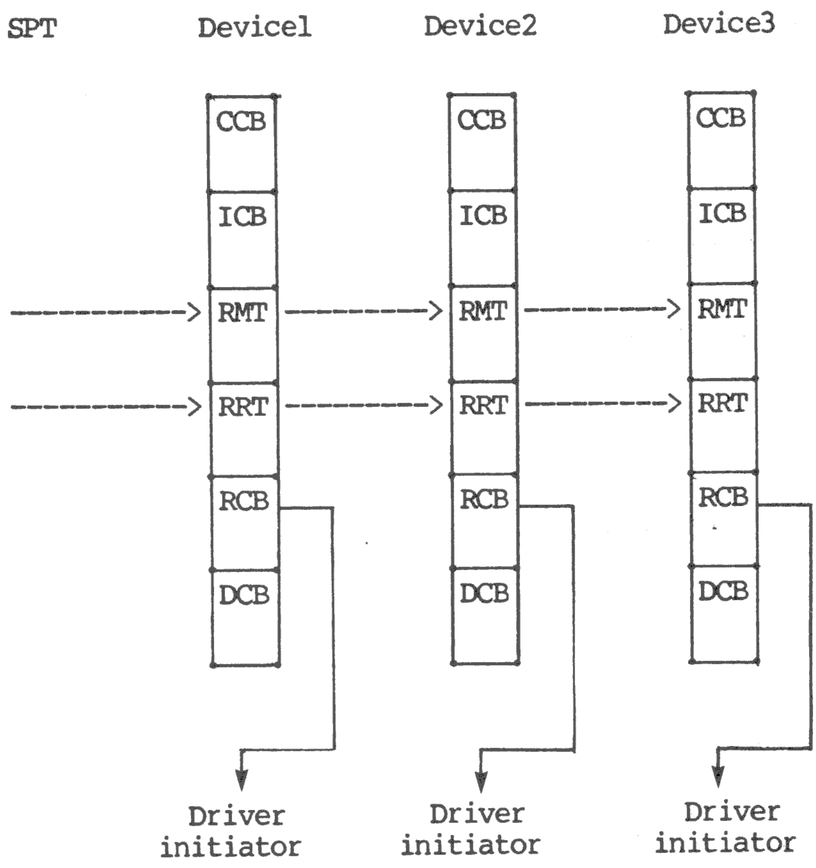
Pic 6.3 The complete table structure for the administration of tasks. Task 1 is the current task while task 2 and 4 is present in the ready queue.

DEVICE CONTROL BLOCK

Different devices have different characteristics like:

- If the resource supports read and write.
- The dataformats which the device supports.
- The positioning the device is capable of doing (forward record, forward file, rewind etc)
- The type of the device (dedicated device, task device etc)
- Where buffers are located.

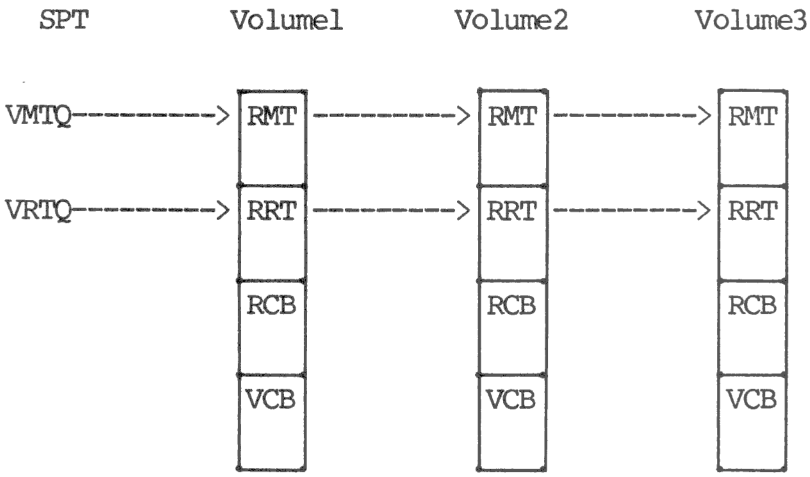
Information like this is found in the DCB. A task requests a resource by issuing a supervisor call. The SVC includes a parameter block which contains information about that special request (how much and what kind of data to be transferred, read or write etc). The parameter block is being copied to the DCB during the initial phase, before the data transfer takes place. The driver initiator uses this information. The driver is also responsible to check if the device supports the functions requested in the SVC call.



Pic 6.4 The complete table structures for the administration of devices.

THE VOLUME CONTROL BLOCKS

For every volume which is opened to the system a VOLUME CONTROL BLOCK (VCB) is built. The VCB is mainly used as a work area for certain SVC calls when accessing directory structured devices.



Pic 6.5 The complete table structures for the administration of volumes.

THE FILE CONTROL BLOCKS

Every time a file is opened by a task, a FILE CONTROL BLOCK is created. It contains information necessary to handle the access of a file by a task. (See Logical Unit, Tasks and task handling).

INTERRUPT DRIVEN DEVICES

Interrupt driven devices also need an INTERRUPT CONTROL BLOCK (ICB) and, if it is a physical device, a CHANNEL CONTROL BLOCK (CCB). The ICB is being used, after an interrupt has occurred, by the interrupt handler to search for the device which is responsibie for the interrupt. It also contains a pointer to the interrupt handling routine (in this case the driver routine).
The CCB contains the card select address of the device and a test mask for the decoding of the status recieved from the device. The ICB and CCBs function will be described more thoroughly in the next chapter.

SUMMARY

The information about all resources is kept in blocks which are linked into list structures. Included among the blocks were:

- The Resource Reference Table (RRT), holding information about the number of the resource.
- The Resource Mnemonic Table (RMT), including the name of the resource.
- The Resource Control Blocks (RCB) controlling the access of exclucive resources.
- The Task Control Blocks (TCB) including information about a task in the primary memory.
- The Device Control Blocks (DCB) holding information about the characteristics of a device.
- The Volume Control Blocks (VCB) which holds information about the volumes present in the system.
- The File Control Blocks (FCB) which are used when working with files.
- The Interrupt Control Blocks (ICB) and the Channel Control Blocks (CCB) which are used to administrate interrupts.

All Tables can be reached from The System Pointer Table (SPT) which also holds all system constants and interrupt vectors.

SYSTEM LEVELS AND INTERRUPT HANDLING

Interrupts are very important in a multitasking computer. This chapter takes a very thorough look at the interrupt handling. Detail knowledge of it is only necessary if you plan to work with advanced program development but everybody using the system should have a basic knowledge.

PRIORITY LEVELS

Depending on the type of work the computer does, it goes through different SYSTEM LEVELS.
There are 16 levels, where the 13 highest are reserved for the OS. Level 13 is the task level, which is divided into 256 task priority levels. (The OS can also work at this level.) When no code is to be executed, the system enters level 15-the stop mode. All levels are reached as a result of an interrupt. The interrupts can be of two types:

- HARDWARE INTERRUPT, a device signals it needs attention.

- SIMULATED INTERRUPT, the OS SIMULATES an interrupt because there is work to be done on a certain level.

The highest 13 levels can be put into four groups:

- HARDWARE DRIVERS, level 0-7. On these levels we have the device driver routines. These levels are normally reached as a result of a hardware interrupt, but can under some circumstances be triggered by a software interrupt.

- SOFTWARE DRIVERS, level 8. Used for non-interrupt driven devices. A short routine is polling the devices.

- QUEUE HANDLING, level 9. This level is entered during the critical time when queues are being manipulated by the OS.

- SERVICE ROUTINES, level 10-12. These levels are only reached if a routine on a higher level has simulated an interrupt on one of them.

We need a place to keep all the information about interrupts. This place is called the interrupt service tables.

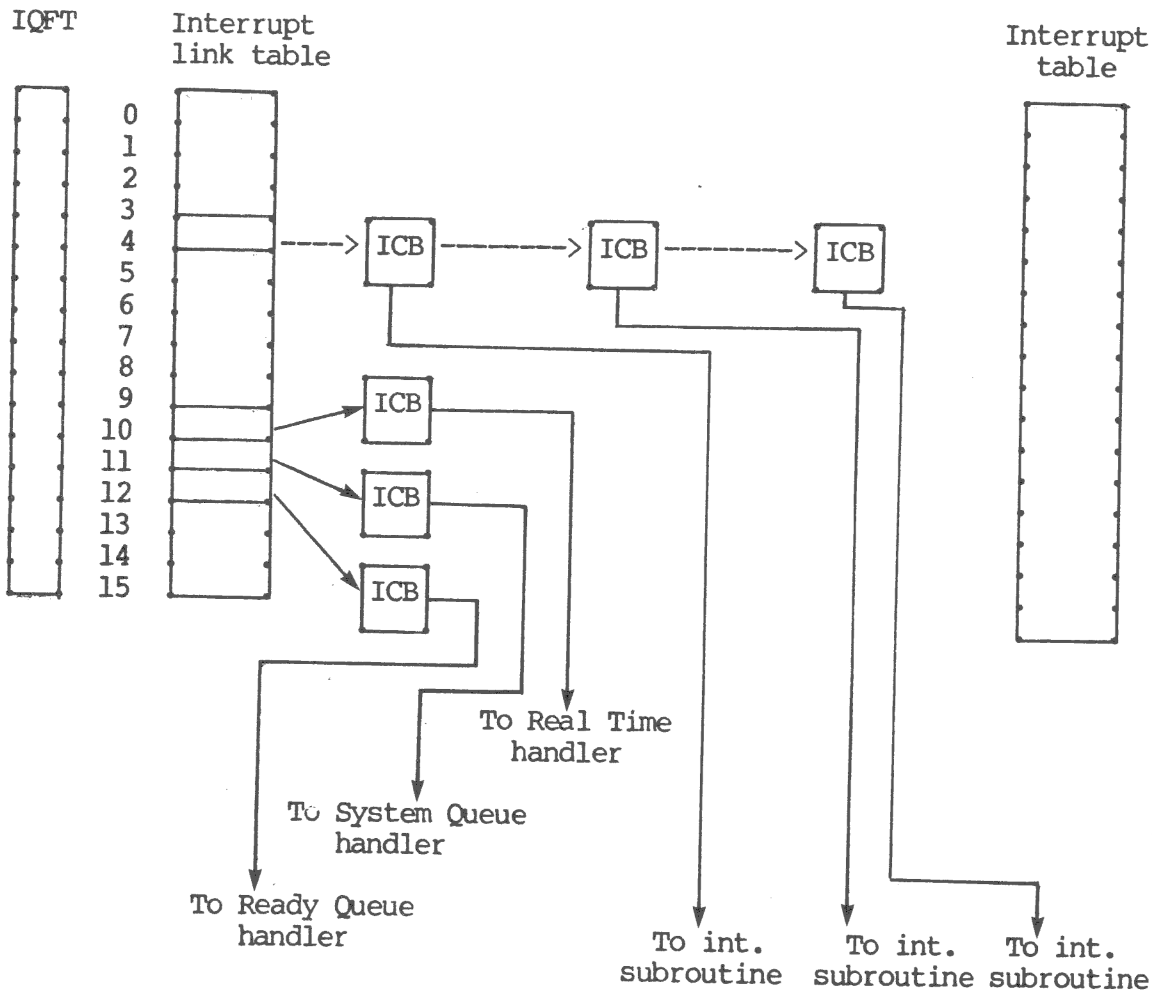
THE INTERRUPT SERVICE TABLES

The interrupt service tables are located in the system pointer table and consist of:

- One INTERRUPT QUEUE ADDRESS TABLE (IQUE), which holds the adresses to the interrupt control blocks of the devices which may give interrupts. If there exists more than one device on some level, the Interrupt control bocks are ordered as a queue on those levels. (For more info on ICBs see Information structures, Interrupt driven devices)

- One INTERRUPT TABLE, which is used by the OS to determine if the driver is DEDICATED or not.(Dedicated drivers will be covered later.)

- A byte vector (IQFT), which is used to mark a software interrupt on a level.



Pic 7.1 The interrupt service tables.

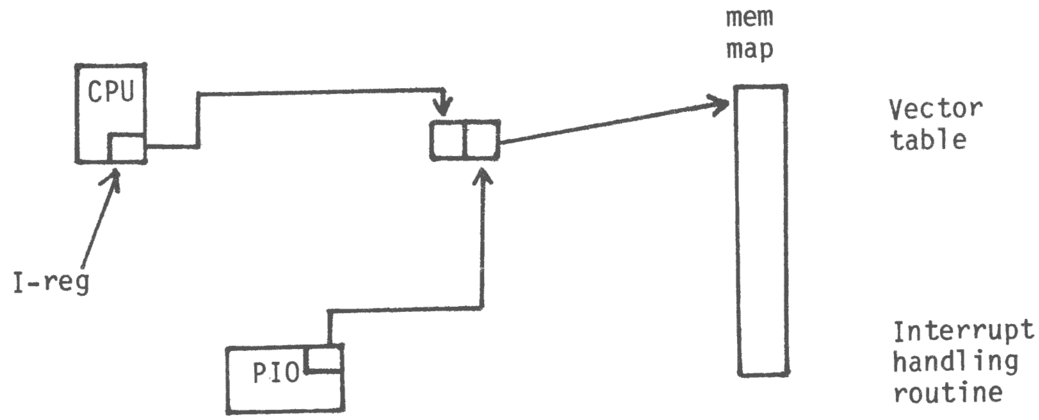
The interrupt service tables are handled by the SYSTEM INTERRUPT HANDLER (SIH), which is entered every time either a hardware or a software interrupt is issued.
Before we describe SIH in a detailed way we need, however, more information about the hardware interrupt's way from a device to the CPU.

THE HARDWARE INTERRUPT

The Z80 CPU has its own interrupt mechanism which is modified by the DATABOARD hardware and OS.8MT. The reason for this becomes clear if we take a look at one of the standard interrupt mechanisms of the Z80 and the devices in the Z80 series (PIO, SIO, etc).

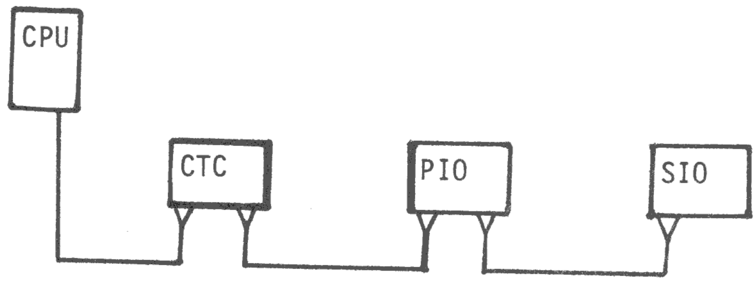
THE INTERRUPT MECHANISM OF THE Z80

The Z80 CPU handles a technique called VECTORISED INTERRUPT. A device gives an interrupt by lowering the signal level on its "INT" pin, which is wired to the "INT" pin of the CPU. The CPU acknowledges by giving a unique combination of signals.
The device responds to this by putting out an INTERRUPT VECTOR on the data bus. The CPU combines the interrupt vector with another vector stored in the CPU (in a I-register). The result gives a memory location where the address to the interrupt handling is held. A jump to the interrupt handling routine is then made.



Pic 7.2 The standard interrupt handling of the Z80 CPU.

If more than one device is used, the devices are wired into a DAISY CHAIN. The daisy chain is ordered in priority as shown in pic??. When a device issues an interrupt, it disables the ability to give interrupts on all devices lower in the chain. A device lower in the chain will not have its interrupt acknowledged until the first device's interrupt handling routine is finished.



Pic 7.3 An example of a Daisy chain

THE INTERRUPT MECHANISM OF OS.8MT

Vectorised interrupt has some shortcomings, as there can only be one device on each interrupt level. DATABOARD and OS.8MT thus expand the Z80 interrupt system to eight hardware interrupt levels. More than one device can be on connected to each level. The CONTROL BOARD and the I/O boards in the DataBoard series are designed to override the vectorised interrupt system.

THE CONTROL BOARD

The control board contains eight interrupt inputs, corresponding to the eight hardware interrupt input levels and one interrupt output which is wired to the "INT" pin on the CPU.
The interrupt control logic on the board is programmable so that the CPU can determine on which levels an interrupt will "pass through" the control unit.
The control unit will provide the interrupt vector to the Z80 CPU.

THE I/O BOARDS IN THE DATABOARD SERIES

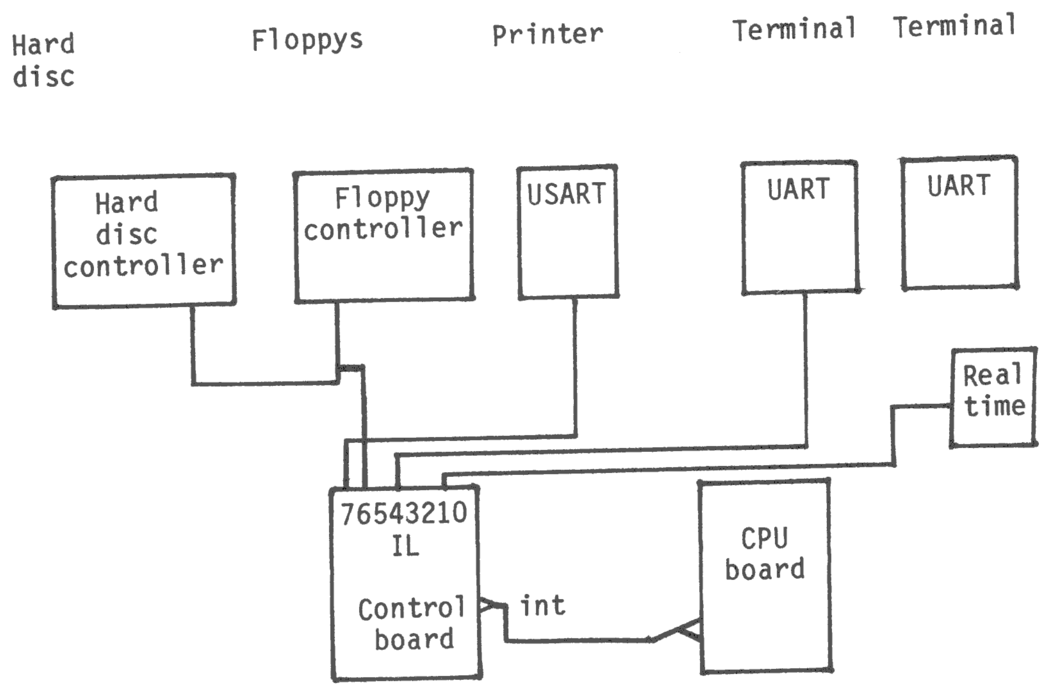
The DataBoard series contains a lot of different I/O boards, which are designed to interface the peripherals to the CPU. Most of the boards which contain inputs of some kind are interrupt driven. The boards give an interrupt by lowering the current on pin 5a on the board. Pin 5a is thus wired to one of the eight interrupt input pins on the control unit.
There are standard levels for all devices, but they can be substituted if the user has special demands.

Interval clock	Interrupt level 1
Terminals	Interrupt level 4
Printer	Interrupt level 5
Mass storage units	Interrupt level 6

Pic 7.4 Examples of standard interrupt levels for different devices.

A COMPLETE INTERRUPT

So far we have only looked at the individual parts involved in the interrupt handling, and everything may seem a bit confusing but the picture will be clearer as we go through an example.



Pic 7.5 The hardware in this example which is involved in the interrupt handling. The hard disc and the floppys are connected to interrupt level 6. The printer is connected to interrupt level 7. The terminals are connected to interrupt level 4. The real time clock is connected to interrupt level 1.

1. Let us say the computer is executing a task, the system level is 13.

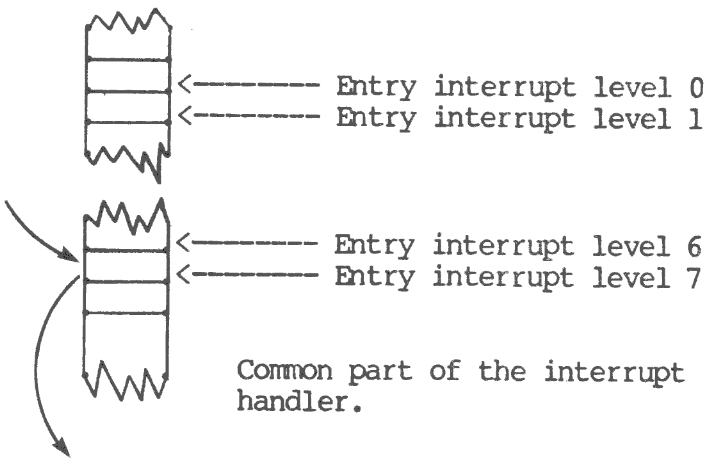
2. A hardware interrupt occurs on level 6, it can be any of three sources (The hard disc, or one of the floppy discs).

3. The control unit signals the interrupt line to the CPU.

4. The CPU acknowledges and the control unit puts out the interrupt vector for level 6 on the data bus, thereby "pretending" to be a device.

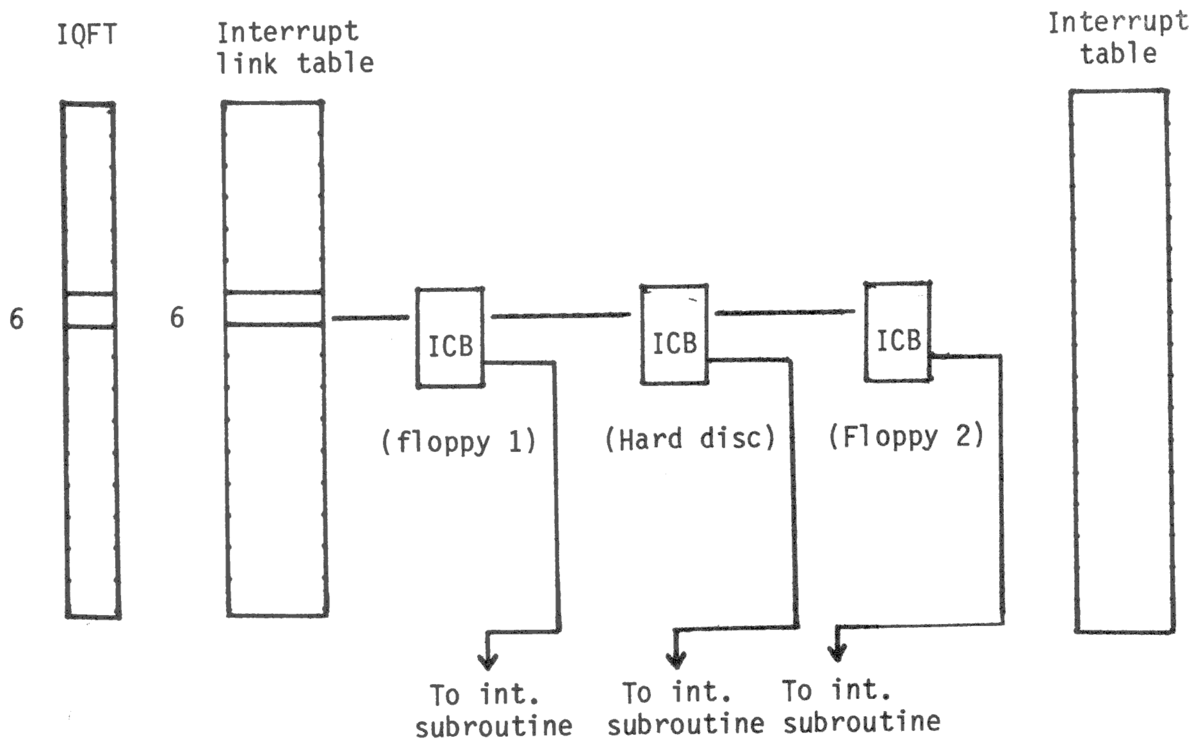
5. The CPU combines the vector with another vector in the I-register. The result gives a memory location.
6. That memory location holds the address to the system interrupt handler (SIH). The SIH actually has eight entry addresses corresponding to the eight interrupt levels.

7. In our example we end up at the entry for level 6. The first thing we do here is to make a note on which level we are, then we make an OUT instruction telling the control unit only interrupts with higher level than the present level will "pass through" to the CPU.
After this a jump is made to the part of the SIH which is common for all system levels.



8. The task's primary registers are then saved on the task's stack, and the system stack is selected.

9. The SIH now looks at the interrupt table on level 6. If the value is greater than 255, it means the driver is dedicated (we will return to this) and the value is the address to the Interrupt Control Block (ICB). But if the value is less or equal to 256, we must look in the interrupt link table for the address to the ICB.



Pic 7.6 For each device with an ICB present on the innterrupt chain the system interrupt handler tests if the device has made an interrupt.

10. The SIH begins to search the interrupt linkage on level 6. For each device it:

    1. Looks in the channel control block for the card select code.

    2. Addresses the board requesting a status value.

    3. The status value is compared with a test mask in the CCB

11. When the test masks from the device and the CCB match each other, the device which made the interrupt is identified by the OS.

12. A jump is now made to the driver routine specified by the interrupt control block which handles the data transfer.

13. When the driver routine is finished, a return is made back to the SIH (as the driver routine is a subroutine).

14. If more work is to be done on a lower level as a result of the work done by the driver routine the SIH simulates an interrupt on this level by making a note in the IQFT.

15. The rest of the linkage on level 6 is scanned, as maybe some other device on this level also has made an interrupt.

16. When finished, the SIH tells the control unit to open all hardware interrupt levels.

17. Then the SIH enters the highest on which an interrupt (hard or simulated) has been made on.

18. On the levels 9-12 there exists only one ICB so no scan is needed on this level.

19. When we are down to level 13, the task level, the current task (it does not have to be the task which was interrupted in the first place) can start to execute instructions again, until another interrupt occurs.

20. If there does not exist any ready tasks, level 15 is entered (stop mode) and the CPU halts, waiting for the next hardware interrupt

It is highly unlikely, however, that the work of the computer would proceed as described above, as an interrupt on a higher level would interrupt the interrupt handling on this level. The interval clock, for example, issues an interrupt every 10 milliseconds.

SPECIAL DEMANDS BY SOME DEVICES

As some devices, like the interval clock, issue very frequent interrupts, it would be a waste of time to go through the scanning routine every time an interrupt from one of them occurred.
Other devices need to have their interrupt serviced in a short time. The solution is to make the driver routine DEDICATED.

DEDICATED DRIVERS

If the SIH looks in the interrupt table (IQFT) and finds a value greater than 255, the value is the address to the ICB of the device with a dedicated driver. The following actions are taken:
If the status test mask shows the device has made an interrupt, a jump is made to the driver routine.
It is not possible to have more than one device on an interrupt level with a dedicated driver.

The average handling time for an standard interrupt is about 500 micro s while 250 micro s for a dedicated interrupt.

THE INTERRUPT HANDLING OF THE REAL TIME CLOCK

The interval clock has a special clock interrupt handler. This lowers the overhead of the real time handling.
Instead of entering the SIH, the clock interrupt handler is entered as the result of a level 1 interrupt. Very little work is done by the driver, but the driver may activate other types of work on lower levels by simulating interrupts on those levels.

ILLEGAL INTERRUPTS

All the devices which are permitted to give interrupts have their ICBs on the interrupt linkages. A device can, however, due to for instance static electricity, issue an interrupt when it is not supposed to do so. The ICB is in that case not found on the interrupt linkage.
The system pointer table includes an illegal interrupt counter, which is initialized to 64H at system generation time. Every time an illegal interrupt occurs, the counter is decremented. If the counter reaches zero, the crash handler is called. The illegal interrupt counter is restored by the interval clock at each clock tick.
The crash dump shows the level of the OS when the crash occured. This way the device which caused the illegal interrupts can be traced.

SUMMARY

Interrupts play a very important role in OS.8 MT as several different system routines are reached as the result of simulated interrupt, in addition to the hardware interrupts issued by hardware devices.
The interrupt mechanism of OS.8MT and DataBoard hardware makes it possible to expand the interrupt system to 16 levels (0-15).
The transition from a low system level (15,14..) to a higher system level (13,12,...) is done by an interrupt (hard or soft).
The reverse function is done by the interrupt handler.
Everything may seem a bit confusing at the time, as we have used parts of the OS which we have not discussed yet, but everything will be clearer as we learn more about those parts.

****
THE REAL TIME HANDLING
****

THE NEED OF THE REAL TIME IN A COMPUTER

Many things in the computer are dependent on the real time.

- If time sharing is used, the dispatcher needs to know when to pick a new task to run on the CPU.

- A device is given a certain time, during which it must issue an interrupt. (This is called device time-out and will be covered when we discuss resources.)

- A task may want to put itself, or another task, to sleep for some time.

- When a file is used, it is useful to have the date and year when it was created and last updated.

The OS puts a lot of different demands on the real time handling concerning the resolution of the time. Therefore four different resolutions exist in OS.8MT:

- Milliseconds

- Seconds.

- Time of day.

- Date and year.

The problems are now defined. We need to handle requests for time service by tasks on three resolution levels and be able to read the current time and date. The solution is three list structures with requests for different time services and six bytes which together hold the current second, minute, hour, day, month and year.

THE TIME QUEUES

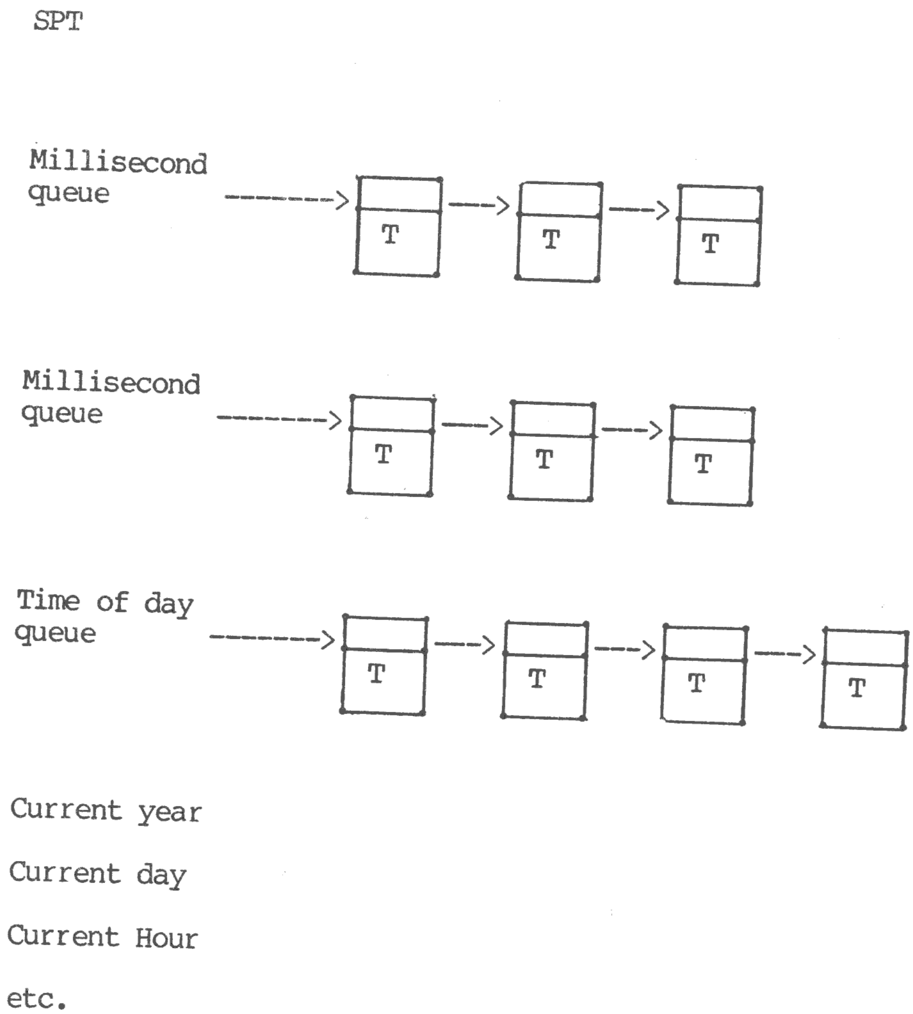
The system pointer table includes pointers to the three time queues. The request, called a node, is a table which holds information about:

- The actions to be taken when its time has elapsed.

- From where or whom the request comes.

- On the millisecond and second levels, the time before their time will elapse, and on the time-of-day level the absolute time when their time will elapse.

The system pointer table also holds the six bytes mentioned above. Two routines are responsible for managing the time queues, the clock interrupt handler, operating on level one and the real time handler, operating on level 10.

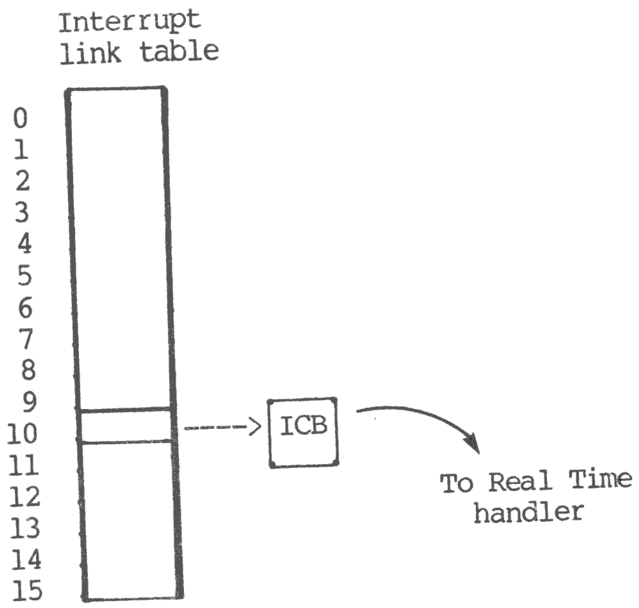


Pic 8.1 The time queues and the time information held in the System Pointer Table (STP).

THE CLOCK INTERRUPT HANDLER

The interval clock issues an interrupt every millisecond, causing the clock interrupt handler routine to be entered. The only work done by this handler is to decrement the time value on the first node on the millisecond level and, if the time value becomes zero, simulate an interrupt on level 10. (See System levels and interrupt handling, Priority levels).
The small amount of work done by the handler reduces the overhead of the time handling. When the system level has been

allowed to drop down to 10, the real time handler is entered.



Pic 8.2 The real time handler is the interrupt subroutine on level 10.

THE REAL TIME HANDLER

The real time handler removes the first node in the millisecond queue and takes the actions specified by it. This can be:

- A second has passed. The time value on the second-queue is decremented and a node with a time value of 100 is placed in the millisecond-queue. The current-time bytes are also updated.

- A device time-out counting is needed. (Device time-out will be discussed later).

- The time slice, or a time-wait node for a task has come to an end. The real time handler simulates an interrupt on level 12 which will cause the ready queue handler to be entered later.

When the time value of the second-queue's first node becomes zero, that node is removed and examined. The real time handler performs the actions specified in the node which can be:

- A minute has passed and the current-time bytes are updated. A node with the time value of 60 is placed in the second-queue. If the time of day value stored in the head node of the time of day-queue matches the current time, the actions specified by the head node are taken.

- A time-wait node (seconds) from a task has come to an end.

TASK REQUESTS

Tasks can, by making an SVC, syncronise themselves with the real time. An SVC 3 causes the task to be put in wait state for a time interval, or until a time of day occurs.
A node is put in the time queue corresponding to the request. When the time value on that node has elapsed the ready queue handler is triggered, and the task is put in the ready queue again.
A task request can also be made so that the task receives a message when the time has elapsed.
The BASIC SLEEP command uses SVC 3.

SVC 2.7 is used to set and fetch the current time and date.

The TIME command, which uses SVC 2.7, is used to set and fetch the current time.

Example: TIME 83-12-12 12.15.00 for the setting of the time.

TIME will give the current time.

SUMMARY

The real time is used in many applications by the operating system. The most important are:

- The time sharing system.
- Real time requests by tasks.
- Device Time-out conting.
- The file handling system.

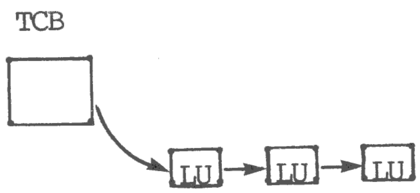
For the administration of the real time system the OS uses a number of queues, a clock interrupt handler and a real time handler.
The real time and date can be set and fetched by means of SVC 2.3 or the TIME command.
SVC 3 calls are used to delay the execution.

****
TASKS AND TASK HANDLING
****

In this chapter we are going to further discuss the control information needed to handle the tasks and the states a task can be in. We are then going to look at the parts of the OS which handle tasks.

LOGICAL UNIT

In OS.8MT device independent I/O is used. This means when you make an I/O request you specify a LOGICAL UNIT (LU) to which the I/O should be directed. LU is a number from 0 to 255. A task knows the LUs which is assigned to devices by keeping an LU QUEUE.
I/O can be directed to resources of different classes: "Devices", "Tasks", "Task Devices" and "Files".



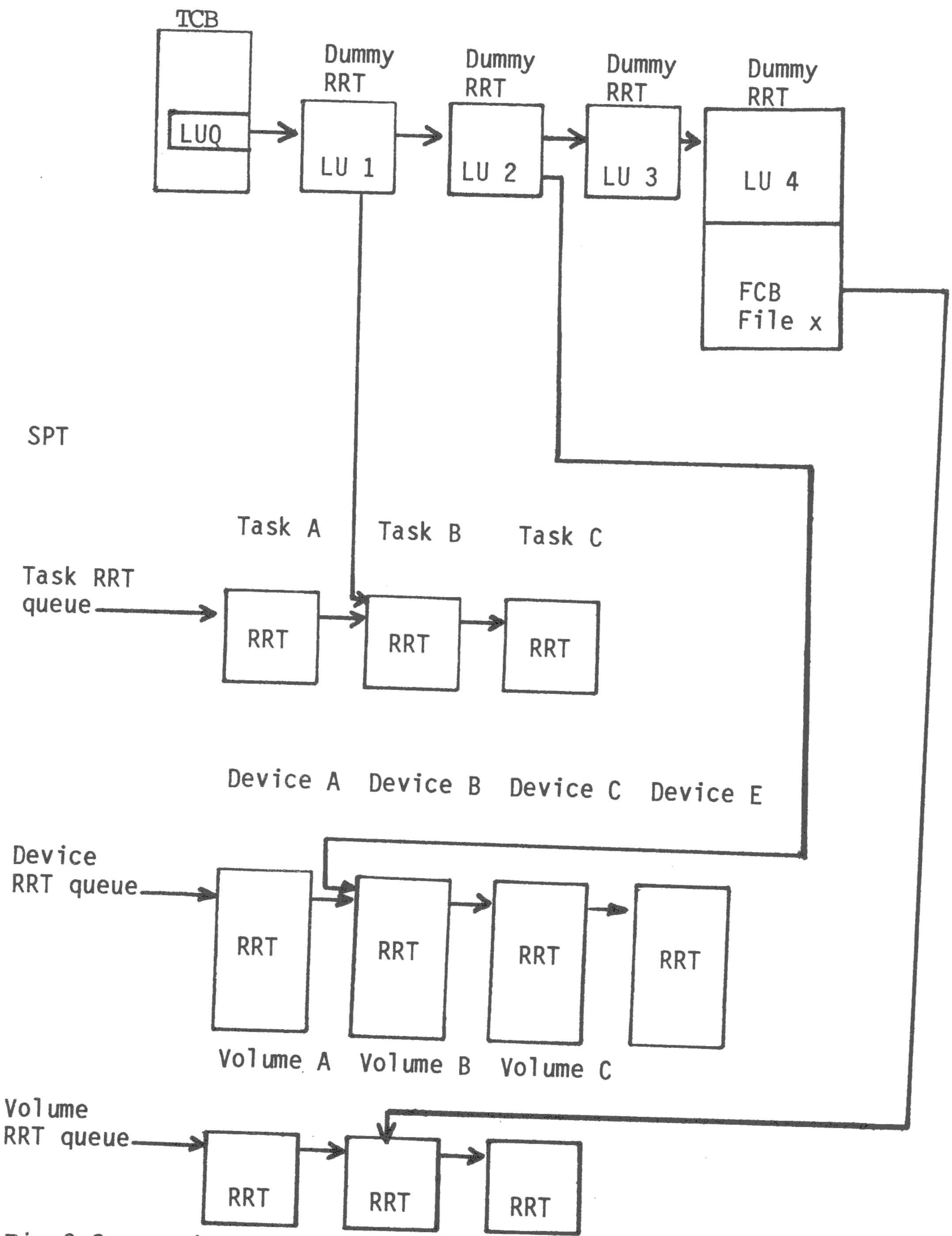
Pic 9.1 An LU queue.

The LU queue consist of RCBs with the TYPE field indicating they are DUMMIES, all holding a LU number. Before a resource can be requested by a task, the resource must be ASSIGNED to a LU in the LU queue. This is done by pointing to the requested resource's RRT from a dummy RRT in the LU queue.
If the resource is a FILE on some external memory an RCB and a FILE CONTROL BLOCK (FCB) is added to the dummy RRT.
The FCB is similar to a VCB and holds informaton about the opened file.
The procedure of assigning a device to a task is sometimes referred to as to OPEN the device. Please notice that this comes from the command "OPEN" in Basic and is NOT accomplished by the OS.8MT command OPEN, which will put a device on-line.

We will go through some examples to make things clear.

- When the editor (a task) is called, you give the command "EDIT file" where "file" is the name of the file you are going to work with. The editor task must open the file before it can use it.
- When you want an output on the printer from BASIC you make

the statement OPEN "PR:" AS FILE 1. The result of this statement is the printer will be associated with "logical unit 1".



Pic 9.2 Logical unit. Task E has opened...
- Task B as LU 1
- Device B as LU 2
- File x (on volume B) as LU 4
- Device E as LU 3

EVENT QUEUE

Sometimes the OS or another task needs to communicate with a task. It might be the OS telling the task about an SVC-call from the task having gone to completion or another task giving a message. These things can happen while the task is busy with something else and wants to deal with them later. Several of these messages can also drop in with such a rapid rate that the task has not the time to handle them.
The solution is to have a queue belonging to the task holding these messages. It is called the EVENT QUEUE and is pointed to from the TCB. Before anything can be added to the event queue, the task must enable the queue. It does so by making a SVC 6 (S6F.QENI).

TASK OPTIONS AND TYPE

You may assign the following different options to a task:

- R = resident. The task will remain in the primary memory after it has terminated.
- N = Non resident. The task is removed from the primary memory after it has terminated.
- A = Abortable. The task can be canceled from another task.
- P = Protected. The task can not be cancelled from another task.

The task option is either set with the command OPTION or an SVC 6 call.

Example: The command OPTION,PR OLLE makes the task OLLE protected and resident.

TASK PRIORITY

OS.8MT recognizes 256 priority levels within the task level. 0 is the highest level and is reserved for the systems use. Level 1-255 is available to user tasks.
Each task has two priorities associated with it:

- Task Priority, the priority currently assigned to the task. A byte in the tasks TCB holds the value. The priority can be changed by a SVC-6 call (S6F.PRIO) from the task itself or another task.
- PROPAGATED (DISPATCH) PRIORITY, a temporary priority the system sets up for a task. The dispatch priority may be raised over the task priority in some situations, which

will be described when we talk about resources.

The priority of a task can either be set with the command PRIORITY or with an SVC 6 call.

Example: PRIORITY OLLE,68 sets the priority of the task OLLE to 68.

Example: Load and start two simple BASIC programs, one which prints on the terminal and one which prints on the printer. Change the priorities of them and watch the result.

TASK STATES

Before going into the routines which handle the tasks, we are going to recapitulate the states a task can be in. When the task is loaded from external memory to the computer it becomes DORMANT, and must be started before it becomes READY and is placed in the ready queue. The head of the ready queue is picked to be the CURRENT task, the task executing instructions. Any ready task can be PAUSED by the operator or another task. A paused task will be paused until it is continued by another task. For different reasons the task can be put into WAIT STATE. Among those reasons are:

- Connection wait, the task waits for a I/O to be initiated.
- I/O wait, the task waits for an I/O request to go to completion. The task may in some cases specify that it does not want to wait for the request to go to completion. (No-wait calls)
- Time wait, the task waits for a time interval.
- Trap wait, the task waits for a task handled event, i.e. for a node to be added to the event queue.
- Task wait, the task waits for another task to change its task status or be terminated.

A paused task may be paused by the command PAUSE and continued by the command CONTINUE.

Example: PAUSE OLLE pauses the task OLLE.
CONTINUE OLLE continues the task OLLE.

THE CANCELLATION OF A TASK

A non-resident task is cancelled when it has gone to completion (end-of-task). A task is cancelled with the command CANCEL or an SVC 6 CANCEL call.

Example: CANCEL OLLE cancels the task OLLE.

Example: The following SVC call in an assembler program will cancel the program when the call is executed.

```
                    .
                    .
                    .
                    SVC 6     6,S6CAN
S6CAN      DA       S6F.CAN,0,0,0,0,0,0
                    .
                    .
```

Example: The BASIC END instruction uses SVC the SVC 6 cancel call.

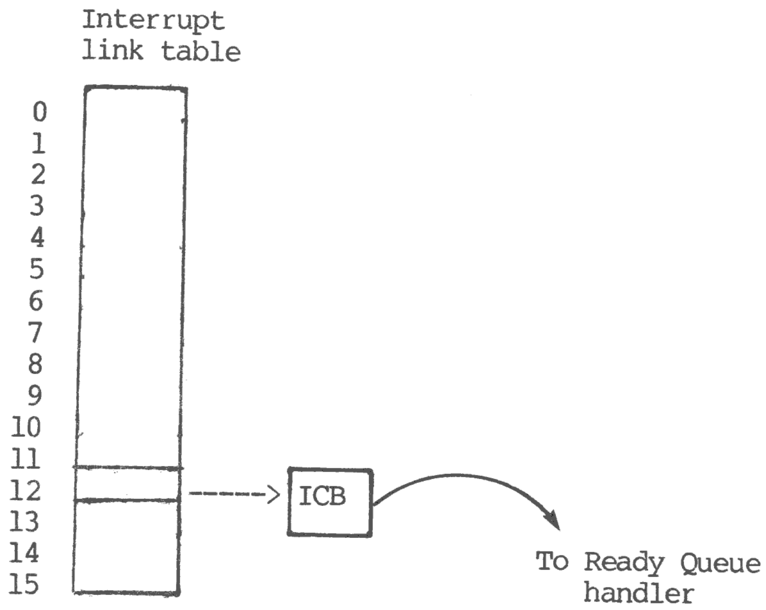
Note that if No-wait calls (See Supervisor Calls) are used the task MUST put itself in non-abortable state, before such a call is made.

THE READY QUEUE

All ready tasks are held in a ready queue. The ready queue consists of the TCBs of the ready tasks all linked together. They are ordered so that the task with the highest priorety is at the head of the queue.

THE READY QUEUE HANDLER

The ready queue handler is the interrupt subroutine on level 12, and is thus entered as a result of a simulated interrupt on that level.



Pic 9.3 The Ready queue handler is the interrupt subroutine on level 12.

The Ready Queue Handler is either called from a handler on a higher level (the real time handler or system queue handler) or by a task issuing some types of supervisor calls. The ready queue handler's job is to:

- Pick the head of the ready queue to be the current task, if no current task exists. This is called to DISPATCH the task.

- Place tasks which become ready in the appropriate place in the ready queue.

- Make suitable replacements in the ready queue if a task changes its priority. The current task can for instance lower its priority below another ready task.

In those cases, when a task issues an SVC which changes the state of a task (it can be one of several SVC 6 functions, like PAUS TASK, CHANGE PRIORITY, SUSPEND, etc) the SVC code simulates an interrupt on level 12 and the ready queue handler is entered. The system queue handler on level 11 calls the ready queue handler if a task is about to execute the initiation or termination phase of an exclusive resource. (This will be discussed later) If a task has been in wait state as a result of a real time request, the ready queue handler is triggered by the real time handler. (See the chapter about real time handling.) The real time handler also triggers the ready queue handler if time sharing is used.

TIME SHARING

As mentioned, time sharing is a technique which permits several tasks to run "at the same time". Every task has a time limit, the time it may be the current task before it is put in the ready queue again. The time limit can be individual for every task or global affecting all tasks. The TCB holds the individual time slice while the SPT holds the global.
The ready queue handler works like this when time sharing is used:

1. The head of the ready queue is dispatched for execution and puts a node in the millisecond queue. The node's time value is the time slice of the task.

2. When this time has elapsed, the real time handler triggs an interrupt on level 12 and the ready queue handler is entered.

3. The ready queue handler puts the current task in the ready queue behind the other tasks of the same priority and everything repeats from #1.

The time slice can either be set from a user task by issuing an SVC 2.7 or set by the terminal operator using the utility SLICE.

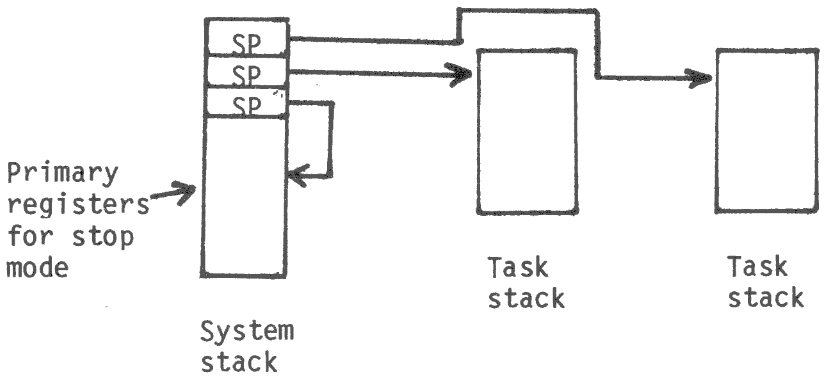
Example: See appendix A and OS.8 MT PM

Example: The command SLICE 100 set the time slice to 100 ms. The command SLICE without parameters will give the current slice.

Example: Load and start two simple basic programs which both prints something on the console. Change the slice with the SLICE command and watch the result.

STACKS

The priorities of the ready tasks are also reflected by the order in the system stack. The system stack is sometimes used when system code is executed. It includes stack pointers to the stacks of all ready tasks. The higher in the stack the pointer is, the higher is the task's priority.



Pic 9.4 The stack structures.

****
THE ADMINISTRATION OF RESOURCES
****

This chapter will describe the OS routines which connect a task to a resource and the grouping of resources into resource trees. Although this chapter is rather "heavy" it adds to the understanding of Supervisor calls and should be studied if your goal is a complete overview of the system and to master very advanced programming.

ENTERING A RESOURCE

The system code which is used when a task enters a resource is called the CONNECTION HANDLER. Let's say a task has issued a SVC to request a resource. The actions taken to handle the SVC have come to a point where the resource is about to be entered. This is how the connection handler works:

1. The handler takes a look at the TYPE byte in the RRT of the requested resource which indicates if the resource is exclusive or sharable.
   Depending on the type two actions are possible:

ENTERING A SHARABLE RESOURCE

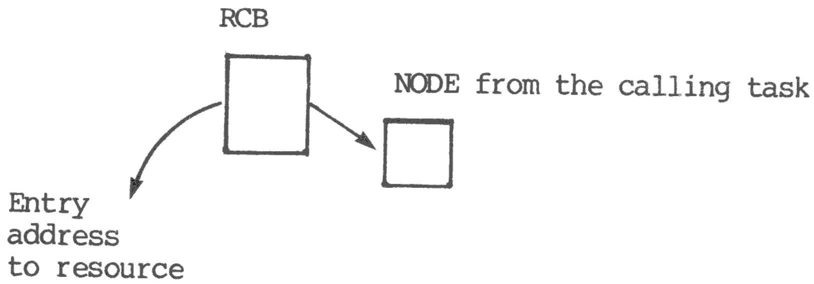
2. If the resource is SHARABLE the job is fairly simple. The RRT holds the entry address to the code of the resource, which can be used immediately, even if some other task is using it.

ENTERING AN EXCLUSIVE RESOURCE

2. On the other hand, if the resource is EXCLUCIVE things are more complicated, as only one task can use the resource at a time. As mentioned earlier, it is the RESOURCE CONTROL BLOCK (RCB) which controls the task's access to a resource. A pointer in the RRT goes to the RCB.

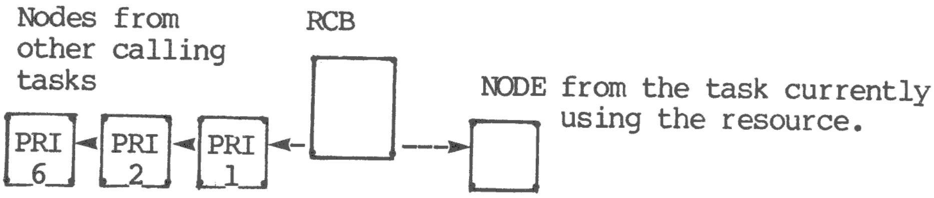
3. The connection handler examines the STATUS byte in the RCB which indicates if the resource is free or busy.

4. If the resource is free the handler changes the RCB's status to busy and connects a NODE to the RCB. The node contains information about the calling task.



Pic 10.1 A node is connected to the RCB of the resource.

The RCB now contains the entry address to the resource which now can be used by the task. If the resource is busy, the handler queues the node at the RCB's request queue. If the request queue is not empty, the handler puts the node in the appropriate place in the queue. The queue is ordered according to the calling tasks priorities.
If the resource is a task, most requests are queued as the task must be active to receive the request. A task's request queue is called the event queue (See TASKS AND TASK HANDLING).



Pic 10.2 The Request queue is seen to the left of the RCB. It is sorted in priority fashion.

The connection handler operates at the calling task's priority level but as the system is vulnerable when the handler modifies queues, the priority level is raised to 9 (Queue handling) during this time.

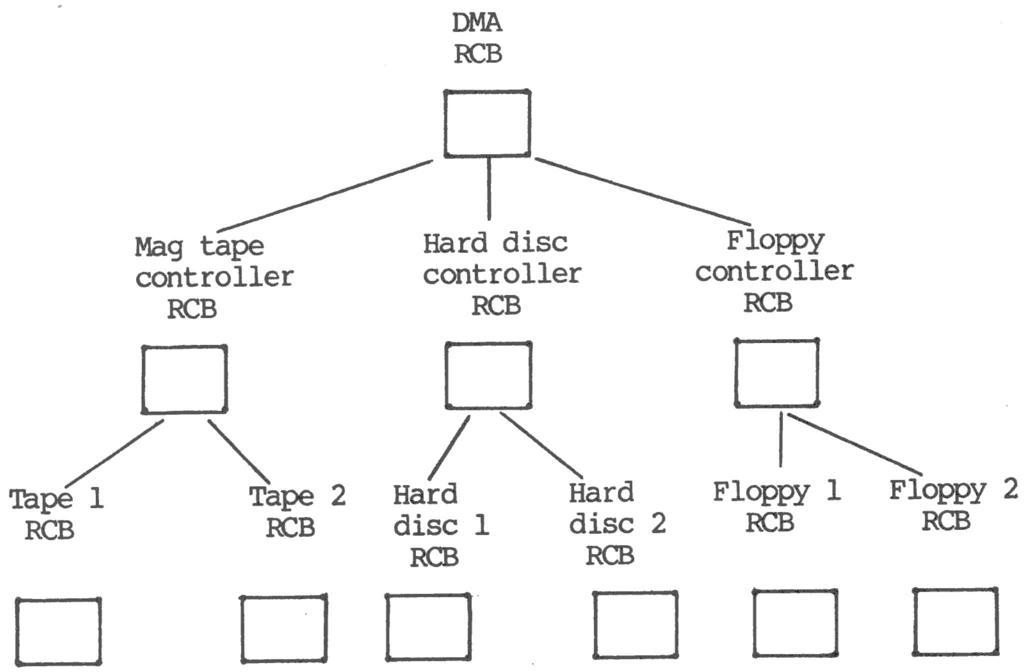
RESOURCE TREES

Up to now we have assumed that the resource requested by a task can function on its own, but this is not always the case. The floppy disc drive, for example, needs a controller routine and the DMA (Direct Memory Acces unit) to function, all of which are exclusive resources.
The connection handler must have a way of knowing which resources are dependent on which. The solution is to link the RCBs of the resources into tree structures. (See: The software

of the computer, Tree structures)
When the connection handler is about to enter a resource like the floppy disc, which needs the assistance of other resources, it finds an address to a PARENT in the RCB. The parent of the floppy disc is the controller routine. The controller RCB holds the address of another parent, the RCB of the DMA.
As the floppy disc is not the sole user of the controller and the DMA, they can be busy while the floppy disc is free. This means a task can not have access to the floppy disc or any other resource with parents, until the parents also are free.
The queueing of task requests now becomes more complicated, and is best illustrated by an example.

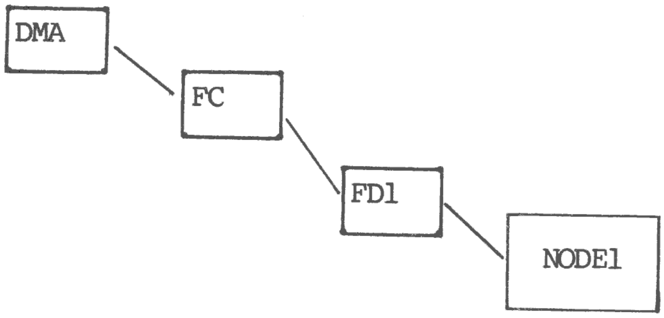
QUEUEING BY A RESOURCE TREE

Let's say we have a system with 2 floppy discs, 2 hard discs and 2 magnetic tape devices. Three different controllers and one DMA are needed. The resource tree looks like pic 10.3. When we start no task is using any of the resources.

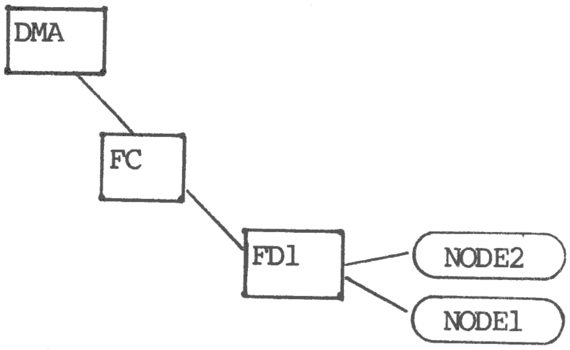


Pic 10.3 A resource tree

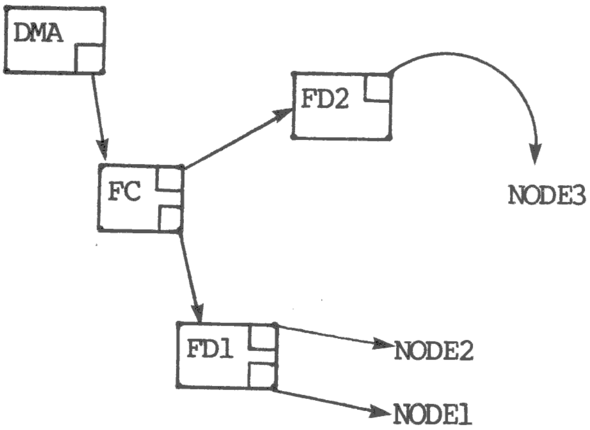
1. A request(1) is made for the floppy disk #1 which is granted and the node is connected to the floppy RCB.



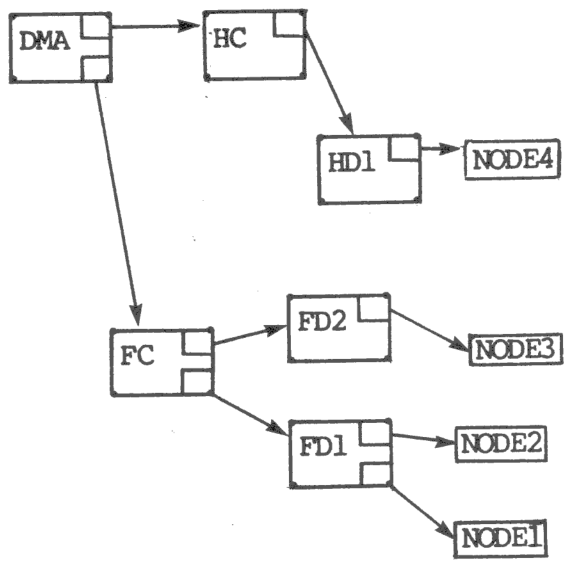
2. Another request(2) is made for the floppy disc #1 and that node is put in the request queue of the RCB.



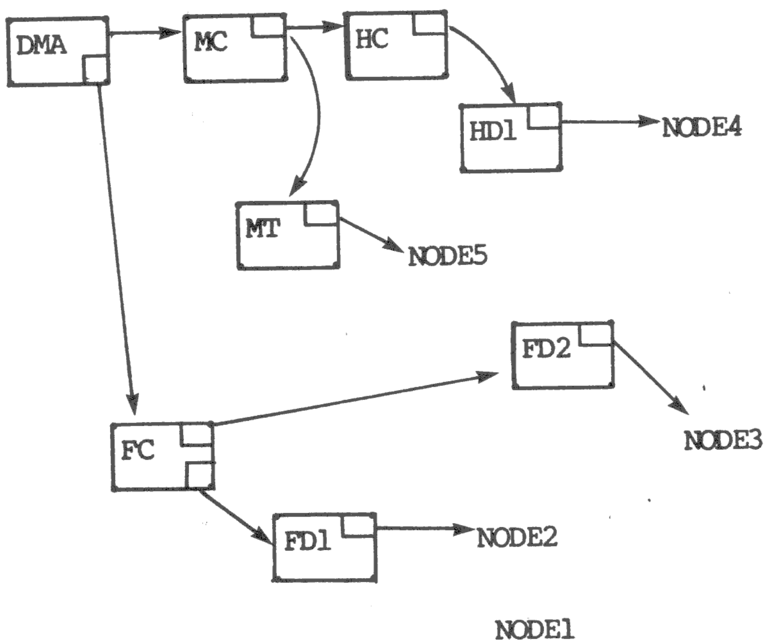
3. Now, a request(3) is made for the floppy disc #2. The node is put in the request queue and the RCB of the floppy disc #2 is pointed to from the RCB of the floppy disk controller.



4. Then, a task requests(4) the hard disc #1. The drive and controller is free, but the DMA is busy (still with request #1) The controller RCB is queued at the request queue of the DMA.



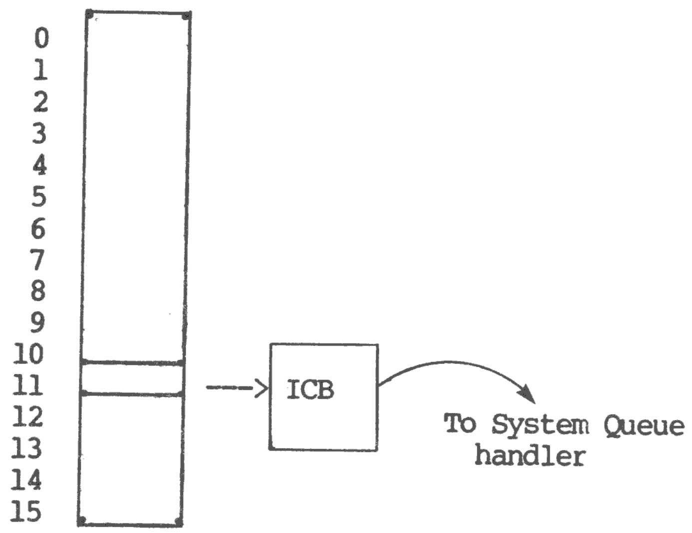
5. If a request(5) now is made for the mag tape #2 the tape station and tape controller is free, but the DMA is still busy. We have to queue the tape controller RCB at the DMA RCB. Depending on whether the most recent request has higher priority than request #4 or not, the RCB is placed before or after the hard disc RCB. (The priority of the request is the same as that of the calling task's.) In our example the call for the mag tape has the highest priority.



You can go on like this adding requests, and the result will be an increasingly complex request structure. Normally, however, calls will go to completion and be disconnected before the structure becomes too complicated.
We will for the time being ignore what happens when the actual communication between the device and the OS takes place. When we come in the request has gone to completion and it is time to disconnect the node from the RCB.

THE DISCONNECTION OF A CALL TO A RESOURCE

The system interrupt handler has added the floppy drive RCB to the SYSTEM QUEUE which include the RCBs off all resources with completed requests. The system queue handler has also been triggered.



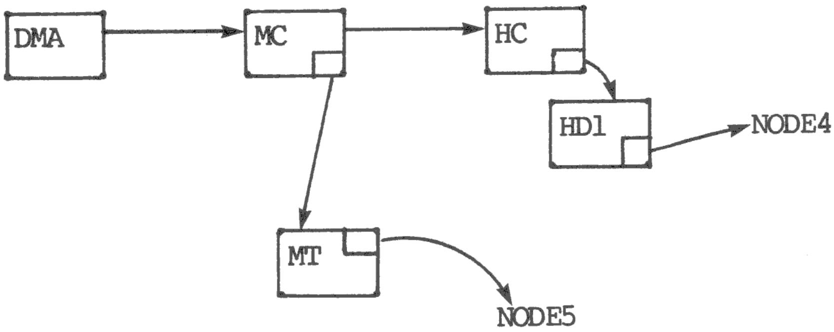
Pic 10.4 The System Queue handler is the interrupt subroutine on level 11.

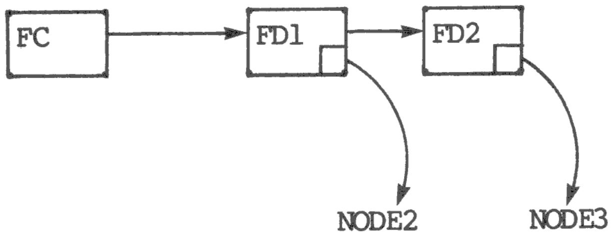
6. The system queue handler executes on level 11. Its work is to examine the RCB added to the queue and call the DISCONNECTION HANDLER. As more then one device can be present on the system queue, the system queue handler does this work for every device which has gone to completion.

7. The disconnection handler first notifies the calling task the request is completed, a process which may differ depending on the type of SVC, and which will be described later.

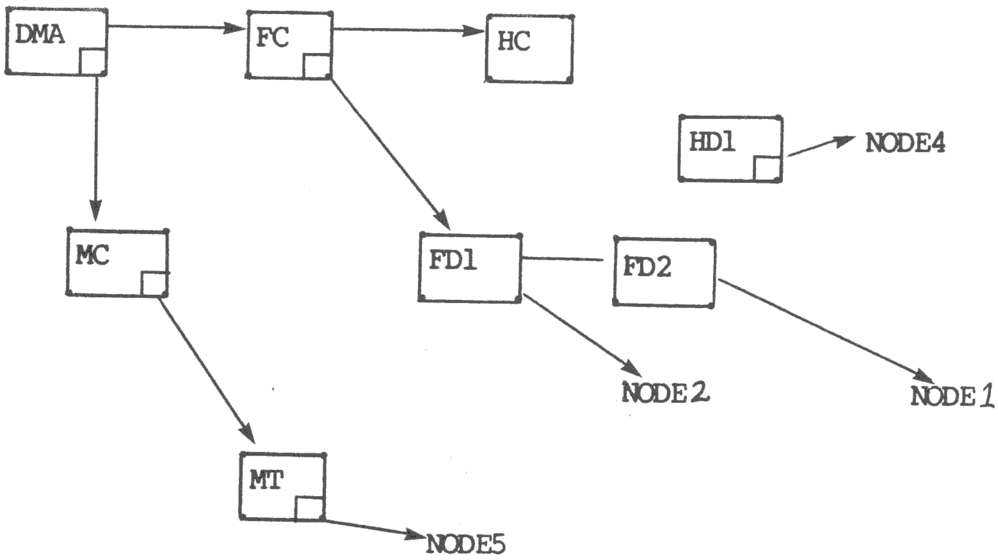
8. The disconnection handler then looks at the floppy drive 1 RCB and finds a request for it (req #2). It also finds an address to a parent, the floppy controller.

9. The RCB of the floppy controller is examined, and another request for it is detected (req #3). The handler takes the floppy drive 1 RCB and puts it in the request queue of the floppy controller RCB.
The order between floppy drive 1 and 2 is dictated by the priority of the tasks which makes the requests #2 and #3. In our example request #2 has the highest priority. The address to the DMA RCB, the parent of the floppy controller, is also found.





10. Three requests now exist for the DMA. Request #4 and #5 are already in its request queue and request #2 and #3 is about to be put in it.
As the relations between the requests priority are 5>2>4, the order in the request queue becomes 5-2-4. The handler now calls the connection handler.



11. The connection handler connects request #5 to the tree and it can start to use the resource.

PROPAGATED PRIORITY

If a task with higher priority requests the use of a resource tree, while a task with lower priority is already using it, the higher priority is "lent" to the connected task. This is called to PROPAGATE a task's priority.
In this way the tasks are connected to the resources the shortest possible time.

SUMMARY

The connection handler handles the entering of resources. Many resources are grouped into resource trees. The disconnection handler releases a resource from a resource tree.

****
DRIVER ROUTINES
****

This chapter will give a principal explanation of drivers and how they are used.

WHAT IS A DRIVER ROUTINE?

When you collect all instructions reponsible for the control of a device is a separate routine you call this a driver routine. This control can be separated into three phases:

- The initialisation of a device.
- The data transfer.
- Post processing.

By using driver routines you do can omitt device dependent instructions in the tasks.

INTERRUPTS

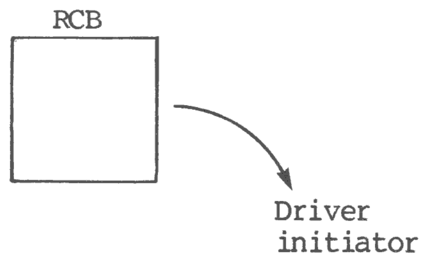
The data transfer phase is entered as the result of an interrupt which can be of two types:

- A hardware interrupt by a device.
- A simulated interrupt by the system time-out handler. This is used if the device is unable to generate hardware interrupts, or if the application demands it.

THE DRIVER ROUTINES OF OS.8MT

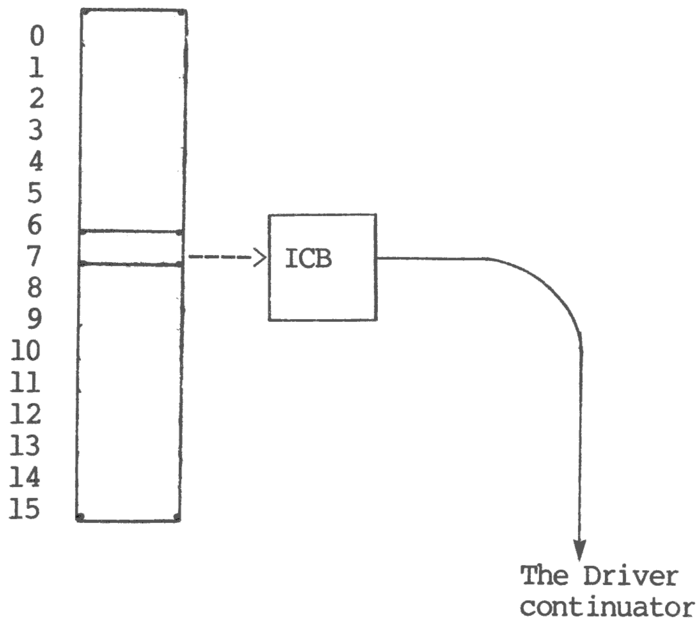
More specifically the work of the driver routines in OS.8MT is to:

1. Prepare the datatransfer. This is performed by the DRIVER INITIATOR. When this is done (if it has been done successfully), the driver initiator enables the device and allows it to issue an interrupt, or marks in the Channel Control Block that a device time-out should result in an interrupt.
   It then returns to the connection handler.
   The initiator executes with the calling tasks priority.



Pic 11.1 The driver initiator's address is found in the Resource Control Block (RCB) of the device.

2. Handle the transfer. This is peformed by the DRIVER CONTINUATOR which is the interrupt handling routine of the device and executed with the priority of the device. The continuator can either be called as the result of a hardware interrupt or a device time-out counting.



Pic 11.2 The address of the driver continuator is found in the Interrupt Control Block (ICB) of the device

3. Perform optional post-processing, which is done by the DRIVER TERMINATOR. The post processing can consist of code converting etc. The terminator can re-enter the initiator if more I/O is needed from the device.
   If the calling task is in wait state, the task is taken out of wait state and the terminator code is executed on task level.
   If the request was a no-wait request, the system handles the terminator code in interrupt mode.

4. A device TIME-OUT HANDLER which takes care of the cases when a device has failed to generate an interrupt.

DEVICE TIME-OUT

The device time-out function can be used in several ways. The normal application is when a device fails to generate an interrupt. The routines and information structures involved in the time-out system are:

- The Channel Control Block (CCB), which holds the device time-out limit and device time-out counter.

- The Interrupt Control Block (ICB), which contains the STATUS of the device (including if a time-out has been generated).

- The Device Control Block (DCB), which holds information about what to do when a time-out has been generated.

- The SYSTEM TIME-OUT MANAGER (a part of the real time manager), which is entered every 100 ms. The manager decrements the time-out counters of all devices present on the interrupt chain.
  If a counter becomes 0 the manager examines the DCB to find out what to do. Three resorts are possible:

  - The continuator of the device is called, just as if an interrupt has been issued.

  - The system TIME-OUT HANDLER is called on behalf off the device.

  - A USER TIME-OUT HANDLER is called. The handler is normally a part of the device driver.

The time-out handler is also called when an SVC-CANCEL REQUEST call has been made.

THE DATA FORMATTER

Drivers which operate on a byte basis often uses the DATA FORMATTER.
The data formatter consist of:

- The FORMATTER INITIATOR, which sets up a buffer and and can check the function code of the request for the driver initiator.

- The FORMATTER CONTINUATOR, which manages the buffer in the primary memory.

- The FORMATTER TERMINATOR, which can calculate the number of transferred bytes.

A complete example of a simple device driver is found in appendix C.

THE "CRUDE" WAY OF I/O PROGRAMMING

You may naturally use I/O instruction directly from a task to a device and poll its status. This takes, however, a long time, draines the CPU-time of the system, and is NOT recomended except when using digital inputs and other simple devices.
It takes some time to learn how to write device drivers but the advantages of having an interrupt driven system is well worth it.

A complete description of how to write device drivers is found in the OS.8 PM.

SUMMARY

Drivers contain all the device dependent instructions. A Driver consist of:

- An initiator which does pre-prosessing like checking of the SVC call and the setting up of a buffer. The initiator can use the formatter initator to acomplish this.

- The driver continuator handles the actual datatransfer and is the interrupt subroutine of the device. The continuator can use the formatter continuator to load or store bytes in the primary memory.

- The driver may also include a terminator which performs post-prosessing, and an user time-out handler.

****
SUPERVISOR CALLS
****

SUPERVISOR CALLS

We have mentioned SVCs several times before, but not specified how the OS administrates them. The different SVCs are thoroughly described in the OS.8MT Programming Manual.
There are, however, some important parts which perhaps need to be explained further to improve the understanding of chapters to come.

SVC TYPES

The SVCs are grouped into eight types depending on their function.

- SVC 1 I/O request. It is used by a task to perform all data transfer requests (I/O). It is often used together with SVC 7 calls.

- SVC 2-SUBFUNCTIONS. This SVC contains several subfunctions related to the task communication with the console operator as well as memory handling and text processing.

- SVC 3-TIMER REQUESTS. This request is used when a task wants to coordinate itself with a time interval or the time of day.

- SVC 4-TASK DEVICE. Used by tasks having defined task devices.

- SVC 5-LOADER HANDLING. Used to load an overlay.

- SVC 6-TASK REQUEST. With this SVC a task can manipulate with itself or other tasks and handle the event queue.

- SVC 7-RESOURCE ACCESS (FILES/DEVICES/TASKS). This request is used to create and delete files as well as to assign files, tasks and devices to a logical unit.

- SVC 8-RESOURCE HANDLING. This request is used to establish and remove resources while the system is running.

The SVCs are in turn divided into functions, like read or write in an SVC 1. This function along with additional information about the request are specified in a PARAMETER BLOCK.

COMMON FUNCTION CODES

There are some common function codes used to specify the request. These common function codes are supported by most hardware devices, but may be ignored by some resources. The File Handler do not, for example, support no-wait calls.

- WAIT/NO-WAIT. Wait means the calling task is to be put into wait state until the request is complete, while No-wait means control is returned to the task after initialization of the request without waiting for completion.

- UNCONDITIONAL PROCEED. If specified, the request will be rejected if the requested resource is busy. If not specified, the task will be put into connection wait until the resource is free.

- WAIT FOR COMPLETION. The task is put into wait state until a specified no-wait request has gone to completion.

- CANCEL REQUEST. This request is used to terminate a previously issued no-wait request.

THE SVC HANDLING

The OS treatment of an SVC call can be looked upon as a subroutine to the calling task. What happens is:

1. A number of routines guide the task to the right SVC-code.

2. The SVC code takes care of the request, sometimes using parts of code common to other SVCs or issuing other SVC calls.

3. When finished, a return is made to the instruction after the SVC-call (wait call) or a node is added to the tasks event queue (no-wait call).

The actual SVC process is, however, much more complicated using many routines on different priority levels, some of which have already been described. Before going into a complete example of an SVC, we will return to the modes the system goes through.

MODES

In the introduction we only made a distinction between user mode and supervisor mode. Supervisor mode is, however, divided into three different modes in OS.8MT. These are the modes of OS.8MT:

- USER MODE (UM), the mode in which all tasks run. The system level is task level. The only way out of this mode is via an interrupt, or if a task issues an SVC.

- SYSTEM MODE USER (SMU), which is used when system code is executed on behalf of a task. This mode is entered at a SVC. The system level is task level.

- SYSTEM MODE SYSTEM (SMS), the mode used when the system changes critical information such as queues, and is vulnerable. The system level is higher than task level.

- INTERRUPT MODE (IM), which is only used for interrupt service routines within device drivers, real-time update, system queue service, ready queue service and idle loop. All higher system levels than the present are enabled.

- STOP MODE (SM)

AN SVC 1 EXAMPLE

Now we are ready to look at a complete SVC example. The SVC 1 calls are made very frequently, so we will begin by looking at one.
An SVC 1 call to a file structured device, like a floppy disc, involves the file manager, which we have not described yet, so this example will concern a non-file structured device. An example of such a device is a printer, which operates on byte level.
Prior to a request for the printer the task must open the printer. It does so by making an SVC 7 assign call, specifying the device name and the LU number the printer will be associated with. A reference will then be made from the dummy RRT in the tasks LU queue to the RRT of the printer. When the request will be made, only the LU number is needed to identify the device, not the device name.

1. The point comes in the task where the SVC 1 is being made. The function code of the SVC includes information about: the requested function (write).

    - If it is a wait or a no-wait request.

    - If the request is unconditionally-proceed, or not.

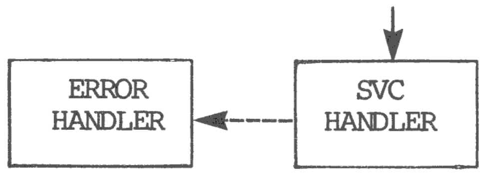
    - The LU number.

- The address to a buffer where the data to be written are held.

- The size of the buffer.

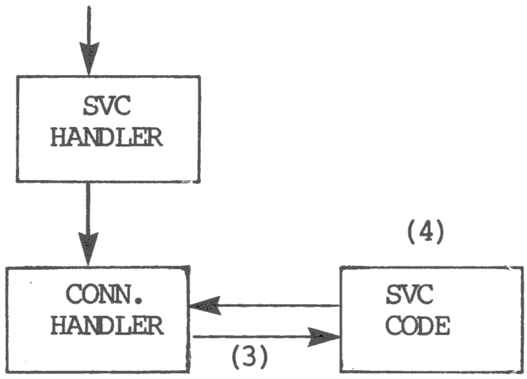
A jump is always made to the SVC handler when a SVC call is detected.

2. The SVC handler first saves the primary registers on the task's stack, and verifies that the rest of the stack has enough space for the demands of the OS. The handler then compares the SVC number with the number contained in the nodes in the SVC reference queue pointed to from the system pointer table (SPT). If no match is found, like if we have tried to make an SVC 23 (non existant), the ERROR HANDLER is called.
If a match is found, the connection handler is called.

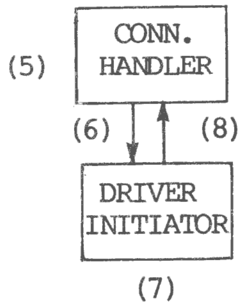


Pic 12.1 The SVC handler checks the SVC number. If invalid the error handler is called.

3. The connection handler finds the entry address of the actual SVC code in the RRT in the SVC reference queue. This is if the SVC as a whole is a sharable resource, like SVC 1. (Some SVCs like SVC 8 are exclusive resources, but we will return to them later).



4. The SVC 1 code scans the tasks LU queue to find the match for the LU number in the parameter block. When found the connection handler is called.



5. The connection handler's work is, as described earlier, to enter a free resource, or queue the request if the resource is busy. If the resource is busy and unconditional proceed is specified, the request is sent back to the task with return status 3. When the resource becomes free, the handler connects the request to the resource. The parameter block is copied into the DCB of the resource, if specified in the DCB.

6. The connection handler finds the address to the driver initiator in the RCB of the device, which is entered.

7. The initiator prepares for the transfer. With the aid of the formatter initiator, it initiates a buffer. The initiator also enables the interrupts from the device and an Interrupt Control Block (ICB) is put in the interrupt queue on the right level.

8. When the initiator is finished, return is made to the connection handler which:

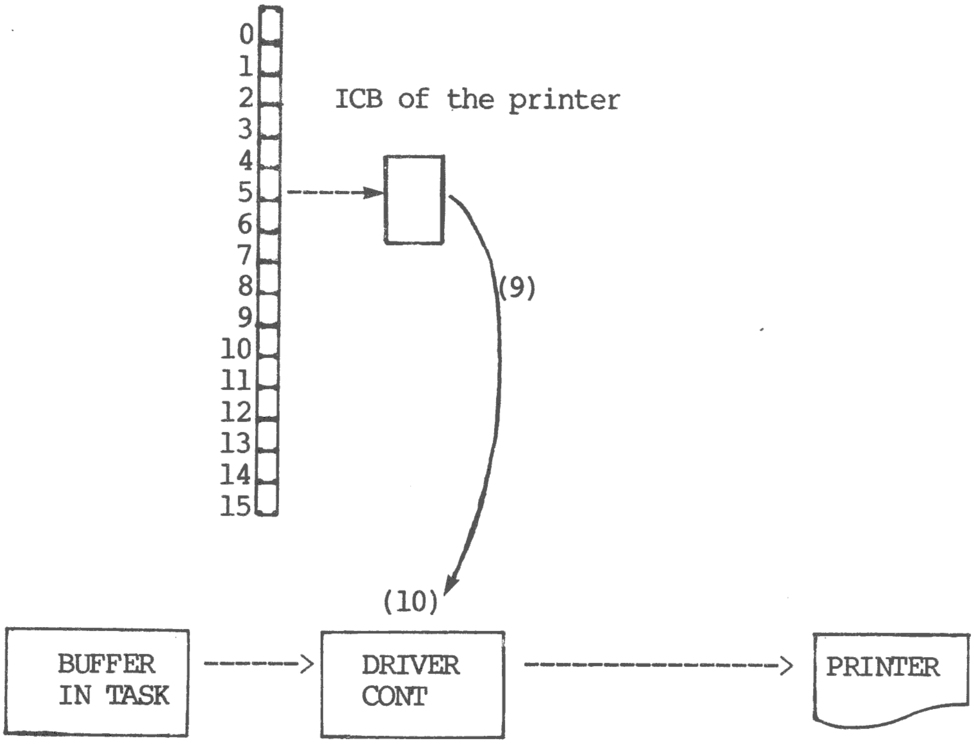
   - Starts the task if no-wait is specified

   - Puts the task into I/O wait state, if wait is specified.

When the datatransfer is ready to be done, the device issues an interrupt.

9. The SIH checks the devices, takes appropriate actions if no device is found or enters the continuator if the device is found.

Interrupt link table



Pic 12.2 The continuator is the interrupt subroutine.

10. The continuator takes care of the actual communication between the device and the buffer. The formatter is used to handle the buffer and is called by the continuator. When the transfer has been completed a return is made to the SIH, which triggers an interrupt on level 11, where the systen queue handler is reached.

11. The system queue handler calls the disconnection handler which releases the task from the resource.
We must now make a difference between wait and no-wait calls.

12. If the task is in wait state as a result of a wait call, the task is taken out of wait state and a return is made to the instruction after the SVC call.
If a no-wait call has been made, a node is added to the task' event queue, if the queue has been opened.

13. If the task is in wait state as the result of a wait call, the task is taken out of wait state and performs the termination phase in SMU-mode (system code executed at task level). When finished the task continues the execution from the instruction after the SVC call.
If a no-wait call has been made, the system handles the termination phase in IM-mode (higher than task level). When finished a node is added to the tasks event queue, if the queue has been opened.

To avoid any confusion we will show the difference between wait and no-wait calls once more.

- WAIT CALL. The task is put into connection wait state when the SVC is done, and remains in wait state until the request has gone to completion. The task is then taken out of wait state and can continue to execute instructions (if it is still is the task with the highest priority).

- NO-WAIT CALL. The task is first put in connection wait until its request has been initiated. After this the task is taken out of wait state and can continue to execute instructions.
There are three ways for a task to see if a no-wait call has gone to completion.

1. Issue a wait for completion call to the requested resource. The task is then put into wait state until the request is completed.

2. Issue a wait for event call (SVC 6 S6F.QWAI). The task will then be started when an item is added to the task's event queue. The task will be started when ANY no-wait call has gone to completion. This is the difference between wait for event and wait for completion.

3. Issue a test for event call to detect when a completion node has been added to the event queue. This is not a recommended method as the task remaines active and takes CPU time from other tasks while not doing useful work.

4. Check the return status in the parameter block of the request.
0 means the request has been done successfully, while a positive number indicates that something has gone wrong. This is not a recommended method for the same reason as method number 3.

****
THE FILE MANAGER
****

When we made the example of the SVC 1 call, we mentioned the file manager. We are now going to take a look at its structure more closely. The logical layout of a disc will also be discussed.

THE STRUCTURE OF THE FILE MANAGER

The parts of the OS which are involved when something is to be done with a file are:

- The SVC 7 code and the SVC 1 code, from which the file manager is called.

- The VCB of the disc on which the file exist on.

- The FCB which describes the file and which is created at a SVC 7 request.

- A directory manager which can manipulate with the directory on the disc.

- A bit map manager which can change the bit map of the space available on the disc.

- System buffers, which are administrated by a buffer handler.

- The driver of the device which holds the disc.

- The volume itself.

As you see, quite a lot of software and hardware are involved when we are dealing with file structured devices. The software uses 8 Kb of the primary memory.
To sort things out, we will first describe the parts individually and then look at some examples.

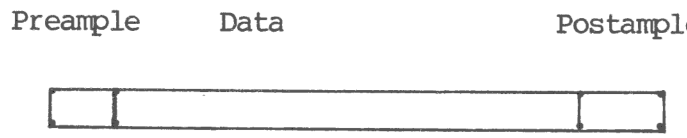
THE MAGNETIC STRUCTURE ON A DISC

The first thing which must be done when a new disc is to be used is to generate a magnetic structure on the disc. All discs are divided into a number of SECTORS. Each sector consist of:

- A PREAMBLE which consist of a sync sequence, an ADDRESS MARK with information about the sector number and a gap of three bytes.

- A DATA PART of 256 bytes.

- A POSTAMBLE with a checksum and an inter record gap before the next sector.



Pic 13.1 A sector

The OS.8MT utility DISKFORM is used when a new disc is to be formatted. DISKFORM generates the structure above and fills the data parts with binary ones.

THE LOGICAL STRUCTURE OF A DISC

Now we are ready to place the logical structure on the disc. It is normally done with the utility DISKINIT which:

1. Checks the disc for bad sectors.

2. Notates the bad sectors (if any) in an ALLOCATION TABLE, which is a bit map of the sectors on the disc.

3. Places an empty DIRECTORY on the disc. The directory is placed on default sectors for every type of mass storage units, but, if the default sector is bad it may be placed in any sector on the disc.

4. Then a VOLUME DESCRIPTOR SECTOR (VDS) is placed on the first sector of the disc. The VDS contain information about the disc and pointers to the allocation table and the directory. The first sector is the only sector on a disc which must be immaculate

You can determine the smallest allocatable element on the disc. This is called a CLUSTER and can be set with the command CLUSIZE when doing the initialization. As you can not allocate a smaller element the bit map will work with clusters.

When you do a SVC 7 allocate call the default number of clusters which will be allocated is called a BLOCK. The BLOCKSIZE command affects the size of the block.
When we have formatted and initialized the disc we have the structure described in pic 13.2.

THE DRIVERS

There exist drivers routines for every type of mass storage devices available in the DataBoard system. The complexity of the drivers is primarily determined by the intelligence of the physical mass storage device driver.

THE BUFFERS

In some access modes buffers are used to bring down the number of disc accesses. Initially the buffers are held in a buffer pool in the OS. The number of buffers is a system generation matter. Every buffer is controlled by a BUFFER CONTROL NODE which can be assigned to a task. The node holds information about:

- The currently associated device.

- The associated sector.

- If the buffer is free or not.

- If the buffer is the last one used of all buffers.

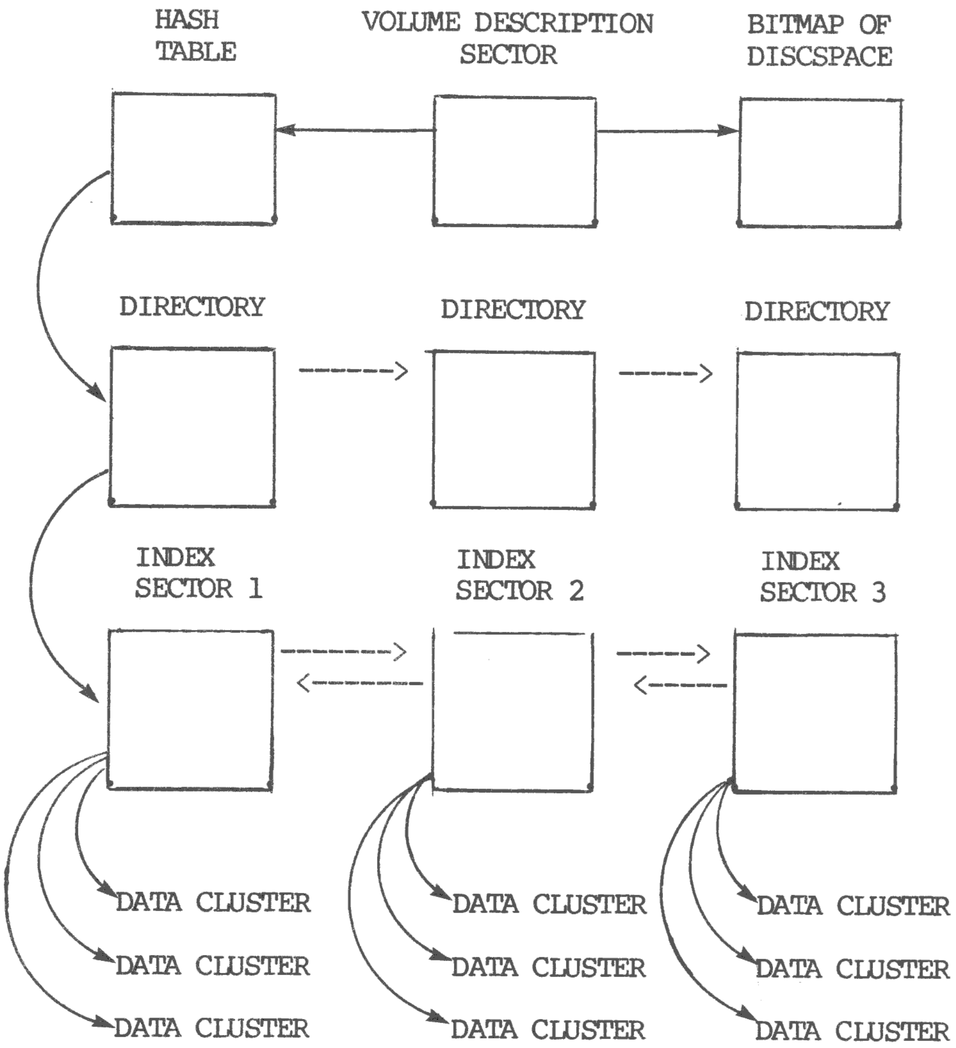
The buffers are handled by a BUFFER MANAGER, which can search the nodes and connect a task to a node. Only one task may write to the buffer at a time. When a task is being connected to a node, it increases its priority temporarily to priority level 9.

THE DIRECTORY MANAGER AND THE ALLOCATION MANAGER

In order to understand the routines which manage the allocation of a file we will take a look at how a file is placed on a disc. The directory is used to reach all files on the disc. It consists of:

- A sector with information about the number of entries in the directory and a hash table. The hash table is used to find the right directory entry. It is also used as a bit map for free space in the directory.

- One directory entry for each file on the disc. The entry keeps information about the file name and tells if somebody has opened the file for writing (only one task may write on the file at the time) A pointer also exists to the first index sector of the file.



Pic 13.2 The logical layout of a disc. The data clusters are of the size specified in the CLUSIZE command in the utility DISKINIT.

The index sectors are used to locate the sectors where the data is held. The first index sector also has information about such things as creation time of the file. While the first index sector can point out 32 clusters of data, every subsequent sector can point out 62. The routines which manage the allocation of a file are:

- The ALLOCATION MANAGER, which has access to the allocation table pointed to from the VDS on the disc. The manager can search the table for free clusters on the disc or notate in the table if a cluster has become free or occupied.

- The DIRECTORY MANAGER, which can change the contents of the directory and manage the index sectors.

It would be dangerous if two tasks used these managers at the same time when using the same disc, as they could easily allocate the same area on the disc. The allocation manager and the directory manager are thus exclusive resources within the same disc. Two tasks can naturally use the managers when working with different discs.

DISKDUMP

The OS.8MT utility DISKDUMP provides a method for viewing the logical structure on a disc. First the disc must be opend non-file structured with the command OPEN,N then the DISKDUMP command is given.
The command IN assigns DISKDUMP to a volume.

Example: IN XBC: assigns a whinchester disk.

The DISKDUMP command DUH gives the apperance of a sector on the volume.

Example: DUH 1 gives shows the volume description sector.

****
USING THE FILE MANAGER
****

To understand the function of the various parts of the file management system we need to go through some examples. First we will open a volume, a procedure which always must be done when a new disc is introduced to the system. Then we will assign a file on that volume to a task. Finally an I/O call to that file will be made.

THE OPENING OF A VOLUME

The operator opens a volume by giving the command OPEN (,(N)(P)) fd. If the device is a direct-access device, like a disk of some kind, the <fd> used is the device mnemonic, not the volume name. If <P> is specified the device is opened write protected. <N> is used to open the device non-file structured. This means no directory is present and no volume name will be established at open. The code activated by the open command uses the SVC 2.12 OPEN call. Any task can also use the SVC 2.12 call to open a device. The procedure when opening a device is fairly simple:

1. The directory manager is used to look for some information on the volume, such as volume name, directory address, cluster size, etc.

2. The information is used to build a Volume Control Block (VCB), which is put in the volume queue. The VCB contains a pointer to the device holding the volume.

Example: OPEN xbc: opens the winchester disc with the device mnemonic xbc:

A BASIC EXAMPLE OF AN OPEN CALL

```
 .
120 !
130 ! OPEN A DEVICE
140 ! ============
150 !
160 INTEGER : EXTEND
170 !
175 DIM Svcblk%(3%),Fd¤=33%
176 !
180 Fd¤="XBC "+SPACE¤(24%)
185 !
190 Svcblk(0)=2 !                Function open device
200 Svcblk(1)=12 !               Subfunction 12
210 Svcblk(2)=VARPTR(Fd¤) !      Pointer to device string
220 !
230 SVC 2,Svcblk !               Make the SVC call!
```

THE ALLOCATION OF A FILE

Files are allocated by means of an SVC 7 call. When making the call you specify:

- Fixed or variable record length,

- Record length (if fixed)

- Name

- A modifier which indicate the type of data the file containes.

- If the file is to be indexed or contignous.

AN EXAMPLE OF THE ALLOCATION OF A FILE WITH FIXED RECORD LENGTH

The PREPARE statement in BASIC is normally used to create a file, but if you want to create a file with fixed record length you must use the SVC 7 ALLOCATE call. Note that an SVC 7 ASSIGN call also must be made to establish a logical unit.

```
10 INTEGER : EXTEND
20 DIM A%(8%),B%(6%)
30 DIM A¤=29%
40 ;
50 ; "** THIS PROGRAM CREATES A FILE WITH FIXED RECORD LENGTH **"
60 ;
70 INPUT "NAME OF THE FILE TO BE CREATED    "B¤
80 INPUT "RECORD LENGTH ?    "R%
90 !
100 ! SVC 2.3 is used to give the filename the right format
110 !
120 B%(0%)=1%
130 B%(1%)=3% ! Subfunction 3
140 B%(2%)=VARPTR(B¤)
150 B%(3%)=VARPTR(A¤)+1%
160 !
170 SVC 2%,B% ! Make the SVC call!
175 !
180 ! SVC 7 is used to allocate the file with the record length R%
185 !
190 A%(0%)=1% ! Function: Allocate
200 A%(1%)=256%*16%+1%
210 A%(2%)=VARPTR(A¤)+1%
220 A%(3%)=0% ! Reserved
230 A%(4%)=R% ! The record length
240 A%(5%)=0% ! Reserved
240 !
250 SVC 7,A%
270 ; : ; "THE FILE ***  "; : ; B¤; : ; " *** HAS BEEN CREATED,"
280 ; "WITH A RECORD LENGTH OF ***"; : ; R%; : ; " ***."
```

THE ASSIGNMENT OF A FILE TO A TASK

A task must always assign a file to a logical unit before it may read data from it or write data to it.
When a file is opend by a task, different ACCESS PRIVILIGES can be specified. Some of the available priviliges are:

- Sharable read only. The file may be read by more than one task at a time, but write is not possible.

- Exclusive read only. Only the assigned task may read the file. Write is not possible.

- Sharable read/write.

- Sharable read/exclusive write

- Exclusive read/write. As long as the file is open no other tasks can access the file.

- Etc.

Example: The OPEN and PREPARE commands in BASIC given without arguments opens a file for sharable read/exclusive write.

The ACCESS MODE is also specified while making the SVC 7 assign call. We will return to access mode later.

THE HANDLING OF AN ASSIGNMENT CALL

1. A task wants to open a file on a volume and makes a SVC 7 assign call. The parameter block includes the file discriptor and the LU number which will be associated with the file. Access privilidge is also specified.

An assignment in assembler may look like this:

```
        .
        .
        SVC     7,S7ASG               The actual call
S7ASG   DA      S7F.ASGN,LU.IN,INFD   Specify the function, LU
*                                     and pointer to file to assign
        DB      S7A.EWSR              Access privilidge:
*                                     exclusive write sharable read
        DA      0,0,0,0
        .
        .
```

2. The SVC handler searches the SVC reference linkage and finds the SVC 7 block. The SVC 7 call exists! The handler then calls the connection handler.

3. The connection handler searches the list of VCBs and finds

the volume on which the file exists (the file descriptor in the parameter block of the SVC includes the volume name) The VCB block points at the SVC 7 function code, which is entered.

4. The SVC 7 code now needs assistance from the directory manager and must make an SVC 2.10 call. The list of SVC 2 reference blocks is scanned and the SVC 2.10 block is found. The block indicates that the function is exclusive, and the request is queued by the VCB of the volume the file exists on. (Remember that the directory manager is only an exclusive resource within a volume!!) When the directory manager becomes free, the request is connected and the manager is entered.

5. The manager searches the directory for the specified file and when found returns with information about the file. It also marks in the directory that the file has been opened.

6. The information is used to build an FCB, which is connected to the LU node in the LU queue of the task which made the request.

7. The call returns

DATA TYPES

Before going into an example of an SVC call to a file structured device we need to know something about the data types which are used and in which ways you can read or write data on a volume. The data types used with OS.8MT are:

- BINARY, eight bit data.

- ASCII, seven bit data with the most significant bit in the byte cleared. ASCII data can be stored in two ways:

  - Image ASCII, where the data is stored byte by byte on the file.

  - Formatted ASCII. Text files often contain a lot of spaces, which take up a lot of space on the volume. In formatted ASCII space strings are compressed into single bytes with the value 80 hex + the number of spaces. The most significant bit in every byte is set. Disc devices normally work with ASCII data in compressed form, even if image ASCII is specified.

When working with file structured devices you often work with RECORDS. A record is a number of bytes in a file. The number can be fixed or variable. In OS.8MT you determine the size of the record length, when you allocate the file. Record length=0 specifies variable record length.

A typical example of where variable record length is used is a text file where each record is a line. You could have stored the text with fixed records with the size of 80 bytes (=the length of a row), but that would mean a waste of space on the file.

ACCESS MODES

There are three ways to have access to a file:

- PHYSICAL ACCESS, read and write are performed on a sector level. This means 256 bytes are always transferred. No formatting or buffering are used, which means the only supported data format is image binary.

- LOGICAL ACCESS, read and write are performed on a logical record level. Fixed or variable record length can be used. You can either use image binary or ASCII transfer.

- BYTE ACCESS, where the file is treated as a file of bytes. In this mode you can reach any byte in the file. Logical access with variable record length is the same as byte access.

The access mode is determined when you assign the file to a logical unit with an SVC 7 call, BASIC OPEN command, etc.

AN SVC 1 CALL TO A FILE STRUCTURED DEVICE

I/O calls to file structured devices are made through SVC 1 calls. The parameter block of the SVC specifies:

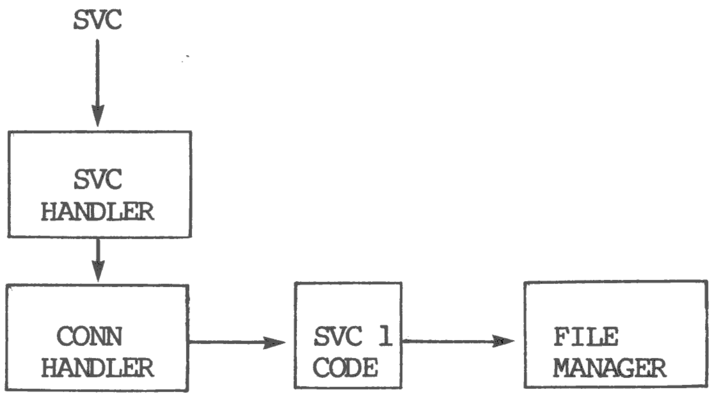
- The requested function (read/write).
- Unconditionally proceed, or not. (No-wait is not allowed!)
- The LU number.
- The buffer address in the task.
- The size of the buffer.

An assembler example of a write call may look like this:

```
      .
      .
          SVC  1,UTDATA               The SVC call
UTDATA    DA   S1F.WRIT+S1F.FASC      Function write + formatted ASCII
          DA   LU.UT                  The logical unit number
          DA   INBUFF                 Pointer to a buffer
          DA   0,0,0,0
*
INBUFF    DMB  256,' '                The buffer to fetch data from
      .
      .
```

THE HANDLING OF SVC 1 CALLS TO FILE STRUCTURED DEVICES

1. The SVC 1 call is issued. Note that the file must be assigned! The SVC handler is entered as the result of the call.
2. The handler scans the linkage, finds the address to the SVC 1 code which is entered.
3. The SVC 1 code searches for the right device in the tasks LU queue and finds an FCB containing the address to the file manager.
4. The file manager examines the parameter block to determine the requested function.



Pic 14.1 The handlers which guide the SVC call to the file manager.

Depending on the function a number of procedures can be taken.

DATA TRANSFER ON SECTOR LEVEL

If datatransfer is requested on sector level (physical access) the connection handler is immediately called.

5a. The connection handler tries to connect the task to the resource tree which at least consists of the disc and the disc controller. (See "The handling of resources, Resource trees).

6a. When connected, the driver initiator is called which tells the device what to do (sector number, read or write etc.). When finished the initiator enables the interrupts from the device and a return is made to the connection handler.

7a. The connection handler retuns to the file handler and keeps the task in wait state as previously described. (No-wait is not supported by the file handler.)

8a. When an interrupt is received from the device the driver continuator takes care of the actual data transfer. When finished the system queue handler is called.

9a. The system queue handler calls the disconnection handler for every request which has gone to completion including "our" request.

10a. The disconnection handler releases the task and takes the task out of wait state.

DATA TRANSFER ON RECORD LEVEL

If we want to have access to the file on record level (logical access) system buffering is used. The file manager must therefore first call the buffer handler.

5b. If we want to read a record there is a chance the record already exists in a buffer. The buffer handler therefore scans the buffers, and if the record is found it can immediately be transferred to the buffer in the task. If the buffers don't contain the record, the task must be connected to a buffer by the buffer handler. When this has been done the buffer handler calls the connection handler.

6b. The connection handler does its usual job, and when the task is connected the driver initiator is called.

7b. When the initialisation is finished, control is handed back to the file handler.

8b. When the device issues an interrupt the continuator takes care of the transfer from the volume to the system buffer.

9b. When the transfer is finished the system queue handler is called, which in turn calls the disconnection handler. If some formatting is needed of the transferred data (perhaps de-compress ASCII code), the termination handler is called. We must now make a difference between wait and no-wait calls.

If a task wants to write a record on a file, the formatting takes place before the data is transferred from the task to the buffer. Otherwise the procedures are similar to reading a record.

THE CLOSING OF A FILE

A file is released from a Logical Unit by making a SVC 7 CLOSE call. The BASIC CLOSE call performs the same function.
Note that it is best to close a file with an SVC call that has been assigned with an SVC call and to close a file with a BASIC CLOSE call that has been assigned with a BASIC OPEN in order to avoid any mis-match in LU numbers.

FUNCTION DELETE AT CLOSE

When working with temporary files you may specify DELETE AT CLOSE when making the SVC 7 ASSIGN call. When the file is closed it is also deleted.

DISKCHECK

If the system crashes while files are assigned, they will be marked assigned when the system is started again. The OS.8MT utility DISKCHECK is used to close all assigned files.

SUMMARY

The file manager handles all I/O calls to file structured devices. The service of the file manager is mainly requested through SVC 7 calls to allocate (create) and delete files.
When a file is allocated the name, type (indexed or contignous) and size of the file are specified.
When a task want to communicate with a file it must assign itself to the file while specifying the access privilidge (sharable/exclusive read/write etc).

****
MEMORY MANAGEMENT
****

As the 64 Kb address range offered by the Z80 CPU is too narrow for most applications OS-gives the possibility of expanding the address range to max 256 Kb. OS.8MT uses a MEMORY ACCESS CONTROL (MAC)-board to do so.

A AND B SEGMENTS

A task can consist of pure and/or impure code. The pure part only contains code, while the impure part also can include data. The pure part can be used as reentrant code by many tasks. One example is the BASIC interpretator which is written in pure code so that only one copy of it is needed while every user has a separate data area.

THE PHYSICAL ADDRESS RANGE

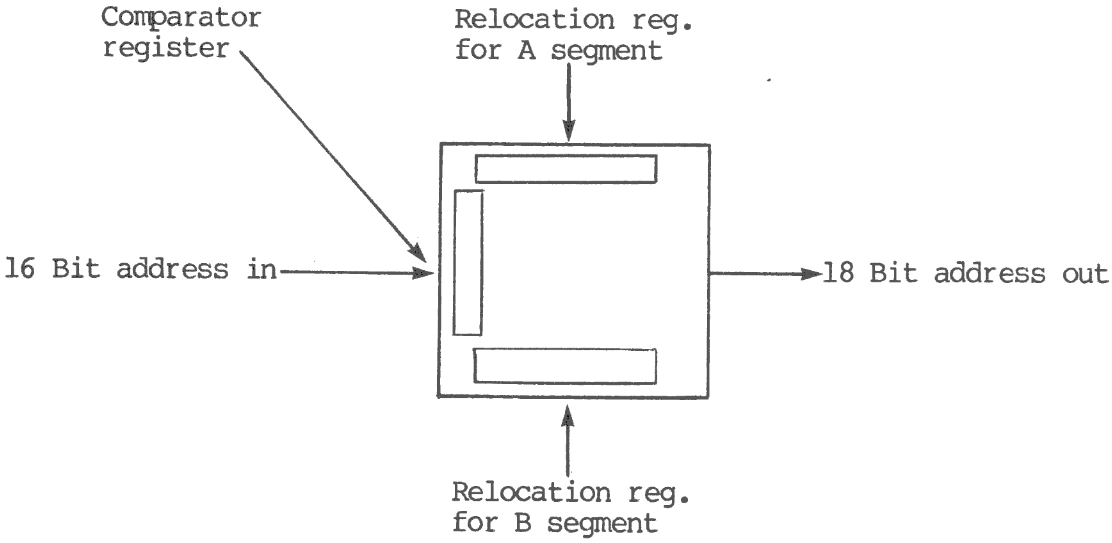
The physical address range of OS.8MT has a maximum size of 256 kbyte. The OS itself resides in the lowest 40 Kb (if both the file manager and the MTM are included). The lowest 16 Kb includes the most important parts of the OS like the system pointer table, the memory manager and the SYS-area where the dynamic data structures can be found. This part is called the Z-segment.
The part of the memory over the OS is free for tasks to use and can consist of both A and B segments.
Seee picture 4.13.

THE LOGICAL ADDRESS RANGE

The logical address range can be looked upon as a "window" of 64 Kb which sees a certain part of the physical address range. The Z-segment is always fixed at the lowest 16 Kb of the OS. The remaining 48 Kb of the logical address range is devided into an A and a B segment. The boundary between the segments can be changed but it normally leaves 8 Kb for the A segment and 40 Kb for the B segment. The two segments can be placed anywhere in the physical memory.
All logical addresses does not need to be defined in the physical memory. If the sum of the A-segment and B-segment is less than 48 Kbytes the logical addresses between the top of the A-segment and the bottom of the B-segment has no corresponding physical address.

THE MAC BOARD

The Memory Access Controller (MAC) board can be seen as a comparator with two relocation registers. It also contains a comparator register for the boundary between the segments. The relocation registers contain the physical address of the bottom of the A segment and the top of the B segment, which are used with the logical address to give the physical address. The comparator register is used to determine if the A or B relocation registers will be used.
There actually exist two pairs of relocation registers and two comparator registers which are switchable with a single OUT instruction from the OS.
More detailed information about the MAC board can be obtained in the separate data sheet.



Pic 15.1 The Memory Acess Controller.

THE PREPARING OF A TASK

In order to understand how the memory manager works, we must look at how the code and data in a task file are stored. Every task to be run under OS.8MT must be prepared with the task linker ESTAB.
If the task segments only consist of max 40 Kb of data together the task can be addressed with fixed registers in the MAC board. The memory manager only has to read where the tasks segments are stored in the TCB(s) of the task and give them to the MAC. If, however, the task is larger than 40 Kb, the task has to be segmented.
Segmentation means the A segment of a task is divided into segments of variable or fixed size. An example is the BASIC interpretator, which has four A segments of 8 Kb each and a user

area segment of up to 40 Kb.
When you use ESTAB to prepare a task you can define the parts of the pure part of the code that should be included in each segment.
The entire pute code area is always loaded into the physical memory and the segment swiching is handled by a special routine which is included in the beginning of each A-segment.

When ESTAB is executed the task:

1. Goes through the code and notates when a CALL, JUMP or RET instruction is used and which segment he is in at the moment.

2. Goes through the code once more and leaves the CALL, RET and JUMP instructions alone if they refer to a location inside the segment. If,however, they refer to a different segment it writes down the instruction plus a RESTART instruction and the segment number the location refers to. The RESTART instruction gives the entry to the memory manager.

THE MEMORY MANAGER

The memory manager has a rather simple job now when the tasks have been prepared by ESTAB. When a task becomes the current task, it sets the relocation register of the A-segment to the bottom of the A-segment of the code. The biggest A-segment of the task then determins the value of the comparator register. After this the B relocation register is set to the highest address of the task's B-segment.
When a change of A-segments is needed the memory manager just makes an OUT instruction to the MAC, which changes the value of the relocation register to the base of the new A-segment.

****
LOADING AND OVERLAYING
****

This chapter will discuss how a task is loaded from external memory to the primary memory. We will also show some examples of how overlays can be used.

ESTABLISHING A TASK

The task is the only type of code that may be executed under OS.8MTunder the identity of its own. Tasks may in turn use data supplied in different forms. It may be the editor using ASCII files, or the BASIC interpreter using binary files.
The user creates a task file from a source code with the ESTAB task.
ESTAB can link together programs written in different languages and have a very powerful command repertoire. For a complete documentation see the UTILITIES manual.

Source code
↓
Assemblation or compilation
↓
object module
↓
linker
↓
task file

Pic 16.1 The creation of a task file. The linker may link together several object modules.

THE TASK FILE

ESTAB produces a task file which contains the generated executable code with added control information. The information has the form of a LOADER INFORMATION BLOCK (LIB) which includes:

- Creation information (date, time, version)
- The tasks type, option and priority.
- The code type of the task (pure or impure)
- Name (Pure code only).

If the task contains both pure and impure code, two LIBs are needed. The format of the task file then becomes:

1. Impure library, one sector.
2. Impure code, N sectors.
3. Pure library, one sector.
4. Pure code N sectors.

If the task only contains pure code, the impure code section is omitted. The impure library then specifies the task's priority, type and options.
If, on the other hand, the task only contains impure code the pure code section is omitted. The impure code section must in that case contain executable code.
The pure and impure parts of a task can be regarded as two different tasks, which have a TCB each.

THE LOADING OF A TASK

A LOADER MANAGER is responsible for the loading of tasks in the primary memory. It is normally called from the SVC 6 when task loading is specified.

1. The manager first examines the librarys of the task file to determine the type and the size of the code that is to be loaded. It then builds one SVC 8 parameter block for the pure part and one for the impure. (If the task only contains one type of code only one block is needed.)
2. First an SVC 8 call is made for the pure part to inquire if the code is already present. (The pure part is only needed in one copy in the memory.)
3. The Loader takes additional size (given att the load call) into consideration and knows exactly the memory space

needed for the segments. A memory handler is called to demand memory. The handler uses an allocation table to keep order in the memory. (You can find the allocation table on address 3F00 Hex, just give the OS.8MT command EXAMINE 3F00,100)

4. If enough memory space exists, the handler allocates the space for the task, otherwise the call is rejected.
5. The code is now loaded into memory and after this the SVC 8 calls are made in order to build the TCBs.
6. The SVC 6 call returns.

After the task has been loaded, it must be started if it is to be added to the ready queue. This is also done with an SVC 6 call. Loading and starting a task can be combined as in the following example .

EXAMPLE OF THE LOADING AND STARTING OF A TASK

```
 .
 .
 .
80 INTEGER : EXTEND
90 !
100 DIM Svcblk(7),Fd¤=33,Cd¤=4
110 !
130 !
140 ! LOAD & START A TASK.
150 ! ===================
155 !
170 !
180   Fd¤="    KALLE      "+SPACE¤(12%) ! Start the task "KALLE"
185 !
186   Cd¤="HEJ" ! The task ID will be "HEJ" when loaded
187 !
190   Svcblk(0)=1+2 !           Function code: load and start
191   Svcblk(1)=0 !             Reserved
192   Svcblk(2)=VARPTR(Cd¤) !   Pointer to the task ID-name
193   Svcblk(3)=0 !             No parameters are given to the task
200   Svcblk(4)=0               Reserved
201   Svcblk(5)=VARPTR(Fd¤)     Pointer to the task to be started
203   Svcblk(6)=0               No extramemory is given to KALLE
204 !
210   SVC 6,Svcblk !            Make the SVC call!
250 !
260 !
 .
 .
```

The program listing in appendix A shows a general BASIC funktion to be used when loading and starting tasks.

OVERLAY HANDLING

An overlay is a piece of code that is loaded to a place inside a task. Among the reasons for using overlay technique are:

- The task is too big to be loaded into the primary memory as a whole.

- A task wants to change some parts of the code during the execution for some reason.

The overlay must also be in a task file i.e. the code must be prepared by ESTAB.
The calling task and the overlay do not have to be written in the same language. One common example is a BASIC program calling an assembler routine. SVC 5 is used to load an overlay. The needed parts of the parameter block are:

- Function code

- The address inside the task where the overlay has to be loaded. The overlay must fit within the impure segment of the calling task.

- A file descriptor of the file holding the overlay task.

It is possible to chain directly to the new code by specifying a start address within the overlay code. In this way the entire program code may be overlayed, without changing the data areas.

EXAMPLE OF OVERLAY HANDLING IN BASIC

The following example loads and starts an overlay using an SVC 5 request. Another example is found in appendix A.

```
150 !
160 ! DUE TO THE FACT THAT THE MEMORY ADDRESSING IS DONE BY A PIECE
170 ! OF HARDWARE (MEMORY-ACCESS-CONTROLLER), YOU WILL NEVER KNOW
180 ! WHERE IN THE MAIN MEMORY YOUR PROGRAMS ARE LOACATED.
190 !
200 ! THE TRICK IS TO PLACE THE ASSEMBLER-CODE IN A VECTOR INTERNALLY
210 ! IN THE BASIC-PROGRAM (MAY ALSO BE A COMMON-VECTOR).
220 !
230 ! IN ORDER TO SOLVE THAT PROBLEM, YOUR ASSEMBLER-CODE SHOULD
240 ! BE LOADED BY THE OPERATING SYSTEM AND IT'S OVERLAY-LOADER.
250 ! THE LOADER KNOWS HOW TO RELOCATE THE CODE TO THE PROPER
260 ! MEMORY ADDRESS.
270 !
280 ! THE CODE TO BE LOADED MUST BE PREPARED BY THE 'ESTAB'-PROGRAM
290 ! BEFORE IT CAN BE LOADED.
300 !
310 ! WHEN MORE THAN ONE PARAMETER SHALL BE PASSED TO THE ASSEMBLER-
320 ! ROUTINE, THE SMARTEST WAY IS TO PUT THE PARAMETERS IN A VECTOR,
330 ! AND PASS THE ADDRESS OF THE VECTOR TO THE ROUTINE.
340 !
350 !
360 ! DECLARE THE SIZE OF THE ASSEMBLER-CODE
370 ! ---------------------------------------
380 !
390 Size%=1234% ! SIZE OF THE CODE.
400 !
410 !
420 ! RESERVE SPACE FOR THE CODE
430 ! --------------------------
440 !
450 DIM Subroutine¤=Size% !         HERE WILL THE ASSEMBLER-CODE BE PLACED.
460 DIM Loadblock%(5%) !            PARAMETER-BLOCK TO LOAD THE CODE.
470 !
480 !
490 ! DECLARE THE NAME OF THE CODE-FILE TO BE LOADED
500 ! -----------------------------------------------
510 !
520 Filename¤="VOL FILENAME    ELEMENT     " ! TOTALLY 28 BYTES (PADDED).
530 !
540 !
550 ! SET UP THE PARAMETER-BLOCK FOR THE SVC-CALL
560 ! --------------------------------------------
570 !
580 Loadblock%(0%)=8%+1% !                  FUNCTION-CODE LOAD OVERLAY.
590 Loadblock%(1%)=0% !                     RESERVED.
600 Loadblock%(2%)=0% !                     RESERVED.
610 Loadblock%(3%)=VARPTR(Subroutine¤) !    WHERE TO LOAD AND RELOCATE THE CODE.
620 Loadblock%(4%)=0% !                     RESERVED.
630 Loadblock%(5%)=VARPTR(Filename¤) !      THE NAME OF THE CODE-FILE.
```

```
640 !
650 !
660 ! NOW LET THE OPERATING-SYSTEM LOAD AND RELOCATE THE CODE
670 ! -------------------------------------------------------
680 !
690 SVC 5%,Loadblock%
700 !
710 !
720 ! SET UP A VECTOR THAT CONTAINS PARAMETERS TO BE PASSED
730 ! -----------------------------------------------------
740 !
750 Parameter%(0%)=Integervar% !          PASS AN INTEGER VALUE.
760 Parameter%(1%)=VARPTR(Floatvar%) !    PASS THE ADDRESS TO FLOAT-VARIABLE(S).
770 Parameter%(2%)=VARPTR(Stringvar¤) !   PASS THE ADDRESS TO STRING(S).
780 !
790 !
800 ! CALL THE ASSEMBLER-ROUTINE AND PASS PARAMETER-LIST
810 ! --------------------------------------------------
820 Result%=CALL(VARPTR(Subroutine¤),VARPTR(Parameter%))
830 !
840 END
```

****
THE MULTI TERMINAL MANAGER
****

In the introduction we mentioned that the monitor is a task handling the communication between the terminal device and the computer. It also decodes commands and delegates the things that should be done. While a monitor in a small computer is fairly simple, a monitor which supports many terminals in a multitasking environment can seem rather complex.

THE MAIN PARTS OF THE MTM

The monitor consists of:

- A main task called MTCM, which contains all command tables and routines to delegate the work demanded by making the commands.
  MTCM can be seen as the "brain" of the monitor.

- One dummy task, called COMx for each terminal that is on line. x is a number from 0 to 7.

- One device driver, called TRMx, for each terminal on line. x is also here a number from 0 to 7.

- A dummy device, called CON, to which all calls from a task to the terminal are sent.

THE MTCM TASK

MTCM consists of two parts:

- An initialisation part administrating the building of the COM tasks.

- A code part which is common for all COM tasks.

- An administration part to guide a call to CON to the right terminal.

We will start by describing the initialisation part. At system generation time you determine the number of terminals that can be added to the system. The maximum number is 8. This will affect the size of a sysgen list held in MTCM. The same number of terminal device drivers must also be present in the system. When the initialisation of the OS is finished after a booting, control is given to MTCM, which takes the following actions:

1. The dummy device CON is built using an SVC 8 call. CON only consists of a RRT which points at MTCM.

2. An SVC 1 ATTENTION call is made to each terminal device in the sysgen list. When a terminal is switched on the ATTENTION call returns to MTCM.

3. When MTCM receives an ATTENTION call, an SVC 8 call is made to build a dummy task. The dummy task consists only of a the necessary control blocks and a small data area. The task gets the name COMx, where X is the number of the associated terminal device. After this an SVC 6 call is made to start COMx.

THE COMX TASKS

While all code necessary for the monitor exists in MTCM, every terminal needs a certain data area. The area contains:

- A stack.
- Two request queues for service by the monitor.
- A flag register to determine which mode COMx works in.
- A small buffer.

When we in the following text say that "COMx does something" we mean "MTCM executes code while the data area of COMx is used." Every operator has the feeling he is alone on the machine as his COMx task does not know anything about the other COM tasks. The only difference is the increasing response time with the number of active operators.

THE TERMINAL DEVICE DRIVER

When a COM task has been started its first action is to make a SVC 1 READ request to its terminal device. The request is for 80 characters i.e. one line. The driver initiator sets with the aid of the data formatter up a buffer in COMx data area. COMx has now nothing more to do before a line is received. For each character the operator enters, the following happens:

1. An interrupt is issued by the terminal device on the level it has been wired to.

2. The continuator of the terminal device driver is entered. It receves the character and calls the data formatter, which puts it in the buffer of COMx.

If the operator gives the CTRL-A character, the request is immediately terminated and COMx is given the control.
When the operator gives a RETURN character, the continuator sees this as request complete and

COMMAND HANDLING

The commands which may be given to the monitor can be divided into several classes, depending on what they do.

- LOCAL COMMANDS. Their code resides in MTCM, and concerns global matters in the system. Some examples are: CLOSE, OPEN, EXAMINE, MODIFY, BIAS, CANCEL, CONTINUE.

- NON-LOCAL COMMANDS. These commands are used to load and start a utility task, which is picked from the file CMD¤ on the system volume.
  Among these commands are: TASK, VOLUME, TIME, SLICE, LIB, SPACE, DEVICES, etc. The started utility task will be given the name UTLx, where x is the number of the terminal requesting its services.

- PRIMARY TASKS. When you want to have a task executed you only have to write the tasks file name. This will load and start the task. The task will be given the task identifier (TID) USPx where x is the number of the terminal requesting its service.

- BACKGROUND TASK. The operator has the option of executing more tasks than USPX. It is done with the RUN command or LOAD followed by START. A TID following the LOAD or RUN commands name the task. An example is RUN BASIC,ABC where the TID of the task will be ABC. In this way the programmer executes several tasks "at the same time", if time sharing is used.
  Please note that only the FIRST background task can be loaded and started by the RUN command. For the following tasks you have to use the LOAD and START commands.
  Compare the example in the chapter LOADING AND OVERLAYING where a task named "KALLE" is named "HEJ" while loaded.

```
task file -----> loader -------> task

  name                           task
                              identifier
                                (TID)
```

Pic 17.1 When the task is loaded it is given a task identifier. The default TID is the first four letters of the task file's name.

- COMMAND STREAMS. A convenient way of working when several commands are to be given following each other is to keep the names of the commands in an ASCII file (prepared by an editor). By giving the command ! followed by the name of the file, all of the commands in the file will be executed sequentially. !CMDA will for example execute the commands in the file CMDA.

When COMx receives a command, it uses the SVC 2.? Scan Mnemonic Table function (see the OS.8 PM) to to decide the nature of the command. The command handling routine has different "levels".

1. First a check is made to find if the first character is a "!" character. A command stream handler is in that case called.
2. The local commands are then scanned and a command routine held in COMx is called if a match is found.
3. If no match is found, the utility names are scanned. The corresponding utility task is loaded and started if requested.
4. This level only recognises the commands LOAD, START and RUN. If a match is found a suitable routine is called.
5. If no match has been found on the former levels, COMx decides the operator wants a task loaded with the filename of the command. A loader rotine is called and the requested task is loaded and started with the TID USPx where x is the name of the terminal requesting its service.
6. If no task with the filename of the command is found, a routine is called which logs the message "Seq-error" on the terminal.

The routines contained in COMx use SVC requests, just like every other task, when they have to use a system resource.

THE CON DEVICE

When a multi-user system is running we have the problem if a task wants to communicate with "its" terminal device, how do we guide the request to the right terminal? In OS.8MT the problem is solved by making all terminal requests to a dummy device called CON.
CON only consists of a RRT which points at an entry address in MTCM where a routine is reached which guides the request to the right terminal.
The terminal accessed by CON is always the terminal from which the task was started.

COMMAND MODES

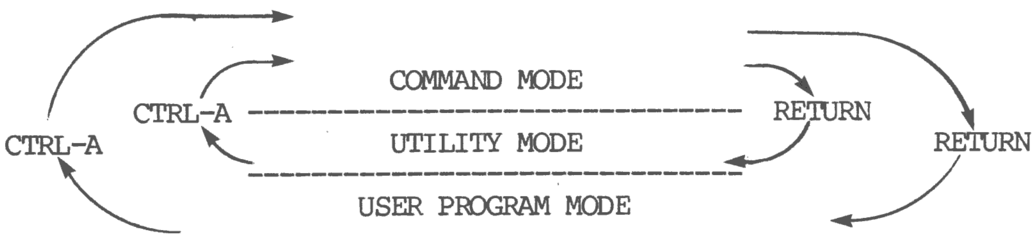
Depending on the nature of the task requesting the use of the terminal, COMx goes through different command modes. These modes have different priorities making them able to suspend eachother.

- COMMAND MODE. This is when commands can be given to the system. A "-" promt indicates the mode.
- UTILITY MODE. The mode when a command - resident or non-resident - is executed on behalf of COMx.
- USER PROGRAM MODE. A task - primary or background - is executed on behalf of COMx.

TASK REQUESTS FOR THE TERMINAL DEVICE

1. When a user program wants to perform terminal I/O it must first assign itself to the CON: device (SVC 7 ASSIGN call, BASIC OPEN command etc.).
2. Then the task can make a read or write request for the terminal (by making a SVC 1 call, BASIC PRINT call etc.).
3. COMx looks in the bit map of the modes to find which mode he presently is in.
4. If no higher mode is present the editor's request is passed to the terminal.

You can naturally have the case where more than one task want to use a terminal. A priority therefore exist between tasks. The local commands have the highest priority followed by the non-local commands and last the user programs.
This is why you can give the CTRL-command which puts COMx in command mode. A task will be paused if it makes a request for the terminal while in command mode.



Pic 17.2 The command mode is entered from utility- and user program mode by a CTRL-A command. To come back to the previous mode - just press the return key.

PROMTS

Promts give you an indication which mode the system was in when CTRL-A is entered.

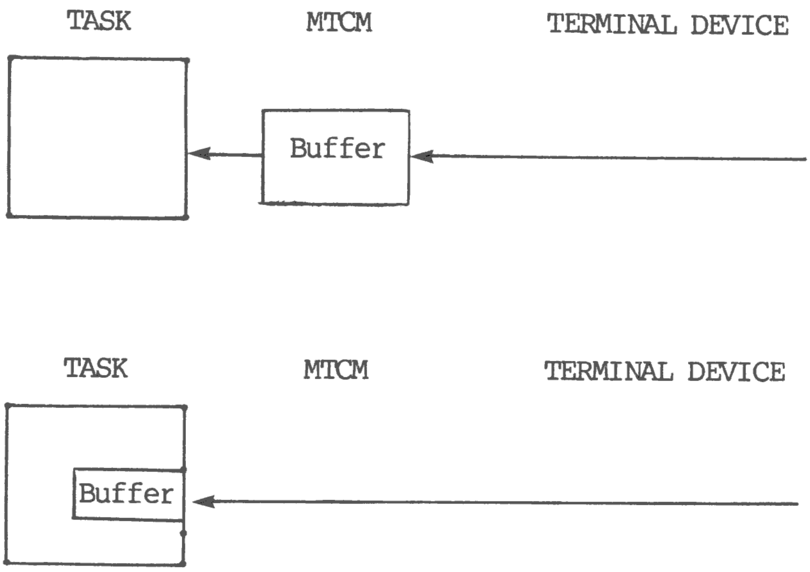
- The command mode is recognised by a "-" promt.
- If a command is interrupted by CTRL-A a "#" promt shown.
- If a utility or user task is interrupted by CTRL-A a "¤" promt is shown.

A BASIC EXAMPLE OF A "LOGG ON CONSOLE" CALL

```
.
.
.
100 INTEGER : EXTEND
110 DIM Svcblk%(4)
120 !
130 ! LOGG MESSAGE WITH TIME ON CONSOLE
140 ! =================================
150 !
155 ! The string Message¤  holds the message to be logged on the
156 ! terminal device.
157 !
160 DEF FNSvc22logg(Message¤)
170   Msg¤=Message¤
180   Msg=LEN(Msg¤) ! The length of the message is calculated
190   !
200   Svcblk(1)=2 ! Subfunction 2
210   Svcblk(2)=VARPTR(Msg¤) ! Pointer to message string
220   Svcblk(3)=Msg ! Length of message
230   !
240   SVC 2,Svcblk ! Make the SVC call
250   RETURN
260 FNEND
270 !
280 !
.
.
.
```

READ AHEAD

OS.8MT offers a read-ahead facility which means that a buffer is held in MTCM so commands can be entered while other commands are processed.
A user task may also use this facility either by keeping a self contained buffer or using a buffer managed by MTCM. Read ahead is specified in the SVC 1 call to the terminal device.



Pic 17.3 Read-ahead. The buffer can either be contained in MTCM or the user task.

Examples of read ahead SVC 1 calls are found in the OS.8MT PM.

****
TASK COMMUNICATION AND SYNCRONISATION
****

When you use tasks which are dependent on external and internal events and the time outside the computer you talk about a real time system.
A multitasking multiuser operating system like OS.8MT is one by definition. You have in the former chapters been given several examples of things happening inside the computer which are dependent on the real time. Among them are the time sharing management and the device time-out handling.
In many applications you may want to add tasks yourself that need to by syncronised with other tasks and the real time. OS.8MT therefore provides the programmer with powerful tools yo make real time programming easier.
We have already mentioned the use of the event queue of a task and how to use the SVC 3 timer request but we will now give a deeper explanation of their value.

THE EVENT QUEUE

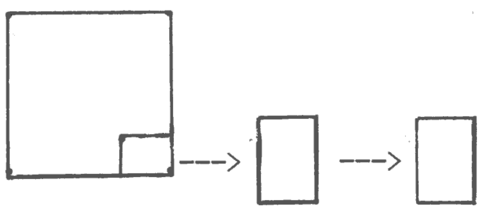
The function of the event queue is given by its name, to tell the task something has happend outside the task. These events include:

- The completion of a no-wait request to a device.

- Another task wants to leave a message or request.

The items which are added to the event queue are nodes which contain the address of the parameter block to the SVC which caused the event. They also include the TID of the task which made the SVC. If the TID is 0 it means the task itself is responsible for the SVC. Before the event queue can be used it must be opened. This is acomplished by making an SVC 6 S6F.QENI call.

Task Control Block (TCB)



Pic 18.1 The event queue is the "mailbox" of a task.

MONITORING THE EVENT QUEUE

When a task wants to check the existanse of nodes in the event queue the following resorts can be taken:

- A SVC 6 S6F.QTST (test event queue) is made. If the queue is empty the return status is 0, else the return status will be 67. A closed event queue will result in a return status of 64.

- The making of an SVC 6 S6F.QWAI (wait for event). If the queue is empty we are put into wait state until an item is added to the queue. When this happens, or if the queue is not empty, the SVC returns and we are taken out of wait state. The parameterblock can be investigated to find out the reason for the event.

If the task have other work to do and does not want to get "stuck" in wait state if the queue is empty, it must test the event queue before doing the wait for event call.

IDENTIFING EXTERNAL AND INTERNAL REQUESTS

The S6.TID field of the WAIT FOR EVENT shows if the request is internal or not.

1. S6.TID=0 An internal request which can be of three types:

   - A message from another task. S6.PRIO=6 and S6 OPT=41 (SVC function S6F.ADDQ). S6.PAR is a 16 bit message from another task without information from which task the request came.

   - A cancel request from another task.S6.PRIO=6 and S6.OPT=33 (SVC funktion S6F.CAN). This is a request from another task that the receiving task should be cancelled, and is only be received if the option of the task is non-abortable.

   - A completion node from a previosly issued no-wait call. The S6.PAR block holds the address of the no-wait SVC block.

2. S6.TID>0 An external request.

   - The S6.TID holds the task number of the external task which made the request. S6.PAR holds the address to a node with more information about the request.
     After having performed any requesting actions the task must terminate the an external request to inform the calling task the request is completed.

TASK COMMUNICATION

Task communication calls can under OS.8MT be divided into three different groups:

- One task leaving a message to another task. The receiving task can determine what to do with the message.
- One task changing the nature of another task. It may mean canceling pausing etc.
- One task wants to syncronise itself with some other tasks actions.

Whatever a task may want to do with another task it must be sure the other task really exist. If doubted a SVC 6 S6F.TST (test task) call can be done.

TASK MESSAGES

The SVC 6 S6F.ADDQ (add to event queue) call is used to give another task a message. The S6.PAR field of the parameter block can either be used itself for the message or contain a pointer to additional data. This causes an internal node to be added to the another tasks event queue. The node does not contain any inforation about from which task it came.
The PAR field of the WAIT FOR EVENT call made by the receiving task holds the address to the node. If the task is busy with other things it can save the node in a self contained queue, and deal with it later. Such a queue is sometimes called a SLOUGH QUEUE.

SVC 1 read/write requests are used to a task, similar to a request to a resource, producing an external on the event queue.

When the node no longer is neccesary it must be returned to the system and the calling task must be informed that the request is completed. This is done by making an SVC 6 S6F.QTRM (TERMINATE EVENT) call.

CHANGING THE NATURE OF ANOTHER TASK

We will go through the different calls in list form.

- S6F.LOAD (LOAD TASK).
- CANCEL TASK (S6F.CAN), is used to terminate a task. All the task's files and devices are closed.
- PAUSE TASK (S6F.PAUS), causes a specified task to enter pause state.

- CONTINUE TASK (S6F.CONT), takes a specified task out of wait state.

- CHANGE TASK TYPE (S6F.TYPE).

- CHANGE TASK OPTION (S6F.OPT).

- CHANGE TASK PRIORITY (S6F.PRI).

SYNCRONISATION WITH OTHER TASKS

Sometimes a task may want to wait for a certain move by another task. The SVC calls which can be used are:

- WAIT FOR TASK TERMINATION (S6F.TSKW) (See example below)

- WAIT FOR TASK STATUS CHANGE (S6F.STSW).

The completion of these task may also be received as a completion node on the event queue (no-wait call).

EXAMPLE OF A BASIC TASK WAITING FOR ANOTHER TASK'S TERMINATION

```
 .
 .
 .
80 INTEGER : EXTEND
90 !
100 DIM Svcblk(7),Fd¤=33
110 !
2040 !
2050 ! WAIT FOR A TASK TO TERMINATE
2060 ! ============================
2065 !
2066 !    The string Task¤ holds the name of the task to wait for.
2067 !
2070 DEF FNSvc6wait(Task¤)
2080   Fd¤=FNSvc23pack¤(Task¤) !   Pack the file descriptor.
2085 !
2090   Svcblk(0)=40 !              Function wait for task term.
2091   Svcblk(1)=0 !               Reserved
2092   Svcblk(2)=VARPTR(Fd¤)+4 !   Pointer to the packed name.
2095   SVC 6,Svcblk !              Make the SVC call!
2097 !
2100   RETURN FNS0rs !             Pick up return status.
2110 FNEND
2120 !
2280 !
2290 ! CONVERT A HUMAN-FILENAME TO OS.8-FILEDESCRIPTOR
2300 ! ==============================================
2310 !
2320 DEF FNSvc23pack¤(Filename¤)
```

```
2330    F¤=Filename¤+CHR¤(0) !          Note the termination character!
2340    Fd¤=CHR¤(0)+SPACE¤(28)
2350    !
2360    Svcblk(0)=9                      Function: Pack
2370    Svcblk(1)=3 !                    Subfunction 3
2380    Svcblk(2)=VARPTR(F¤) !           Pointer to the string to pack
2390    Svcblk(3)=VARPTR(Fd¤)+1 !        Pointer to receiving area
2400    !
2410    SVC 2,Svcblk !                   Make the SVC call!
2420    RETURN RIGHT¤(Fd¤,2)+LEFT¤(Fd¤,1)
2430 FNEND
2440 !
2560 !
2570 ! FEED-BACK RETURN STATUS
2580 ! ======================
2585 !
2590 DEF FNS0rs
2600    S0rs=SWAP%(Svcblk(0)) AND 255
2610    RETURN S0rs
2620 FNEND
```

SELF DIRECTED CHANGES

Most of the calls mentioned can also be used by a task to change some aspect of itself. The changing of TYPE, OPTION and PRIORITY can be done. A task may also PAUSE and CANCEL itself.

SYNCRONISATION WITH THE REAL TIME

We have already mentioned SVC 3 when talking about the real time handling. The task can either make the timer request with wait or no-wait.

- WAIT. The task is put into wait state until the specified interval has elapsed, or time of day occured.

- NO-WAIT. A node is added to the tasks event queue at that time.

EXAMPLE OF A BASIC TASK DELAYING ITS EXECUTION

```
 .
 .
 .

410 !
420 ! DELAY THE EXECUTION
430 ! ==================
440 !
450 DEF FNSvc3delay(Milliseconds)
460   !
470   Svcblk(0)=1 !                Function code "milliseconds"
480   Svcblk(1)=Milliseconds !     Give the millisecond value
490   !
500   SVC 3,Svcblk !               Make the SVC call!
510   RETURN
520 FNEND
530 !
540 !
 .
 .
 .
```

By changing the function code to "2" the delay is given is seconds, instead of milliseconds.

The BASIC command SLEEP uses the SVC 3 call.

****
TASK DEVICES AND EVENT DRIVEN TASKS
****

In many applications it is not sufficent just to make a simple I/O call to a device. Communication protocols or special formatting may be needed. It is not good programming to load the tasks with the burdon of these jobs.
One solution is to use a dedicated task which makes the I/O calls to the device and performs the hard work. The tasks wanting to communicate with the device then make their calls to the dedicated task, not the device.
An example, perhaps is a little more accessable, is a SPOOLER. The function of a spooler is to accept all inputs to a device, for instance a printer, even if the printer is busy. The spooler instead writes the data temporarily on a disc. When the printer becomes free the spooler transfers the data on the disc to it. The tasks have no idea their output does not go directly to the printer as the spooler from their point of view in all actions resembles a printer.
Such a spooler is available as a utility to OS.8MT. The spooler renames the printer device PR: to PRA: and establishes a new device called PR: which is the input to the spooler.

There are actually two kinds of these dedicated tasks:

- An EVENT DRIVEN task, to which the other tasks directly perform their I/O.
  From the users point of view this simply means direct the I/O calls to a task instead of a device.

- A TASK (the owner task) which uses a number of TASK DEVICES, to which the other tasks make I/O calls.

EVENT DRIVEN TASKS

The best way to get an understanding of event driven tasks is to go into an example right away.
The task A handles the communication with another computer via an USART board which is connected to a telefone line. When using syncronous data transfer, a protocol is used to add the neccesary control data when sending data and subtract them when receiving data.
The data to be sent and having been received is kept into separate buffers contained in task A.
When a task wants to make an I/O call to the other computer all it has to do is to make an I/O call to task A.
To go more into details, what happens is:

1. The task assigns itself for read or write to task A by making an SVC 7 ASSIGN call with the task name instead of the device name.

2. The SVC 1 read or write call is made to task A. The same procedure is used as in a regular I/O call to a device. The OS treats the request in the same way, except for the fact that the request is queued or connected to a TCB instead of a DCB (actually the RCB of the TCB).

3. The request connected to the TCB an external event node to task A. If A has nothing to do and has made an WAIT FOR EVENT call the request can be performed, else the node remains in the event queue until task A is ready to perform the request.

4. As described in the previous chapter the address to the external node is given to task A in the parameter block of the WAIT FOR EVENT call.

5. It is now up to task A to check the I/O parameter block to find out what kind of I/O call which has been made, and where to put or get the data in the calling task.

6. If task A can service the request the datatransfer to or from the calling task can start. It is again the responsibility of task A to take the neccesary actions.

7. The other task may be in an other segment than task A. Therefore an SVC 2.6 "Transfer from other segment to me" call must be made. The node from the calling task includes information about the segment base of his buffer. This information is given together with other details about the transfer in the SVC 2.6 call.

8. When the request has been serviced, task A makes an SVC 6 TERMINATE EVENT call. This call terminates the request of the calling task in the same way as a normal I/O call would be terminated returning a return status code to the calling task.

It is important to notice it is the duty of the event driven task to fetch the parameterblock of the I/O call, and to manage the datatransfer. These things are normally done by the OS or a driver.
The OS.8MT PM contains detailed information and examples of event driven tasks.

A complete program listing of an event driven task is found in appendix B of this manual.

OTHER APPLICATIONS FOR EVENT DRIVEN TASKS

Event driven tasks are naturally not limited to handling protocols. One application is to serve as a pipeline to other tasks. A pipeline can be looked upon as a "mailbox" to which

other tasks can leave and collect data.
A pipeline utility is available in OS.8MT (see the OS.8MT UM) but under some circumstances you may want to create your own mailbox.

TASK WITH TASK DEVICES

Tasks with task devices are somewhat more complicated than event driven task but has some added features as other tasks directly can perform their I/O to devices created by an owner task. Task devices may be written so they all ways resemble a physical device. The following parts are involved when using task devices.

- An initalisation part of the owner task which builds the neccesary task devices. SVC 8 calls, specifing TASK DEVICE, are used.

- Device Control Blocks (DCB) for each task device to which other task may make their I/O requests. The DCBs are created by the SVC 8 call

- A small administration part of the owner task which receives any no-wait completion nodes from SVC requests issued by its task devices and re-triggs the appropriate task device for action.
  In many cases the user may chose to have more code directly within the owner task instead of the task device code part.

- Driver routines which performs the actual requests in exactly the same way as a device driver would, but with the priority and identety of the owner task instead of the interrupt level.
  As the routines execute on task level they may do more complicated protocol handling, formatting etc than a device driver and may issue any type of SVC requests which a device driver is forbidden to do.
  The driver routine is within the memory area of the owner task, but is entered directly from the OS at a request to a control block of the task device. The driver routine returns directly to the OS when completed, like described earlier when talking about devices.
  A normal device driver which is mapped in as the pure segment of the calling task may however directly access the data areas in the impure task area, while a task device must use an SVC 2.6 call (Intersegment Data Transfer) to access the data areas in the physical memory of the calling task.
  The owner task itself must be in WAIT FOR EVENT status to enable the execution of the driver routines.

We will now return to the same example as we used when we described event driven tasks.
The task device handles a protocol which is used to receive and transmit data to another computer via an USART-board and a telphone line.
The communication with the USART functions in this way:

- READ. The task device makes a no-wait SVC 1 request to the USART whenever it is able to receive data.

- WRITE. When the task device has received data from a task which is to be sent to the other computer it makes a no-wait SVC 1 write call to the USART.

When the requests to the USART has gone to completion the owner task gets a node added to its event queue. The administration part of the owner task examines the node and re-triggs the appropriate task device to perform the required actions.
takes the neccesary actions. When a task wants to perform I/O to the other computer the following actions are taken:

1. The task assignes itself to one of the task devices by making an SVC 7 ASSIGN call. (in our example two task devices are used. One for read and one for write)

2. An SVC 1 READ call is made to the "read" task device.

3. The SVC call is treated like a normal call to a device, and the OS enters the task device. Only one entry point exist in the task device taking care of all the functions of a device driver: Initialisation, Continuation, Time-out handling as well as cancel request functions.

4. The read call can be handled by the owner task when the node is the head of the request queue, and the task device has made an SVC 6 WAIT FOR EVENT call.

5. The driver can now initialise the data transfer. There is, however, a chance no data exist in the read buffer. In that case the request is suspended temporarily.
   The task device had previosly issued a no-wait I/O request to the USART. When this request is completed the owner task receives the completion node and re-triggs the task device with an SVC 4 Trigg Initiator request, to continue the data transfer to the calling task.

6. When finished the task device exits with "carry" set to indicate that the request is completed, just lika a physical device driver would do.

A write call to the other computer is made in a similar way, exept the call is made to the "write" task device.

A complete listing of a task using task devices is found in the OS.8MT PM.

A WORD OF WARNING

As you may have noticed, many things which normally are taken care of by the OS is performed by the event driven- and symbiont tasks themselves. It is therefore a good idea to be carful when using them, or else the followings may not be what you have expected.

****
USING OS.8 MT FOR THE FIRST TIME
****

When working with OS.8MT for the first time you are perhaps a little unsure on how to use all the commands and utilities. While the OS.8MT Operators Manual in detail show all available functions, this chapter will give a brief guide on how to use the computer so you can get started quickly. Please note that the complete syntax and possibilities with each command in many cases NOT will be given here. For this you must use the OS.8MT Operators Manual.

THE COMMAND SYNTAX

There is a general command structure which is used both in this text and in the OS.8MT OPERATORS MANUAL.

```
MNEMONIC(,(SWITCHES)(,ADDMEM)) ((PARAMETER1),(PARAMETER2),...
```

Example: COPYLIB,GV,20 VOL1,VOL2

- The MNEMONIC is the command, utility or task you want to use. In the example above it is the utility COPYLIB.

- SWITCHES specify options in the specifyed command. In the example above the swich "G" indicates that the COPYLIB utility should by executed immediatly while the "V" swich indicates that the copy process should be verified afterwards.

- ADDMEM specifies the amout of extra memory added to a program. In the example above the work area is expanded 20 Kbytes this makes the copying process faster. ADDMEM can also be given in Kilobytes. COPYLIB,GV,20000...... thus give the same result.

- PARAMETERS are separated from the mnemonic and the switches with one or more spaces. Parameters are separated from each other with a comma. In the example above the parameters are VOL1, the disc to copy from, and VOL2 the disc to copy to.

- BRACKETS () indicate optional arguments. Commas inside brackets must be entered if the optional argument is chosen.

- SPACES may mot be used in any place in a command except before the parameters. If a parameter contains a space it must be given whithin apostrophes.

FILE DESCRIPTORS

The (somewhat incomplete) syntax for a file descriptor is:

VOLN:FILENAME/MODIFIER

VOLN is the name of the volume the file resides on. If not specified the file is fetched from the system volume.

FILENAME is the name of the file.

MODIFIER os the type of the file (see the OS.MT OM). It is not always necessary to include the modifier.

Example: EDIT VOL1:TEST is used to edit the file "test" on the volume "voll". EDIT VOL1:TEST/A would have derived the same result as the editor works with ASCII files indicated by the modifier "A".

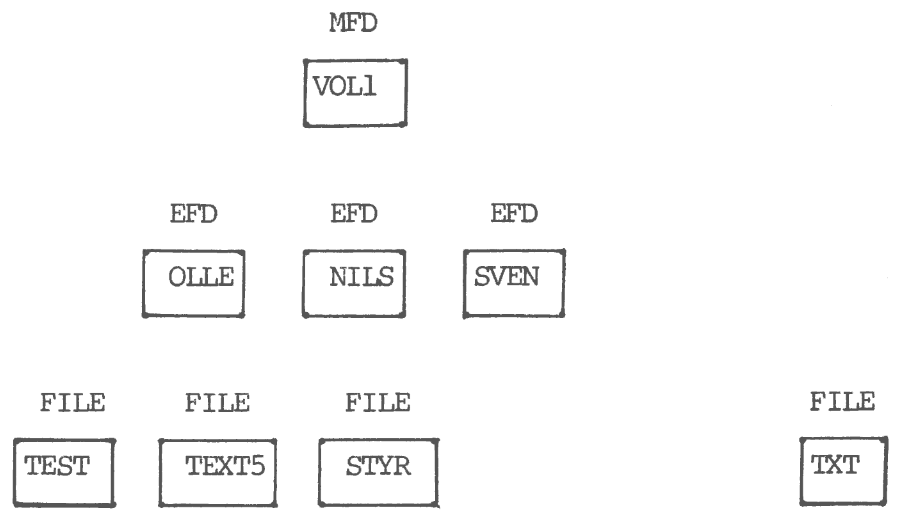
OS.8MT recognises element file directories (efd). An efd is a sub directory to the master file directory

If you want to reach a file in a element directory the syntax is:

VOLN:FILENAME.ELEMENT/MODIFIER

FILENAME is here the name of the element directory.

ELEMENT is the name of the file in the element directory.



Pic 20.1 Some of the files and directorys on a volume. The name of the volume is "VOL1". Three Element File Directories are shown: "OLLE", "NILS" and "SVEN"

Examples: The file "TEST" has the file descriptor: VOL1:OLLE.TEST
The file "TEXT5" has the file descriptor: VOL1:OLLE.TEXT5
The file "STYR" has the file descriptor: VOL1:NILS.STYR
The file "TXT" has the file descriptor: VOL1:TXT

SYSTEM VOLUME

OS.8MT always keeps a system volume which normally holds the system programs and commands. When you swich on a DataBoard computer the system searches for a volume from which it can load the operating system. This volume becomes the system volume. If you want to reach a file on the system volume the file descriptor does not have to include the volume name.

Example: If the name of the system volume is "VOL1" the file TEST in the example above has the file descriptor: OLLE.TST The file desctiptor VOL1:OLLE.TEST may naturally also be used.

ORIENTING YOURSELF

When the OS has been booted you have received a signon on the terminal followed by a promt. You are now free to enter any commands.

SETTING THE DATE AND TIME

Your first step ought to be to set the time in the system. To do this you give the command:

```
TIme YYYY-MM-DD,HH.MM.SS
```

Where YYYY=year, MM=month, DD=day, HH=hours, MM=month, SS=seconds

If you give the TIme command without parameters, the current time will be displayed.

CHECKING THE DEVICES

Now it is time to check the devices present in the system. This is done with the command:

```
DEVices (fd)
```

This result in a list of information about the devices. It may look like this:

MNEM	NR	STAT	TYPE	VOLN	DCB-ADR	REQ	SVC-BLK	CS	IL
PR	6				2597				
MFP0	8		DIR	ABC	2677				
MFP1	9	OFFL			260F				
TRM0	64				28BB				
TRM1	69				24BA				
CON	65				2700				
NULL	66								

From this information we can find out the system is configured with:

- A printer device (PR), to which you direct everything to be printed.
- Two minifloppy drives (MFP0 and MFP1). DIR in the TYPE field indicates a directory oriented device. VOLN shows the name of the volume. OFFL in the STAT field of FPY1 means that FPY1 has not been opened yet.
- Two terminal devices (TRM0 and TRM1)
- A device called CON to which all output on the screen should be directed.
- A NULL device which accepts all input and does nothing with it, like a "waist paper basket". This can be useful sometimes.

CHECKING THE TASKS IN THE PRIMARY MEMORY

The next resort is to check the tasks that presently occupy the primary memory. This is simply done with the command:

TASK

The output may look like this:

TASK	NR	STAT	TYPE	PROGRAM	PRI	TCB-ADR	SIZE	ENTRY
MTCM	1	W	RN		20			
COM0	2	W	EN		90			
UTL0	3		E		90			

- The task field shows the tasks. MTCM is the monitor, COM0 is the task that manages your terminal and UTL0 is the name that has been given the TASK utility while it is executed.

- The STAT field shows the status of the task at the time when the TASK utility runs. W means waiting, for an event.

- The TYPE field indicates the type of the task. It may be: E=executive, N=non-abortable, P=pure code and R=resident.

- The PRI field shoes the current priority assigned to the task

- The NR field show the task number of each task.

If you give the command EXA 3F00,100 you can see the bit map corresponding to the primary memory. Each byte corresponds to a byte in the primary memory. FF in the beginning of the bit map means the area is occypied by the OS.8MT. FF in the end of the map means that that part of the primary memory does not exist. (This is dependent on the amount of memory boards the machine is configured with). Other numbers indicate that an area of the primary memory is "owned" by a task. The number show which task.

CHECKING THE CONTENT OF A VOLUME

Handling different kinds of external memory is an important part of the operators duties. The command:

Library

is used to display the content of the system volume. See the OPERATORS MANUAL for a complete description of the options of this command.

If you want to introduce a new disc to the system the disc must be opened. This is done with the command:

OPEn devname

If you, for example, want to open the second floppy disk (after a diskette as been put in the drive) give the command:

OPEn MFP1:

The system responds with the name of the volume. You can check with the DEVICES command that MFP0 has been opened. The command:

Library volname

where volname is the name of the volume, will display the content of it.

CHANGING THE SYSTEM VOLUME

OS.8MT has always a system volume which is the default volume in the file descriptors. The system volume can be changed with the command:

Volume volname

Where volname is the name of the new volume.

PREPARING DISCS

You want to use a "fresh" disc with OS.8MT it must be formatted and initialized.
The first step is to put the magnetic structure on the disc.
This is done with the utility DISKFORM. Just give the command:

DISKFORM

A formatted disc can be initialized. This will put an empty library on the disc.
Use the command:

DISKINIT

And the required commands will be shown.

COPYING DISCS

Several copy utilitys is avialable with OS.8MT. The most important are: COPYLIB, COPYA and COPYI.

COPYLIB - COPY AND DELETE UNDER DIRECTORY CONTROL

COPYLIB copies and/or deletes files under directory control. You can either use copylib in direct mode or interactive mode.

Example: COPYLIB VOL1:,VOL2 will present a list of the files on the volume VOL1 and you have the option to copy any of the files to the volume VOL2 and you may give the files new names on the new disc, or keep the old name.

Example: COPYLIB VOL1:YOURLIB,VOL1:MYLIB will present a list of the files in the EFD "YOURLIB" to be copied to the EFD "MYLIB".

The swich "G" which indicates that the copying process should be done immediatly without presenting a list of the files on the source volume.

Example: COPYLIB,G VOL1:,VOL2: copies the content of the volume VOL1 to the volume VOL2.

The swich "D" indicates that the COPYLIB utility is used to delete files.

Example: COPYLIB,D VOL1: will present a list of the available files on the volume VOL1 which may be deleted.

A good way to gain confidence using the COPYLIB utility is to try all the variations of COPYLIB while checking with the LIBRARY command if the result is the expected.

COPYI - IMAGE COPY

COPYI makes image (exact) copys of files and volumes. If the source file is continous, the destination file will also be continous.

Example: COPYI VOL1:,VOL2 makes the volume VOL1 an exact copy of VOL2

COPYA - ASCII COPY

COPYA, which copies ASCII data between files or devices.

Example: COPYA VOL1:TEST,PR: copies the content of the file "TEST" on the volume "VOL1" to the printer:

Example: COPYA VOL1:TEST,CON: copies the content of the file "test" on the volume "VOL1" to the terminal.

Example: COPYA VOL1:TEST,VOL2:TEST2 copies the content of the ASCII file "TEST" on the volume "VOL1" to the volume "VOL2" giving it the new name "TEST2"

Several swiches is available. One useful swich is "A" (append). If you append the sourcefile to the destination, the resulting file will consist of the destination followed by the sourcefile.

COMMAND FILES

Routine work like formatting and copying files can sometimes be tedious work if done repetivly. A good idea is to use command files when such work is to be done.
A command file is an ASCCI file containing the commands commands you want to give the system.
If you for example want to format and initialize a disk and then make it a copy of another disc present with the system this is how it is done.

1. Use the editor to create an ASCCI file with one command on each line.

2. Enter the command !CMDFILE, where CMDFILE is the name of your ASCCI commandfile.

An example of such a command file is:

```
1.    DISKFORM DEVICE=M4,DRIVE=MPF0:
2.    DISKINIT DEVICE=M4,DRIVE=MPF0:,VOLUME=OLLE:,CLEAR
3.    COPYLIB,GV,30000 MAST:,OLLE:
```

When the command file is executed the mini floppy disk on drive 1 will be formatted, initialized and given the name OLLE. The content of the volume MAST will then be copied to OLLE.

COMMAND FILES WITH PARAMETERS

You can also give parameters while executing a command file.

1. Create a command file using the characters é1 - é9 instead of the parameters.

2. Execute the command file while including the parameters after the name of the command file. é1 will be substituted for the first parameter é2 for the second etc.

This is best shown with an example:

The command file:

```
1. DISKFORM DEVICE=é1,DRIVE=é2
2. DISKINIT DEVICE=é1,DRIVE=é2,VOLUME=é3,CLEAR
3. COPYLIB,GV,30000 é4,é3
```

And the command:

```
!CMDFILE M4,FPY1:,OLLE,MAST:
```

Will derive the same result as the previous example.

COMMENTS IN COMMAND FILES

A line beginning with a "*" character is interpreted as a command and will not be executed, although shown on the screen. A line beginning with a "#" character will not be shown on the screen while executed. This affects commands as well as comments.

AUTOSTART

A useful feature in the DataBoard system is autostart which means that for each terminal a task can be loaded and started at system start up time, like an application program. A command file may also be executed.
The AUTOSTART utility uses an Efd called AUTO which holds a number of files (TRM0,TRM1,TRM2 etc.), one for each terminal. The files are regular command files to be executed at system start-up.

```
************
*APPENDIX A*
************
```

This appendix contain a number of examples in BASIC and Assembler.

A BASIC SVC EXAMPLE USING FUNCTIONS

```
10 !
20 !
30 ! This is a set of useful SVC-functions You may use in Your
40 ! own BASIC-programs. The calling sequence is more or less
50 ! self-explanatory, but more detailed information will found
60 ! in the OS.8 manuals.
70 !
80 INTEGER : EXTEND
90 !
100 DIM Svcblk%(7%),Fd¤=33%
140 !
150 !
160 ! LOGG MESSAGE WITH TIME ON CONSOLE
170 ! ==================================
171 !
180 DEF FNSvc22logg%(Message¤)
190   Msg¤=Message¤
191   Msg%=LEN(Msg¤)
192 !
200   Svcblk%(0%)=2%
201   Svcblk%(1%)=2%
202   Svcblk%(2%)=VARPTR(Msg¤)
203   Svcblk%(3%)=Msg%
204 !
210   SVC 2%,Svcblk%
220   RETURN FNSOrs%
230 FNEND
240 !
250 !
260 ! CONVERT A HUMAN-FILENAME TO OS.8-FILEDESCRIPTOR
265 ! ===============================================
270 !
280 DEF FNSvc23pack¤(Filename¤)
290   F¤=Filename¤+CHR¤(0%)
295   Fd¤=CHR¤(0%)+SPACE¤(28%)
296 !
300   Svcblk%(0%)=9%
301   Svcblk%(1%)=3%
302   Svcblk%(2%)=VARPTR(F¤)
303   Svcblk%(3%)=VARPTR(Fd¤)+1%
304 !
310   SVC 2%,Svcblk%
320   RETURN RIGHT¤(Fd¤,2%)+LEFT¤(Fd¤,1%)
330 FNEND
340 !
```

```
350 !
360 ! CONVERT AN OS.8-FILEDESCRIPTOR TO HUMAN-FILENAME
365 ! ==================================================
370 !
380 DEF FNPackback¤(Filedescriptor¤)
390   Filetype¤="UALOBTI789abcdeD"
400   Filelang¤=" ABCFP6789abcEeM"
410   Fd¤=FNField¤(1%,4%,Filedescriptor¤,":")+FNField¤(5%,12%,Filedescriptor¤,".")
420   IF RIGHT¤(Fd¤,LEN(Fd¤))=":" GOTO 470
430   Fd¤=Fd¤+"/"+FNField¤(17%,12%,Filedescriptor¤,"")
440   Filetype%=ASCII(MID¤(Filedescriptor¤,29%,1%))
450   Fd¤=Fd¤+MID¤(Filetype¤,((Filetype%/16%) AND 15%)+1%,1%)
460   Fd¤=Fd¤+MID¤(Filelang¤,(Filetype% AND 15%)+1%,1%)
470   RETURN Fd¤
480 FNEND
490 !
500 DEF FNField¤(Start%,Length%,Filedescriptor¤,Delimiter¤)
510   F¤=""
520   FOR Dummy%=Start% TO Length%+Start%-1%
530     IF Dummy%>LEN(Filedescriptor¤) GOTO 600
540     C¤=MID¤(Filedescriptor¤,Dummy%,1%)
550     IF (Dummy%=Start% AND C¤=" ") GOTO 600
560     IF C¤=" " GOTO 590
570     F¤=F¤+C¤
580   NEXT Dummy%
590   F¤=F¤+Delimiter¤
600   RETURN F¤
610 FNEND
620 !
630 !
640 ! SET THE GLOBAL SYSTEM TIME-SLICE
650 ! =================================
655 !
660 DEF FNSvc27setslice%(Value%)
665 !
670   Svcblk%(0%)=18%
671   Svcblk%(1%)=7%
672   Svcblk%(2%)=Slice%
673 !
674   SVC 2%,Svcblk%
680   RETURN FNS0rs%
690 FNEND
700 !
710 !
720 ! PICK-UP THE CURRENT GLOBAL TIME-SLICE VALUE
730 ! ============================================
735 !
740 DEF FNSvc27getslice%
745 !
750   Svcblk%(0%)=17%
751   Svcblk%(1%)=7%
752   SVC 2%,Svcblk%
753 !
760   RETURN Svcblk%(2%)
```

```
770 FNEND
780 !
790 !
800 ! DECODE A COMMAND IN OS.8-MANNER
810 ! ================================
815 !
820 DEF FNSvc28decode%(Command¤,Line¤)
830   Cmd¤=Command¤ : Fd¤=Line¤+CHR¤(0%)
835   !
840   Svcblk%(1%)=8%
843   Svcblk%(2%)=VARPTR(Fd¤)
845   Svcblk%(3%)=VARPTR(Cmd¤)
847   !
850   SVC 2%,Svcblk%
860   RETURN (SWAP%(Svcblk%(1%)) AND 255%)+1%
870 FNEND
880 !
890 !
900 ! OPEN A DEVICE
910 ! =============
915 !
920 DEF FNSvc212open¤(Device¤)
930   Dummy%=FNSvc212close%(Device¤)
940   Fd¤=FNSvc23pack¤(Device¤)
945   !
950   Svcblk%(0%)=2%
951   Svcblk%(1%)=12%
952   Svcblk%(2%)=VARPTR(Fd¤)
953   !
960   SVC 2%,Svcblk%
970   RETURN FNPackback¤(MID¤(Fd¤,5%,24%)+SPACE¤(4%)+RIGHT¤(Fd¤,29%))
980 FNEND
990 !
1000 !
1010 ! CLOSE A DEVICE
1020 ! ==============
1025 !
1030 DEF FNSvc212close%(Device¤)
1040   Fd¤=FNSvc23pack¤(Device¤)
1045   !
1050   Svcblk%(0%)=1%
1051   Svcblk%(1%)=12%
1052   Svcblk%(2%)=VARPTR(Fd¤)
1053 !
1060   SVC 2%,Svcblk%
1070   RETURN FNS0rs%
1080 FNEND
1090 !
1100 !
1110 ! DELAY THE EXECUTION
1120 ! ===================
1125 !
1130 DEF FNSvc3delay%(Milliseconds%)
1136 !
```

```
1140   Svcblk%(0%)=1%
1141   Svcblk%(1%)=Milliseconds%
1142 !
1143   SVC 3%,Svcblk%
1150   RETURN FNS0rs%
1160 FNEND
1170 !
1180 !
1190 ! LOAD AN OVERLAY (MACHINE-CODED)
1200 ! ================================
1215 !
1210 DEF FNSvc5load%(Overlay¤,Address%)
1220   Fd¤=FNSvc23pack¤(Overlay¤)
1225 !
1230   Svcblk%(0%)=9%
1231   Svcblk%(2%)=0%
1232   Svcblk%(3%)=Address%
1233   Svcblk%(5%)=VARPTR(Fd¤)
1234 !
1240   SVC 5%,Svcblk%
1250   RETURN FNS0rs%
1260 FNEND
1270 !
1280 !
1290 ! LOAD A TASK
1300 ! ===========
1310 !
1310 DEF FNSvc6load%(Task¤,Extramemory%)
1320   Fd¤=FNSvc23pack¤(Task¤)
1325 !
1330   Svcblk%(0%)=1%
1333   Svcblk%(1%)=0%
1334   Svcblk%(2%)=VARPTR(Fd¤)+4%
1340   Svcblk%(5%)=VARPTR(Fd¤)
1345   Svcblk%(6%)=Extramemory%
1346 !
1350   SVC 6%,Svcblk%
1360   RETURN FNS0rs%
1370 FNEND
1380 !
1390 !
1400 ! START A DORMANT TASK
1410 ! ===================
1415 !
1420 DEF FNSvc6start%(Task¤,Switch1%,Switch2%,Parameter¤)
1430   Fd¤=FNSvc23pack¤(Task¤)
1440   Pd¤=CVT%¤(LEN(Parameter¤))+Parameter¤+CHR¤(0%)
1445 !
1450   Svcblk%(0%)=2%
1451   Svcblk%(1%)=0%
1452   Svcblk%(2%)=VARPTR(Fd¤)+4%
1460   Svcblk%(3%)=VARPTR(Pd¤)
1464   Svcblk%(4%)=0%
1465 !
```

```
1470   SVC 6%,Svcblk%,Switch1%,Switch2%
1480   RETURN FNS0rs%
1490 FNEND
1500 !
1510 !
1520 ! WAIT FOR A TASK TO TERMINATE
1530 !
1540 DEF FNSvc6wait%(Task¤)
1550   Fd¤=FNSvc23pack¤(Task¤)
1555   !
1560   Svcblk%(0%)=40%
1562   Svcblk%(2%)=VARPTR(Fd¤)+4%
1563   !
1564   SVC 6%,Svcblk%
1570   RETURN FNS0rs%
1580 FNEND
1590 !
1600 !
1610 ! LOAD & START A TASK, THEN WAIT FOR IT.
1620 !
1630 DEF FNSvc6run%(Task¤,Switch1%,Switch2%,Extramemory%,Parameter¤)
1640   Fd¤=FNSvc23pack¤(Task¤)
1650   Pd¤=CVT%¤(LEN(Parameter¤))+Parameter¤+CHR¤(0%)
1655   !
1660   Svcblk%(0%)=3%
1661   Svcblk%(1%)=0%
1662   Svcblk%(2%)=VARPTR(Fd¤)+4%
1664   Svcblk%(3%)=VARPTR(Pd¤)
1670   Svcblk%(4%)=0%
1672   Svcblk%(5%)=VARPTR(Fd¤)
1675   Svcblk%(6%)=Extramemory%
1677   !
1680   SVC 6%,Svcblk%,Switch1%,Switch2%
1690   Svcblk%(0%)=40% : SVC 6%,Svcblk%
1700   RETURN FNS0rs%
1710 FNEND
1870 !
1880 !
1890 ! FEED-BACK RETURN STATUS
1900 !
1910 DEF FNS0rs%
1920   S0rs%=SWAP%(Svcblk%(0%)) AND 255%
1930   RETURN S0rs%
1940 FNEND
```

****
APPENDIX B
****

This appendix contain an exampel of task communication using event driven tasks. The first task sends data to the other task. A command file is included to execute the example.

TASK TO SEND DATA

```
10 ! SAVE GSEVENT.EVENTT
20 !
30 ! WRITE TO OTHER TASK
40 ! ==================
50 !
60 ! 83-11-14 / Greger Sernemar / DataSweden AB
70 !
80 INTEGER : EXTEND
90 !
100 !
110 DIM Parblk%(7%) ! Size of an svc block. (in words)
120 !
130 Lu%=1% ! Logical unit
140 !
150 OPEN "EVEN:" AS FILE Lu% ! EVEN = task to write to.
160 !
170 ! ******************************************
180 ! FNSvclwrit : WRITE DATA TO OTHER TASK
190 ! ===================================
200 !
210 ! At call :
220 ! B% = Adress to buffer containing data to send. (i.e VARPTR(B%(0%)))
230 ! L% = Logical unit to write to.
240 ! Z% = B% size.
250 !
260 ! At return : Return status from SVC 1
270 !
280 DEF FNSvclwrit%(L%,B%,Z%) LOCAL L%,B%,Z%
290   Parblk%(0%)=2% ! Write request.
300   Parblk%(1%)=L% ! Logical unit
310   Parblk%(2%)=B% ! Address in my segment
320   Parblk%(3%)=Z% ! Buffer size
330   Parblk%(4%)=0% ! Bute count at completion.
340   !
350   SVC 1%,Parblk%
360   !
370   RETURN FNS0rs% ! Return return-status.
380 FNEND
390 !
400 ! ******************************************
410 ! FNS0rs : FEED-BACK RETURN STATUS
420 ! ==============================
430 !
440 DEF FNS0rs%
```

```
450   S0rs%=SWAP%(Parblk%(0%)) AND 255%
460   RETURN S0rs%
470 FNEND
480 !
490 !
500 Bad%(0%)=10%
510 Bad%(1%)=20%
520 Bad%(2%)=30%
530 Bsz%=6%
540 !
550 ; "Data sent :"
560 FOR I%=0 TO (Bsz%-(Bsz%/2%))-1
570   ; "BAD%(" I% ")=" Bad%(I%)
580 NEXT I%
590 !
600 Status%=FNSvclwrit%(Lu%,VARPTR(Bad%(0%)),Bsz%)
610 !
620 ; "Sending program terminated !"
630 BYE
```

TASK TO RECEIVE DATA

```
10 ! SAVE GSEVENT.EVENTR
20 !
30 ! GET DATA FROM OTHER TASK IN OS 8, USING SVC 2.6 (OS 8 Rev. 4.13 or later)
40 ! ===========================================================================
50 !
60 ! 83-11-14 / Greger Sernemar / DataSweden AB
70 !
80 INTEGER : EXTEND
90 !
100 ! DEFINE SOME CONSTANTS
110 !
120 Nodseg%=5% ! Offset to segment number address in node.
130 Nodsvc%=6% ! Offset to svc address in node.
140 Slbad%=2% ! Offset to buffer address filed in svc 1 block.
150 Slbsz%=3% ! Offset to buffer size filed in svc 1 block.
160 Slstr%=7% ! Max length of an svc block. (in words)
170 !
180 ! OS 8 SVC-FUNCTION CODES
190 !
200 S2f6get%=1% ! Transfer from other segment to me.
210 S6fqwai%=9% ! SVC 6 function code - waite for event.
220 S6fqtrm%=10% ! SVC 6 funtion code - terminate event.
230 !
240 ! DIMENSION SOME BUFFERS
250 !
260 DIM Parblk%(Slstr%) ! Parameter block for SVC call.
270 !
280 ! FUNCTION DEFINITIONS
290 !
```

```
300 ! ****************************************
310 ! FNGet : GET DATA FROM OTHER TASK.
320 ! ==================================
330 !
340 ! At call : B% = Address to buffer to get data in. (i.e VARPTR(B%(0%)))
350 !
360 ! At return : Returns the number of bytes recveied.
370 !
380 DEF FNGet%(B%) LOCAL T%,B%
390   Dummy%=FNSvc26get%(VARPTR(T%(0%)),FNSvc6qwait%,Seg%,Slstr%)
400   Size%=FNSvc26get%(B%,PEEK2(VARPTR(T%(Slbad%))),Seg%,PEEK2(VARPTR(T%(Slbsz%))
410   RETURN Size%
420 FNEND
430 !
440 ! *******************************************
450 ! FNSvc26get : GET DATA FROM OTHER SEGMENT
460 ! ========================================
470 !
480 ! At call : M% = Address to buffer where to put data from caller.
490 !
500 ! At return : Return the number of bytes recveied.
510 !
520 DEF FNSvc26get%(M%,C%,S%,B%) LOCAL M%,C%,S%,B%
530   Parblk%(0%)=S2f6get% ! Transfer data from other segment.
540   Parblk%(1%)=256%*S%+6% ! S%= Segment address of caller.
550   Parblk%(2%)=M% ! Address in my segment
560   Parblk%(3%)=B% ! Buffer size
570   Parblk%(4%)=C% ! Caller address
580   !
590   SVC 2%,Parblk%
600   !
610   RETURN Parblk%(3%)
620 FNEND
630 !
640 ! ***************************************
650 ! FNSvc6qwait : WAIT FOR EVENT
660 ! ===========================
670 !
680 ! At call : Nothing.
690 !
700 ! At return : The function returns the address to the svc block that
710 ! caused the event.
720 ! Seg% holds the segment address of the caller.
730 !
740 DEF FNSvc6qwait%
750   RETURN FNSvc6%(S6fqwai%,0%)
760 FNEND
770 !
```

```
780 ! ************************************
790 ! FNSvc6qtrm : TERMINATE EVENT.
800 ! ============================
810 !
820 DEF FNSvc6qtrm%(P%)
830   Dummy%=FNSvc6%(S6fqtrm%,P%)
840   RETURN FNS0rs% ! Return-status.
850   !
860 FNEND
870 !
880 ! ************************************
890 ! FNSvc6 : DO SVC 6
900 ! ================
910 !
920 ! At call : F% = Function code to svc 6.
930 ! P% = Parameter to svc 6. (for S6fqtrm, this is s6.par from S6fqwai )
940 !
950 ! At return : The function returns the address of caller data.
960 ! In the global variable Seg% is the segment address for caller
970 ! In the global variable is the paramater from SVC 6 waite for event.
980 !
990 DEF FNSvc6%(F%,P%) LOCAL F%,P%
1000   Parblk%(0%)=F% ! Function code.
1010   Parblk%(1%)=0% ! S6.PRIO, S6.OPT
1020   Parblk%(2%)=0% ! S6.TID
1030   Parblk%(3%)=P% ! S6.PAR
1040   !
1050   SVC 6%,Parblk% ! Execute svc 6
1060   !
1070   S6par%=Parblk%(3%)
1080   Seg%=PEEK(Parblk%(3%)+Nodseg%) ! Segment address of caller.
1090   RETURN PEEK2(Parblk%(3%)+Nodsvc%) ! Return svc address of caller.
1100 FNEND
1110 !
1120 ! *******************************************
1130 ! FNS0rs : FEED-BACK RETURN STATUS
1140 ! ===============================
1150 !
1160 DEF FNS0rs%
1170   S0rs%=SWAP%(Parblk%(0%)) AND 255%
1180   RETURN S0rs%
1190 FNEND
1200 !
1210 ! MAIN PROGRAM
1220 !
1230 DIM Buffer%(100%) ! Buffer in my program that shall get data.
1240 !
1250 Buffz%=FNGet%(VARPTR(Buffer%(0%)))
1260 !
1270 ; : ; "Number of bytes recieved = " Buffz%
1280 ; "Data recieved :"
1290 FOR I%=0 TO (Buffz%-(Buffz%/2%))-1
1300   ; "BUFFER%(" I% ")=" Buffer%(I%)
1310 NEXT I%
```

```
1320 !
1330 ! Terminate sending program
1340 !
1350 Status%=FNSvc6qtrm%(S6par%) ! Use the global parameter S6par%
```

COMMAND FILE TO START THE EXAMPLE

```
LOAD BASIC,EVEN
ST EVEN GSEVENT.EVENTR
BASIC GSEVENT.EVENTT
```

```
*************
*APPENDIX C*
*************
```

This appendix contain an example of a simple device driver which is established at run time. The link stream for the device driver is also included. For a complete documentation on how to write driver routines see OS.8MT Programmers Manual.

```
EST¤EXMPL PROG  ** ESTABLISH DRIVER AT RUN-TIME **
*
*
* This is a little example of how to establish a driver and
* device in runtime under OS8MT !
* The driver is very simple, can just send characters !
*
*
          PLC   0                   THIS PART MAY BE IN A-SEGMENT
*
START     EQU   *
          SVC   2,HELLO             HELLO THERE !
          SVC   8,DEVICE            DO THE ESTABLISH
SOME.ERR  EQU   *
          SVC   6,PAUSE             THEN JUST SLEEP
*
*  If we wake up, lets try to remove it
*
          LI    A,S8F.RMOV
          ST    A,DEVICE+S0.FC
          SVC   8,DEVICE            REMOVE
          JNCS  CANCEL              GOOD WORK, GO TO CANCEL
*
          SVC   2,BADREM            DIDN'T WORK, MAYBE ASSIGNED
          JMPS  SOME.ERR
*
CANCEL    EQU   *
          LI    A,S0F.CAN
          ST    A,PAUSE+S0.FC
          SVC   6,PAUSE             BYE, BYE
          JMPS  CANCEL              CATASTROPHICAL ERROR !!!!!
*
*
          PLC   1                   SVC-BLOCKS MUST BE I B-SEGMENT
*
HELLO     DB    S1F.IASC+S1F.WRIT,0,2,0
          DA    HELLOBUF,HELLOSIZ,0
*
HELLOBUF  DB    'Establish test-device',0
HELLOSIZ  EQU   *-HELLOBUF
*
BADREM    DB    S1F.IASC+S1F.WRIT,0,2,0
          DA    BADBUF,BADSIZ,0
*
BADBUF    DB    'Cant remove, maybe assigned?',0
```

```
BADSIZ   EQU    *-BADBUF
*
PAUSE    DB     S6F.PAUSE,0,0,0
         DA     0,0,0,0,0
*
DEVICE   EQU    *                   THIS IS A SVC8-BLOCK
         DB     S8F.EST,0           FC AND RS
         DB     0,0                 LET OS CHOOSE THEM
         DA     NAME                NAME POINTER
         DB     S8C.DEV             IT'S A DEVICE
         DB     RTT.RCB             AND IT'S EXLUSIVE
         DA     RDT.TST             POINTER TO RDT
         DB     4Q,5                CARD SELECT AND INTERUPT-LEVEL
*
NAME     DB     'TEST'              IT'S NAME
*
RDT.TST  DB     RCT.DCB             WE WANT A DCB
         DB     0                   NO DCB-EXTENSION
         DA     DRV.INIT            INITIATOR ADDRES
         DA     0                   NO TERMINATOR
*
* The RDT is folowed by a DDT .
*
         DA     ATR.WRIT            ONLY SUPPORTS WRITE
         DA     0                   VARIABLE RECORD LENGTH
         DB     0                   DEVICE CODE
         DB     DCT.ICB             MUST HAVE AN ICB
         DB     S1.STR              WAN'T THE WHOLE SVC1-BLOCK COPIED
*
* The DDT is followed by an IDT
*
         DB     ICT.CCB             WE NEED A CCB
         DA     0                   NO CONTINUATOR YET
*
* The IDT is followed by a CDT
*
         DA     0                   NO TIME-OUT HANDLER
         DA     5*10                TIME-OUT, (10=1 SECOND)
*
* The CDT is followed by an EDT
*
         DB     0,0                 NO EXTENSION
*
*
         PLC    0                   THE DRIVER MUST BE IN A-SEGMENT
*
*
```

```
*
*
*
**************************************
* Driver Initiator
**************************************
*
 *  At entry:   X -> DCB
 *              SVC - block in DCB
 *              Formatter address in DCB.FMTE
 *              Y -> SVC - block
 *
 *  On return:  X -> DCB
 *              Y -> SVC - block
 *              CY=0: Not complete
 *              CY=1: Complete
 *              A=Return status
 *
 DRV.INIT  EQU   *
           CALL  DATA.FMT
           JTCS  WRONGFC
           DECR  E
           JTZS  WRONGFC
           DECR  E
           JFZS  DONE
           CALL  CHECK.PNT
 *
 **************************************
 *DRIVER CONTINUATOR
 **************************************
 *
 DRV.CONT  EQU   *
           OR    A                  CLEAR CARRY BIT
           CALL  DATA.FMT           LOAD THE NEXT BYTE
           JTCS  COMPLETE
           OUT   DATA               SEND DATA TO DEVICE
           RET
 *
 * REQUEST COMPLETE
 *
 COMPLETE  EQU   *
           CALL  DATA.FMT           POSTPROCESS THROUGH DATA FORM.
 *
 DONE      EQU   *
           XR    A                  RETURN STATUS 0
           OUT   C4                 DISABLE INTERFACE INTERRUPT
           RBT   DCS.INT,DCB.STAT(X)DISABLE INTERRUPT POLLING
           STC                      MARK COMPLETE
           RET
```

```
*
************************************
* CALL THE DATA FORMATTER
************************************
*
DATA.FMT  EQU   *
          L     L,DCB.FMTE(X)
          L     H,DCB.FMTE+1(X)
          JDR   HL                    ENTER THE DATA FORMATTER
*
*
WRONGFC   EQU   *
          LI    A,SOS.IFC             TELL HIM
          STC                         AND WE ARE COMPLETE
          RET
```

```
*
**************************************
*ENABLE THE INTERFACE
**************************************
*
CHECK.PNT EQU   *
          POP   HL
          ST    L,ICB.CON(X)       CHECKPOINT CONTINUATOR
          ST    H,ICB.CON+1(X)
          MVI   2,CCB.TM(X)        LOAD STATUS TEST MASK
*                                 FOR ENABLE INTERUPT ON TRANSMITT
*                                 BUFFER EMPTY !
          LI    A,80H              SEND INTERRUPT ENABLE TO UART 4117
          DIS                      DISABLE CPU
          SBT   DCS.INT,DCB.STAT(X)INTERRUPT POLLING ALLOWED
          OUT   C4                 ENABLE THE INTERFACE
          ENI                      ENABLE THE CPU
*
          XR    A                  MARK NOT COMPLETE
          RET
*
*
          END   START
```

LINK STREAM FOR THE DRIVER

```
REMOTE
LOG CON:
NOLIST UNUSED
*
* LINK-STREAM FOR TEST-DEVICE
*
ABS       MUST BE !
PLCBASE   0FC00H          B-SEGMENT BASE
PLCORDER 0,1
OPTION PURE
OPTION EXCLUSIVE
PURENAM TSTE
SEGMENT 1024,0            A-SEGMENT SIZE 1024, AND ONLY PLC0 CODE
INC TESTEX
IMPURE
LIB,A OS4OBJ.M4DEFLB/
CHE
TASK XXX
END
```

```
************
*APPENDIX D*
************
```

This appendix contain a selectfile to the DataBoard linker ESTAB for a complete OS.8MT generation. The selectfile is included as an example only. If you have the intention of generating your own OS complete documentation is found in the OS.8MT Programmers Manual.

AN OS.8MT GENERATION

```
*
*
*   THIS IS A COMPLETE OS8-4.13 - GENERATION
*   ========================================
*
*   CUSTOMER:
*   DATE    :
*   ATHOUR  :
*   ID      :
*
REMOTE                              ABORT IN CASE OF ERROR !
LOG CON:                            TELL WHAT'S HAPPENING.
*
*
*   MEMORY STRUCTURE
*
ABSOLUTE                            GIVES A BOOT MODULE.
AUTORST
PLCLIST 0,2
PLCORDER 1,2,6,3,5,30               DON'T CHANGE THE ORDER !
PLCSTART                            GENERATES THE SYMBOLS ¤¤PLCS..
PLCEND                              GENERATES THE SYMBOLS ¤¤PLCE..
*
*
*   LIST HANDLING
*
NOLIST ABSOLUT                      SUPRESS ABSOLUTE VALUES.
NOLIST UNUSED                       SUPRESS UNUSED ENTRIES.
*
*
*   SYSTEM GENERATION CONSTANTS
*
EQU 83,SGN.YEAR                     GENERATION DATE.
EQU 06,SGN.MNTH
EQU 16,SGN.DAY
EQU 48,SGN.MNOD                     NUMBER OF SYSTEM NODES.
EQU 30,SGN.MFCB                     NUMBER OF DEFAULT FCB'S.
EQU 4,SGN.FMGB                      NUMBER OF FILE-MANAGER BUFFERS.
*
*
*   OS GENERATION OPTIONS
```

```
*
*
*                                         TO PRODUCE A MAPPED OS 8 FOR THE
*                                         THE NEW (4 MHz) CPU SELECT RST.MAP
*
*                                         IF YOU WANT TO HAVE A MAPPED OS 8
*                                         FOR THE OLD (2.5 MHz) CPU
*                                         SELECT RST.OMAP  (Old MAP)
*
SELECT RST.OMAP                           THIS COMMAND WILL PRODUCE A MAPPED
*                                         OS 8 FOR THE OLD (2.5 MHz) CPU.
*
*
*
*
*
*   OPTIONAL NUCLEUS
*
SELECT NMI.HAND                           NMI-HANDLER.
SELECT ILL.HINT                           ILLEGAL HARDWARE INTERRUPT HANDLER.
SELECT ERR.HAND                           SYSTEM ERROR HANDLER.
SELECT FILE.MGR                           FILE-MANAGER, PHYSICAL ACCESS.
*
*
*   CRASH PRINT-OUT FACILITIES
*
SELECT CRS.DUMP                           CRASH DUMP...
EQU 75Q,CS.CRASH                          ...ON SYSTEM CONSOLE.
EQU 2,TM.CRASH                            2 = 4117, 4 = 4110
EQU 0FFH,XM.CRASH                         FF = 4117, FA = 4110
*
*
*   SVC-FUNCTIONS
*
SELECT SVC1,SVC3,SVC4,SVC5,SVC6,SVC7,SVC8
SELECT SVC2.1,SVC2.3,SVC2.4,SVC2.5,SVC2.6,SVC2.7,SVC2.8,SVC2.12
*
*
*   VOLUMES
*
SELECT DEV.FPY                            INTERFACE 4034, FLOPPY 8", 256KB.
SELECT DEV.BFPY                           INTERFACE 4108, FLOPPY 8", 1MB.
SELECT DEV.MFPY                           INTERFACE 4076, FLOPPY 5", 80KB.
SELECT DEV.BMFP                           INTERFACE 4106, FLOPPY 5", 320KB.
SELECT DEV.MH10                           INTERFACE 4610, HAWK 14", 5MB.
SELECT DEV.MM20                           INTERFACE 4109, MARKSMAN 14", 20MB.
SELECT DEV.XBC                            INTERFACE 4105, SEAGATE 5", 5MB.
SELECT DEV.DRM                            INTERFACE 4034, DRUM, 512KB.
*
*
*   DEVICES
*
SELECT DEV.NULL                           DUMMY DEVICE.
SELECT DEV.RE                             INTERFACE 4016, FACIT SP1-REC.
```

```
SELECT DEV.PU                          INTERFACE 4015, FACIT SP1-XMT.
SELECT DEV.PR                          INTERFACE 4015/4017/4117, PRINTER.
SELECT DEV.CR                          INTERFACE 4037, CDC CARD-READER.
SELECT DEV.IEC                         INTERFACE 4025, INSTRUMENT-BUS.
SELECT DEV.MAG                         INTERFACE 4104, 9-TRACK MAG-TAPE.
SELECT DEV.CTU                         INTERFACE 4015/4016, FACIT CASSETTE.
SELECT DEV.GRAF                        INTERFACE 4017/4117, DATACOLOUR.
*
*
*   CONFIGURATE MTM-TERMINALS
*
SELECT MTM.ADM3                        GIVES TWO ADM3-TERMINALS.
DATE                                   GIVES CURRENT DATE FOR SIGN-ON.
*
*
*   COLLECT MODULES FOR EACH SEGMENT
*
SEGMENT 1024                           ---------- A-SEGMENT 0 -------------
INC -:OBJ413LIB.RST413LIB/O.SEG¤HDR
LIB -:OBJ413LIB.RST413LIB/O
LIB -:OBJ413LIB.BAS413LIB/O
SEGMENT                                ---------- A-SEGMENT 1 -------------
INC -:OBJ413LIB.RST413LIB/O.SEG¤HDR
LIB -:OBJ413LIB.SVC413LIB/O
SEGMENT                                ---------- A-SEGMENT 2 -------------
INC -:OBJ413LIB.RST413LIB/O.SEG¤HDR
LIB -:OBJ413LIB.CON3XXLIB/O
LIB -:OBJ413LIB.CON413LIB/O
LIB -:OBJ413LIB.MTM413LIB/O
SEGMENT                                ---------- A-SEGMENT 3 -------------
INC -:OBJ413LIB.RST413LIB/O.SEG¤HDR
LIB -:OBJ413LIB.DEV3XXLIB/O
LIB -:OBJ413LIB.DRV413LIB/O
LIB -:OBJ413LIB.DRV3XXLIB/O
LIB -:OBJ413LIB.FMG413LIB/O.SV2¤10CM,.SV2¤11DM,.FMG¤OPCL
SEGMENT                                ---------- A-SEGMENT 4 -------------
INC -:OBJ413LIB.RST413LIB/O.SEG¤HDR
LIB -:OBJ413LIB.FMG413LIB/O
SEGMENT
LIB,A -:OBJ413LIB.DEF413LIB/O
*
*
*   FINISH-UP PHASE
*
CEXT ER0.PRES                          DEFINE NOT SELECTED HANDLERS AS NOT PRESENT
EQU ER0.PRES,FMG.CHKP
CHECK
END
```