

---

RCSL No: 43-GL9546 pp. 205

Edition: October 1979

Author: Marie Louise Møller  
Dan Andersen  
Knud Henningsen  
Niels Adler-Nissen

---

Title:

MUS SYSTEM  
Programming Guide  
Rev. 1.00

---

---

**Keywords:**

Multiprogramming, monitor, device handling, input/output, catalog system.

---

**Abstract:**

This manual is intended to function as a programming guide to the Multiprogramming Utility System for the RC3600 line of computers.

(210 printed pages)

---

Copyright © 1979, A/S Regnecentralen af 1979  
RC Computer A/S  
Printed by A/S Regnecentralen af 1979, Copenhagen

Users of this manual are cautioned that the specifications contained herein are subject to change by RC at any time without prior notice. RC is not responsible for typographical or arithmetic errors which may appear in this manual and shall not be responsible for any damages caused by reliance on any of the materials presented.

TABLE OF CONTENTS	PAGE
1. INTRODUCTION .....	1
2. BASIC CONCEPTS .....	2
2.1 Processes .....	4
2.2 Programs .....	8
2.3 Messages - Answers .....	9
2.4 Initialization of the Descriptors .....	11
3. MONITOR .....	15
3.1 Monitor Descriptor .....	15
3.1.1 Running Queue .....	17
3.1.2 Process Chain .....	18
3.1.3 Device Table .....	19
3.1.4 Delay Queue .....	21
3.1.5 Program Chain .....	22
3.1.6 Free Core Area .....	22
3.1.7 The Dummy Process .....	22
3.1.8 Area Process Chain .....	23
3.1.9 Interrupt Mask .....	23
3.1.10 Real Time Clock .....	23
3.1.11 Power Failure .....	24
3.2 Monitor Functions .....	24
3.2.1 Function WAITINTERRUPT .....	24
3.2.2 Function SENDMESSAGE .....	26
3.2.3 Function WAITANSWER .....	30
3.2.4 Function WAITEVENT .....	32
3.2.5 Function WAIT .....	34
3.2.6 Function SENDANSWER .....	35
3.2.7 Function SEARCHITEM .....	37
3.2.8 Function BREAKPROCESS .....	40
3.2.9 Function CLEANPROCESS .....	42
3.2.10 Function STOPPROCESS .....	43

<u>TABLE OF CONTENTS (CONTINUED)</u>		<u>PAGE</u>
	3.2.11 Function STARTPROCESS .....	44
	3.2.12 Function RECHAIN .....	44
3.3	Initialization of the Monitor .....	47
3.4	Processor Expansion .....	48
4.	DRIVER PROCESSES .....	49
4.1	Introduction .....	49
4.2	Device Handling .....	50
4.3	Driver Interface .....	54
	4.3.1 Control Messages .....	55
	4.3.2 Transput Messages .....	59
	4.3.2.1 Input .....	60
	4.3.2.2 Output .....	60
	4.3.3 Answers .....	62
	4.3.3.1 Answer .....	62
4.4	System Utility Procedures .....	66
	4.4.1 Formats .....	66
	4.4.2 Procedures .....	67
	4.4.2.1 Procedure NEXTOPERATION ..	67
	4.4.2.2 Procedure WAITOPERATION ..	71
	4.4.2.3 Procedure RETURNANSWER ...	73
	4.4.2.4 Procedure SETRESERVATION .	74
	4.4.2.5 Procedure SETCONVERSION ..	75
	4.4.2.6 Procedure CONBYTE .....	75
	4.4.2.7 Procedure GETBYTE .....	76
	4.4.2.8 Procedure PUTBYTE .....	76
	4.4.2.9 Procedure MULTIPLY .....	78
	4.4.2.10 Procedure DIVIDE .....	79
	4.4.2.11 Procedure SETINTERRUPT ...	79
4.5	Driver Requirements .....	81
	4.5.1 Break Action .....	81
	4.5.2 Device handling .....	82
4.6	Driver Incarnations .....	83

<u>TABLE OF CONTENTS (CONTINUED)</u>	<u>PAGE</u>
5. BASIC I/O HANDLING .....	86
5.1 General Description .....	86
5.2 Zone Format .....	90
5.3 Share Descriptor Format .....	93
5.4 Zone Setup .....	96
5.4.1 Zone Descriptor .....	96
5.4.2 Share Descriptor .....	97
5.4.3 Message Buffer Pool Size .....	97
5.5 Document Identification .....	99
5.6 Exception Handling .....	101
5.6.1 User Giveup .....	102
5.6.2 Giveup Procedure Example .....	104
5.7 Repeat Actions .....	104
5.8 Basic I/O Procedures .....	107
5.8.1 Initialization Procedures .....	109
5.8.1.1 Procedure OPEN .....	109
5.8.1.2 Procedure SETPOSITION ....	110
5.8.1.3 Procedure WAITZONE .....	111
5.8.1.4 Procedure CLOSE .....	111
5.8.2 INPUT/OUTPUT Procedures .....	114
5.8.2.1 Procedure TRANSFER .....	114
5.8.2.2 Procedure WAITTRANSFER ...	115
5.8.2.3 Procedure INBLOCK .....	116
5.8.2.4 Procedure OUTBLOCK .....	118
6. RECORD I/O .....	120
6.1 Physical/Logical Datablock .....	120
6.2 Record-Formats .....	124
6.2.1 Undefined, Unblocked .....	125
6.2.2 Undefined, Blocked .....	126
6.2.3 Fixed, Unblocked .....	126
6.2.4 Fixed, Blocked .....	127
6.2.5 Variable, Unblocked .....	127

TABLE OF CONTENTS (CONTINUED)		PAGE
	6.2.6 Variable, Blocked .....	129
6.3	GETREC .....	129
	6.3.1 Procedure GETREC (Zone, Addr., Bytes)	129
	6.3.2 Pseudo-Algol Description .....	132
	6.3.3 Programming Example .....	137
6.4	PUTREC .....	138
	6.4.1 Procedure PUTREC (Zone, Addr., Bytes)	138
	6.4.2 Pseudo-Algol Description .....	141
	6.4.3 Programming Example .....	145
6.5	MOVE .....	146
	6.5.1 Programming Example .....	148
7.	CHARACTER I/O PROCEDURES .....	149
7.1	Single Character Procedures .....	149
	7.1.1 INCHAR .....	149
	7.1.2 OUTCHAR .....	150
	7.1.3 OUTSPACE .....	150
	7.1.4 OUTEND .....	151
	7.1.5 OUTNL .....	151
7.2	String Oriented Procedures .....	152
	7.2.1 OUTTEXT .....	152
	7.2.2 OUTOCTAL .....	152
	7.2.3 INNAME .....	153
7.3	Utility Procedures .....	153
	7.3.1 DECBIN .....	153
	7.3.2 BINDEC .....	154
7.4	Programming Examples .....	155
8.	CATALOG SYSTEM .....	156
8.1	Introduction .....	156
8.2	Catalog System Disc Structure .....	157
8.3	Catalog System Procedures .....	164
	8.3.1 General .....	164

TABLE OF CONTENTS (CONTINUED)		PAGE
	8.3.1.1	Main Catalogs ..... 164
	8.3.1.2	Sub Catalogs ..... 165
	8.3.1.3	Procedure CREATEENTRY .... 165
	8.3.1.4	Procedure REMOVEENTRY .... 167
	8.3.1.5	Procedure LOOKUPENTRY .... 168
	8.3.1.6	Procedure CHANGEENTRY .... 170
	8.3.1.7	Procedure SETENTRY ..... 171
	8.3.1.8	Procedure INITCAT ..... 174
	8.3.1.9	Procedure NEWCAT ..... 175
	8.3.1.10	Procedure FREECAT ..... 176
8.4	Catalog File INPUT/OUTPUT .....	178
	8.4.1	Procedure OPEN ..... 178
	8.4.2	Procedure SETPOSITION ..... 181
	8.4.3	Procedure CLOSE ..... 181
	8.4.4	Catalog Input/Output ..... 181

## APPENDIX

A.	REFERENCES .....	185
B.	TERMINOLOGY .....	187
C.	DEVICE CODES .....	191
D.	FIRST AND SECOND PAPER TAPE PUNCH DRIVERS .....	194

This page is intentionally left blank



FOREWORD

This publication is to supersede the one currently in use:

44-RT 1306: MUS SYSTEM INTRODUCTION

(part one of two)                      and

MUS PROGRAMMERS GUIDE

(part two of two)

August 1976

This page is intentionally left blank

1. INTRODUCTION.

1.

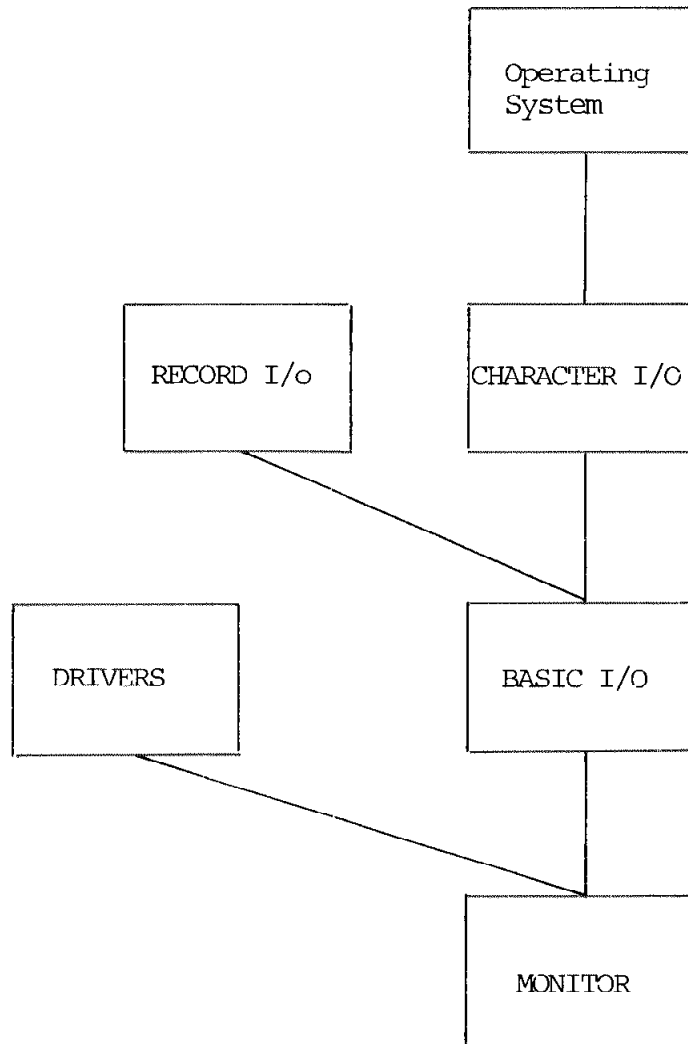
The Multiprogramming Utility System (MUS) for the RC3600 line of computers has the aim

- to simulate parallel processing including interprocess communication and interrupt processing.
- to give a strong framework for I/O processing, both on character level and on record oriented level.
- to support the user in the running of the system, which includes easy operator communication and an operating system that takes care of core administration and creation and removal of processes.

These goals have been reached by creation of the following software modules:

- a monitor, implementing multiprogramming.
- driver programs for common devices. These lay down the rules which are to be followed in coding drivers for new devices. These rules are purely a matter of overall cleanliness, and no real distinction is made between driver programs and other programs.
- reentrant I/O procedures designed around the zone concept, which has shown to be a clean and tidy way to describe device peculiarities, buffering, record formatting and - packing involved in any I/O activity.
- a basic operating system, which caters for program load and deletion, process creation and removal and start or stop of existing processes. The operating system is described in a separate manual.

The different modules may be put together in a hierarcial manner. Below is a figure showing the hierarchy. User processes may be build on top of all modules depending on what is needed to fit the usage.



2. BASIC CONCEPTS.

2.

As an introduction to a detailed description of the monitor this chapter deals with the three fundamental items: programs, processes and message buffers, handled by the monitor.

A program is a collection of instructions. The execution of a program in a given storage area is called a process. A process is identified by a unique process name, used in all references to it. Thus other processes need not be aware of the absolute location of the process in the storage, but can refer to it by name.

Processes can communicate by sending messages, carried in a message buffer to each other. After the processing of a message, the receiving process returns an answer to the sending process in the same message buffer.

The term event denotes a message or an answer. When events arrive to a process from other processes, they are linked to the event queue which is a part of the process description.

This chapter will apply concepts described later in this manual, so it is recommended at first only to read it to get an idea of the contents of program descriptors, process descriptors and message buffers and then turn back to it after reading about the monitor.

All parameter names mentioned in the following are defined as permanent symbols in the assembler and can thus be applied when coding assembler programs. The values are the corresponding displacements in the format descriptions.

Following the three descriptions is an example showing, how they should be initialized in an assembler program.

2.1 Processes.

2.1

Each process running under the MUS (DOMUS) monitor must have a process descriptor, where all the information about the process needed by the monitor to simulate multiprocessing is collected.

The start address of a process (specified after the terminating .END) should be the process descriptor address. In this way, the operating system is able to find the process descriptor, when loading a process.

The first part of a process descriptor must contain the following parameters, each consisting of one word if nothing else is specified.

PROC.NEXT:	Next process in a queue of processes (see PROC.PREV).
PROC.PREV:	Previous process in a queue of processes. Together with PROC.NEXT, this element is used to link the process to the running queue or to the delay queue. If the process is not in a queue, the parameters both point to the process itself.
PROC.CHAIN:	Next process in the process chain. All process descriptors are linked together in a chain.
PROC.CARRY	When interrupted the process uses bit 15 of this field as save location for the carry bit.
PROC.NAME:	Process name consisting of <u>three words</u> . The process is identified by this text of one to five characters; unused character positions must equal zero.
PROC.EVENT:	The event queue head, consisting of <u>two words</u> . The first word points to the first event in the event queue and the second word points to the last event in the event queue. If the queue is empty, both words point to

PROC.EVENT. The event queue contains messages and answers to the process.

PROC.BUFFE: Head of the message buffer chain which contains all message buffers belonging to the process.

PROC.PROG: Address of the descriptor of the program executed by the process.

PROC.STATE: This location indicates the actual state of the process. If the process is waiting and more than one reason is specified it is started again, when one of the conditions is fulfilled. Possible states of a process are:

0: running, i.e. the process is linked to running queue and wants to have time slices for execution, or:  
waiting for software timer, i.e. the process is linked to delay queue.

bit 0: stopped, i.e. the process is stopped.

3 to 63: waiting for interrupt or software timer. i.e. the process is waiting for an interrupt from the device with device no. = state. In addition, it is linked to the delay queue and waits for the software timer.

> 63: waiting for answer. The process waits for an answer to a message it has sent in the message buffer with address = state.

- 1: waiting for event. The process is started again, when a buffer is linked to its event queue, i.e. when either a message or an answer

arrives.

- 2: waiting for event or software timer.

As above, but if an event has not arrived when the timer runs out, the process is started with time-out.

-3 to -63:waiting for event, software timer or interrupt from device no =  
-state.

PROC.TIMER: Timer count. The number of timer periods the process still may wait in the delay queue. If the process is in the delay queue, this timer count is decremented for each timer interrupt and when it becomes zero, the process is started again.

PROC.PRIOR: Priority. Priorities are unsigned 16 bits integers and zero must not be used. The next process to use the CPU is chosen cyclically among the processes with highest priority. (See ch. 3.1 about the running queue).

PROC.BREAD: Break address.  
This address is entered after a break of the process, e.g. caused by a program error or an operator break.

PROC.AC0: Saved AC0. See below.

PROC.AC1: Saved AC1. See below.

PROC.AC2: Saved AC2. See below.

PROC.AC3: Saved AC3.

Before the process is interrupted the registers will be saved here and when the process becomes active again, these locations will be loaded into the registers such that the state of the processor is the same when the execution continues. (The carry bit is saved and reloaded from the proc.carry field).



After loading the process, the registers are loaded with the values of these parameters and the process is started in the address specified in PSW (see below).

- PROC.PSW: Program status word.  
 When the process is interrupted, the program counter is saved in proc.psw (0:15). When the process becomes active again, execution continues from the word address saved here. When initially loaded and started the contents of proc.psw (0:14) is used as a normal relocable start address for the process and must be specified as such. Proc.psw is then reorganized by the loader and proc.psw (15) is saved in proc.carry (15).
- PROC.SAVE: Work location for the system procedures.

Following these locations, the process descriptor may contain any number of optional words. E.g. a driver process using the driver utility procedures must contain six words directly following the before-mentioned:

- PROC.BUF: Saved message buffer address.  
 PROC.ADDRE: Current value of address (BUF.MESS2).  
 PROC.COUNT: Current value of count (BUF.MESS1).  
 PROC.RESER: Process descriptor address of reserving process. Zero indicates no reserver.  
 PROC.CONVT: Conversion table address. Zero indicates no conversion.  
 PROC.CLINT: Interrupt handling entry. This address is entered in disabled mode, when an interrupt arrives from a device which the process wants to supervise.

## 2.2 Programs.

The code executed by a process is a program. A program head contains information about the size and name of the program and a descriptor word.

PROG.PSPEC:           The program descriptor word describes the use of the program.

BIT 0:           Own bit. Set, if the program contains its own process descriptor after the program.

Bit 1:           Reentrant bit. Set, if the program is reentrant.

Bit 5:           Parameter bit. Set, if the program wants parameters transferred from DOMUS.

Bit 6:           Paged program bit. Set, if the program is paged.

Bit 7:           Reservation bit. Set in a reservable driver process.

PROG.PSTART:        Address of the first instruction in the program.

PROG.CHAIN:         Link to next program in the program chain. All program descriptors are linked together in a chain.

PROG.SIZE:          Size in words of the program.

PROG.NAME:          Program name in three words. The program is identified by this text of one to five characters. Unused character positions must equal zero.

Communication between processes is handled by four monitor functions, which enable the processes to exchange information in such a way that only one process at a time accesses the information. The information is carried in a message buffer (short: buffer) consisting of a head of six words and an information part of four words.

BUF.NEXT:           Next buffer in a queue of buffers (see below).

BUF.PREV:           Previous buffer in a queue of buffers.  
Together with BUF.NEXT this element is used to link the buffer to the event queue of a process. When the buffer is not in use, both elements point to the buffer itself.

BUF.CHAIN:          Next buffer in a chain of buffers. All message buffers belonging to a process are chained together.

BUF.SIZE:           Size of the buffer. At present the size equals ten.

BUF.SENDE:          Sender process descriptor address. This value is permanent as the process sends information in a message buffer belonging to itself.

BUF.RECEI:          Receiver parameter.  
The value of this parameter indicates the state of the buffer.

0:                  The buffer is free and can be used to send information.

> 0:                 The buffer contains a message sent to another process and  
BUF.RECEI = receiver process descriptor address.

< 0:                 The buffer contains an answer from a process that has received a mes-

sage in this buffer, and  
 BUF.RECEI = -(process descriptor  
 address of the receiver  
 of the message being  
 answered).

BUF.MESS0: BUF.MESS1: BUF.MESS2: BUF.MESS3:	}	Information words. Optional contents depending on the use. The standard usage in drivers is described in chapter 4.
--	---	--

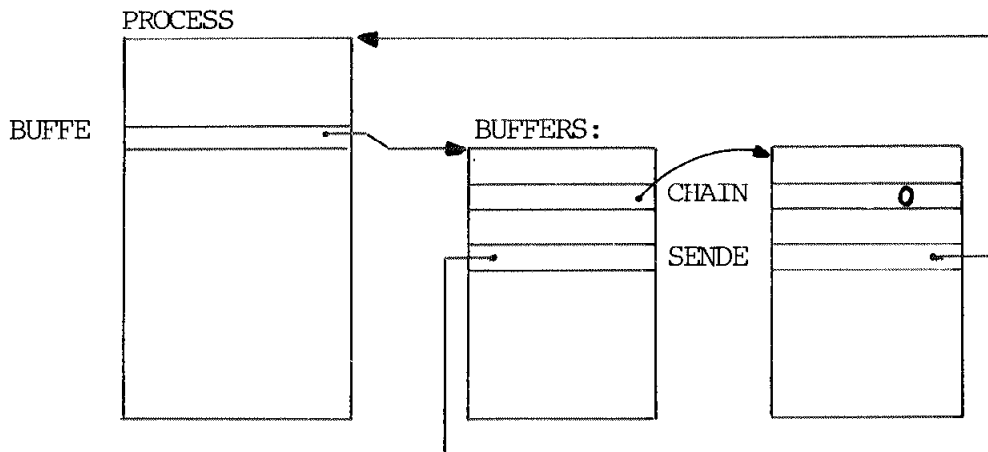


Fig. 1

If a process wants to send information to other processes, it must own a pool of buffers linked together in a chain with head in PROC.BUFFE (fig. 1). By means of the monitor function SENDMESSAGE the information is copied into the first free buffer within the pool and the buffer is linked to the event queue of a named receiver. The receiver gets information about the message by means of WAITEVENT. After processing of the message, the recei-

ving process returns an answer to the sending process in the same buffer, which now is linked to the event queue of the sender (= owner) process. This answering is handled by SENDANSWER. If the original sender gets information about the answer by means of WAITEVENT, the buffer has to be released by using WAITANSWER. If the sender wants to wait for an answer to a specific message, it suffices to use WAITANSWER.

For further details, see the description of the monitor functions, chapter 3.2.

#### 2.4 Initialization of the Descriptors.

2.4

Below is shown in an example, how a program descriptor and a process descriptor with two message buffers are coded. Parameters marked with \* must always be initialized by the programmer with values as indicated in the example. Unmarked parameters are set by the system during execution.

```

                                ;program descriptor:

PG1:    1B0+1B1                ;*this program is reentrant and
                                ;contains its proc.desc.

                                PG2
                                ;
                                0
                                ;*
                                PC1-PG2
                                ;
                                .TXT .PXX<0><0>.
                                ;*program name, three words.

PG2:    .                      ;program:
                                .

PG3:    .                      ;start of execution
                                ;

PG4:    .                      ;break address
                                .

```

```

PC1:                                ;process descriptor:
    .+0                             ;*
    .-1                             ;*
    0                                ;*
    PC2-PC1                          ;
    .TXT .PXX<0><0>.                 ;*process name, three words.

    .+0                             ;*when the event queue
    .-1                             ;*is empty it must point to
                                    ; itself

    MESB1                            ;*1. message buffer, if no
                                    ; message buffers, it must
                                    ; be 0.

    PG1                              ;*used to find the
                                    ; program descriptor
                                    ; e.g. when the process is
                                    ; removed.

    1B0                              ;*state = stopped
                                    ; during load.

    0                                ;
    1B7                              ;*zero must not be used.
    PG4                              ;*address entered
                                    ; at break.

    0                                ;
    0                                ;
    PC1                              ;*proc.desc.addr
                                    ; must be in AC2 at start.

    0                                ;
    PG3*2                            ;*execution is started
                                    ; in PG3 with carry = 0
                                    ; and the registers loaded
                                    ; with the four values above.

    0                                ;
    .                                ; optional words in
    .                                ; process descriptor.

    .                                ;

```

```

;message buffers:
MESB1:  .+0      ;
        .-1      ;
        MESB2    ;*next message buffer
        10       ;*

        PC1      ;*sender.proc.desc.addr.
        0        ;*buffer free
        .BLK 4   ; mess0-mess3 irrelevant

MESB2:  .+0      ;
        .-1      ;
        0        ;*last in chain
        10       ;*
        PC1      ;*sender.proc.addr.
        0        ;*buffer free
        .BLK 4   ;mess0-mess3 irrelevant
        .        ; optional words in
        .        ; process desc
        .
        .

PC2:    ;end of process desc.

        .END PC1 ;*the start address
        ; must be the process
        ; desc.addr.

```

#### Example 1.

Initialization of descriptors. Parameters marked with \* must be initialized by the programmer.



### 3. MONITOR.

3.

The primary purpose of the monitor is to implement multiprogramming that is simulation of parallel execution of several active processes on a single physical processor.

In order to do this, the running process is interrupted at regular intervals (20 msec.) by a Real Time Clock device, RTC. When such an interrupt occurs, the monitor gains control of the processor, saves the registers of the interrupted process and determines, which process is to get the next time slice for instruction execution. Switching from one process to another is also performed, whenever a process must wait for the completion of input/output.

Another purpose of the monitor is to execute indivisible functions in disabled mode, which is necessary to implement multiprogramming. E.g. functions allowing synchronization between processes and exchange of information in such a way that only one process at a time access the information. Also interrupts from all other devices than RTC are intercepted by the monitor, and the interrupt handling monitor functions give processes the ability to synchronize with the devices. The monitor functions are described in chapter 3.2.

#### 3.1 Monitor Descriptor.

3.1

The monitor is itself organized as a process. Its process descriptor contains variables and tables for the monitor. Only the first part of the monitor process descriptor, corresponding to a normal process descriptor, is shown here. Some of these parameters are used as the corresponding parameters in the normal process description in order to let the monitor run as a dummy process when no other process wants execution time. The remaining

locations are used for monitor variables and constants.

CUR: First process in running queue.  
 +1: Last process in running queue.  
 Head of running queue.  
 +2: First in process chain.

WORK1: Monitor work location.  
 WORK2: Monitor work location.

TABLE: Ref. to start of device table.  
 TOPTA: Ref. to top of device table.

DFIRS: First process in delay queue  
 +1: Last process in delay queue.  
 Head of delay queue.  
 +2: Ref. to break process-function.

PFIRS: First in program chain.  
 +1: State. Monitor state is always zero, because the monitor is always in running queue.

RUNNI: Ref. to head of running queue ( $= 40_8$  ).  
 +1: Priority. Monitor priority is zero, which is the lowest possible.

EXIT: Monitor exit address.

EFIRS: First in free core.  
 FFIRS: Last in free core.

DELAY: Ref. to head of delay queue.  
 +1: The dummy program. (jmp .+0).  
 +2: Psw of dummy process. (. -1).

AREAP: Ref. to head of area process chain.  
 AFIRS: First in area process chain.

FREQU: Frequency of RTC. Must always be zero, which corresponds to 50 Hz (=20 msec).

MASK: Interrupt mask.

CORES: Core size.

PROGR: Ref. to head of program chain.  
 +1: Address of monitor clear interrupt routine.  
 +2: Address of RTC clear interrupt routine.

- RTIME: Real time count. Two words, where the number of real time clock interrupts is counted. The number is  $(RTIME+0)*216 + (RTIME+1)$ .
- POWIN: Power failure. The number of power failures since dead start.

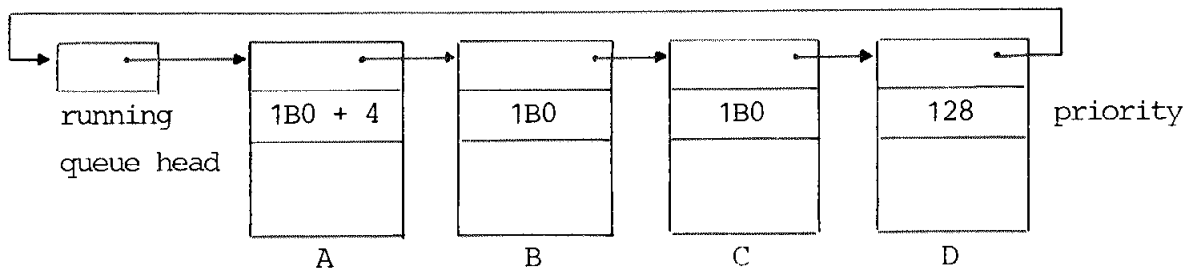
### 3.1.1 Running Queue.

3.1.1

The first two locations of the monitor process descriptor contain the head of the running queue. All processes wanting to compete for time slices are linked to the running queue. The first process in the running queue is the one that is actually running, and therefore a process can always find its process descriptor address in CUR (except in a drivers clear interrupt routine. See chapter 4). This process is called the current process.

The priority of a process is a 16 bit integer. The priority of the monitor process i.e. the dummy process is zero, which is the lowest possible and must not be used by other processes. Insertion in the running queue is done in order of the priority that is, the process with the highest priority is inserted as the first in the queue. Among processes of the same priority, a new is inserted as the last one (see fig. 2).

At each interrupt from RTC, the current process is relinked as the last process with that priority and the process that now is first in the queue, becomes the current process.



After insertion of process E with priority 1B0 the running queue is as follows:

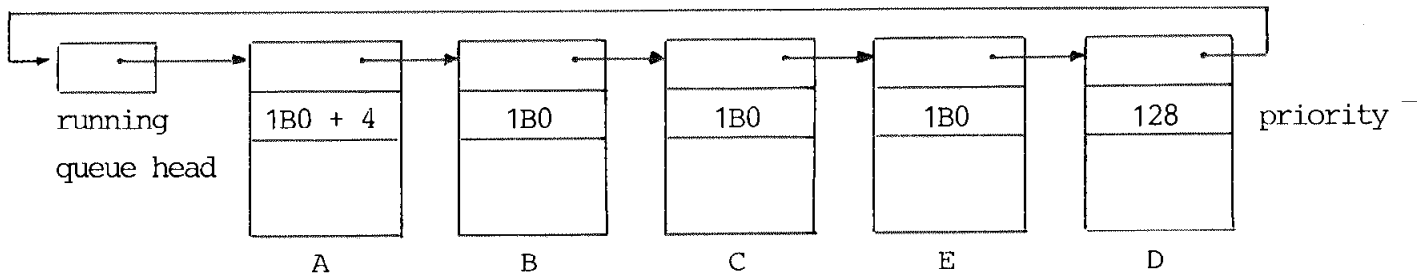


Fig. 2

### 3.1.2 Process Chain.

All process descriptors are linked together in a process chain. The chain field is word no. two relative to the start of the process descriptor, but the contents of the field is the address of the first word of the next process descriptor in the chain. The predefined symbol PROCES contains the address of the monitor process descriptor and can therefore be used as reference to the head of process chain.

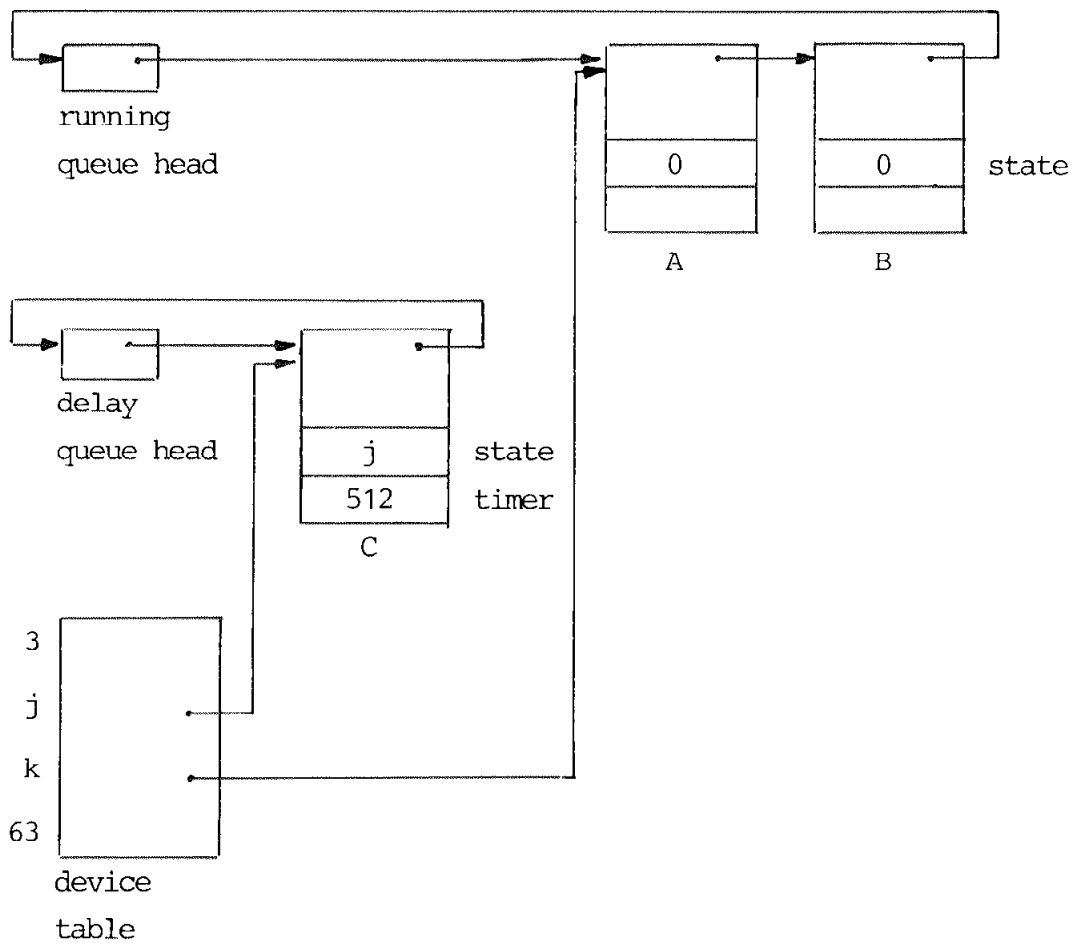
### 3.1.3 Device Table.

3.1.3

All interrupts from devices are intercepted by the monitor, and processes can then synchronize with devices by using the interrupt handling monitor functions. A device table, containing one word for each device number, is maintained by the monitor. When a process wants to supervise a device, it inserts its own process descriptor address in bit 0-14 of the device table entrance belonging to the device. Only one process, called a driver process, can supervise a device. If a driver process wants to wait for an interrupt from a device, it calls the monitor function `WAITINTERRUPT`, which removes the process from the running queue and sets its state to waiting for interrupt (fig. 3).

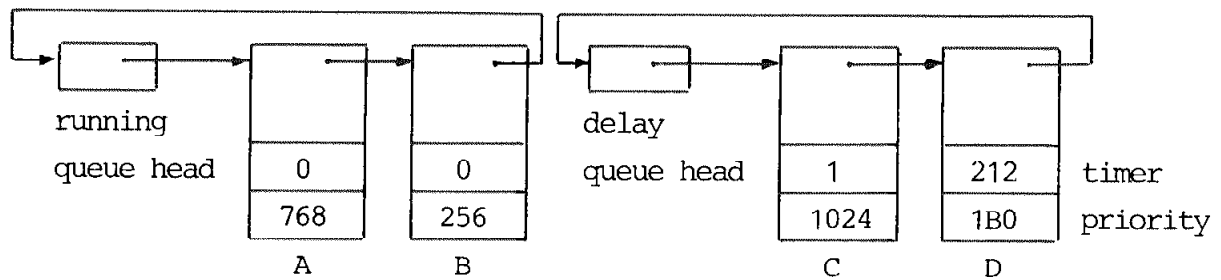
When an interrupt arrives, bit 15 in the appropriate device table entrance is set and the process descriptor is used to find the clear-interrupt routine (in `proc.clint`), which is then executed. If the driver process is waiting for interrupt, bit 15 is cleared again and the process is linked to the running queue. If not, bit 15 indicates that an interrupt is pending and next time the process wants to wait for an interrupt, it is immediately relinked to the running queue.

Driver processes are described in detail in chapter 4.



Device no. k is supervised by process A and device no. j is supervised by process C, which is waiting for interrupt from the device, or for software timer.

Fig. 3.



After the next interrupt from RTC, the timer in process C becomes 0 and the situation will be as follows.

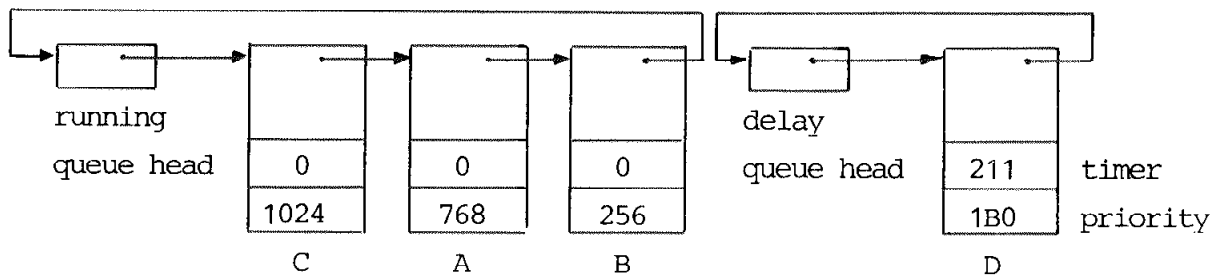


Fig. 4

During the initialization of the system, the monitor process is inserted as supervisor for all devices. I.e. if an interrupt arrives from a device which is not supervised by a driver process, the monitors clear interrupt routine is executed. This routine clears the interrupt and counts the number of such "unknown" interrupts in a variable located just in front of the first instruction of the routine.

#### 3.1.4 Delay queue.

3.1.4

If a process is not running, it may be waiting for an event (i.e. a message or an answer), waiting for an interrupt or stopped. In all cases of waiting, the monitor function used makes it possible to specify a maximum number of timer periods, it wants to wait.

If this possibility is used, the process is linked to the delay queue and the specified number of periods is inserted in the parameter `TIMER` in the process descriptor. For each timer (RTC) interrupt, `PROC.TIMER` is decreased by one for all processes in the delay queue. When the timer count for a process becomes zero it is started, i.e. removed from the delay queue and inserted in the running queue, according to its priority (see fig. 4). `DELAY` contains a reference to the head of the delay queue.

### 3.1.5 Program Chain

3.1.5

All program descriptors are linked together in a program chain. `PFIRS` is the head of this chain and `PROGR` is a pointer to the head of the program chain and can be used as start address when searching for a specific program in the chain (see `SEARCHITEM`, chapter 3.2).

### 3.1.6 Free Core Area.

3.1.6

During the initialization of the system, the core size is found and inserted into the parameters `CORES` and `FFIRS`. `EFIRS` is used by `MUS` and points at the first free location in core, while `DOMUS` uses the parameter `CORE` (not shown here) as head of a chain of free core items.

### 3.1.7 The Dummy Process.

3.1.7

As mentioned before, the monitor process descriptor is organized as a normal process descriptor, containing the parameter values necessary to let the monitor run a dummy process when no other process wants to run. The program executed by the dummy process is the instruction `JMP.+0`. The location corresponding to `PROC.PSW`



points to the dummy program. Other parameters needed to enable the monitor to run as a dummy process are state and priority, which are both zero. Zero is the lowest priority possible and must not be used by other processes.

### 3.1.8 Area Process Chain.

3.1.8

A pool of free area processes are linked together in a chain with head in AFIRST. AREAP is a reference to this head. Area processes are described in chapter 8.

### 3.1.9 Interrupt Mask.

3.1.9

A copy of the hardware interrupt priority mask is kept in MASK. The MUS system initialization applies a zero mask, i.e. interrupt requests are enabled from all devices, and the system does not support the use of the instruction MSKO. It is not recommended to use this instruction but if necessary, it must be carried out in disabled mode and the value needed to disable interrupts from the priority levels wanted must be or'ed to MASK and the result restored in MASK (also in disabled mode ) before the interrupt mask out instruction is applied with the new mask.

After a power failure the system is again supplied with a zero interrupt priority mask.

### 3.1.10 Real Time Clock.

3.1.10

From software it is possible to select between four real time clock frequencies, but in the MUS/DOMUS system, 20 msek. is always used.

3.1.11 Power Failure.

3.1.11

At power failure the accumulators and the program counter are saved in the process descriptor of the current process and a jump instruction to the power restart routine is saved in location 0.

At power-up, an IORST instruction is executed and all processes in the device table are break'ed with error number -4. (See BREAKPROCESS, chapter 3.2).

3.2 Monitor functions.

3.2

This chapter describes the monitor functions. Monitor functions are executed in disabled mode and are called from assembler programs by writing their names. In the assembler, they are defined as permanent symbols and are substituted with jump-subroutine-instructions. In the following 'link' is the address of the first instruction after the function-call, and it is automatically contained in AC3 when the function is entered. The return value of AC3 is for all functions the process descriptor address of the calling process (cur). If nothing else is mentioned, the function returns to the address 'link'.

3.2.1 Function WAITINTERRUPT.

3.2.1

	call	return
AC0		unchanged
AC1	device	device
AC2	delay	cur
AC3	link	cur
link+0	timeout	
link+1	interrupt	

The corresponding entry in the device table is checked for an interrupt. If an interrupt is pending, return is immediately made to (link+1).

If no interrupt is pending, the process is removed from the running queue and is inserted in the delay queue. 'Delay' (AC2) is inserted as timer count in the process descriptor and the process is stopped with state waiting for interrupts or software timer (state = device).

If delay = 0, a maximum delay period of 65535 timer periods ( 25 minutes) is used. If delay = 1, the waiting period is between 0 and 20 msec. Waitinterrupt may be used as a pure timer when device = 0. In this case state = 0 is preserved.

If an interrupt arrives before the time specified by 'delay' runs out the process is set running (i.e. removed from delay queue and inserted in running queue with state = 0) and return is made to link+1. Otherwise, the process is set running when the delay period runs out and return is made to link.

Note: Before any call of WAITINTERRUPT with device  $\neq 0$ , the device table must be initialized to process-descriptor-address\*2. This may be done by procedure SETINTERRUPT (see chapter 4).

Ex:

```

.
.
JSR          DGCOM,2    ;start device
LDA          1         DGDEV,2    ;load device no.
LDA          2         .512      ;load timer
WAITINTERRUPT                ;waitinterrupt (device,
                               ;delay);
JMP          DGTIM     ;+0: timeout return
.                ;+1: interrupt return
.
.
.
.
.
.
.
DGTIM:
.
.
.

```

Example 2.

3.2.2 Function SENDMESSAGE.

3.2.2

	call	return	error return
AC0		unchanged	unchanged
AC1	address	address	address
AC2	name address	buf	error number
AC3	link	cur	cur

Selects a free message buffer belonging to the calling process and copies the four message words placed from 'address' and onwards into this message buffer (BUF.MESS0-BUF.MESS3). The message buffer is then linked into the event queue of the receiving process with name placed in 'name address' and onwards. The receiving process is activated, if it is waiting for an event. The calling process continues execution after being informed about the address of the message buffer in AC2.

If a process with the given name could not be found in the process chain SENDMESSAGE returns with error number = -2. If the message buffer pool of the sending process does not contain a free message buffer, the function returns with error number = -3.

Ex:

```

TTYNA:  .+1

        .TXT.TTY<0><0>.      ;

MESAD:  .+1                  ;address of message
        1                    ;mess0
        80                   ;mess1
        INSTR*2              ;mess2
                                ;mess3, irrelevant here

INSTR:  .BLK 40

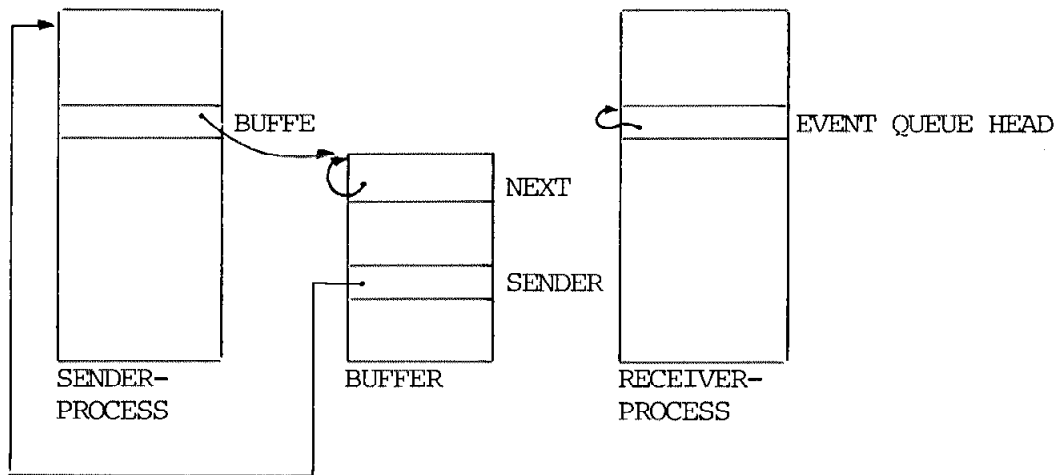
STADR:  LDA      1          MESAD  ;message address
        LDA      2          TTYNA  ;name address
        SENDMESSAGE        ;sendmessage (mesad,
                                ;ttyna)

        MOVZL#   2, 2      SZC     ;if TTY not found
        JMP      ERROR    ;then got error
        WAITANSWER        ;else wait for answer to
        .                ;that buffer
        .

```

Example 3.

Before SENDMESSAGE:



After SENDMESSAGE

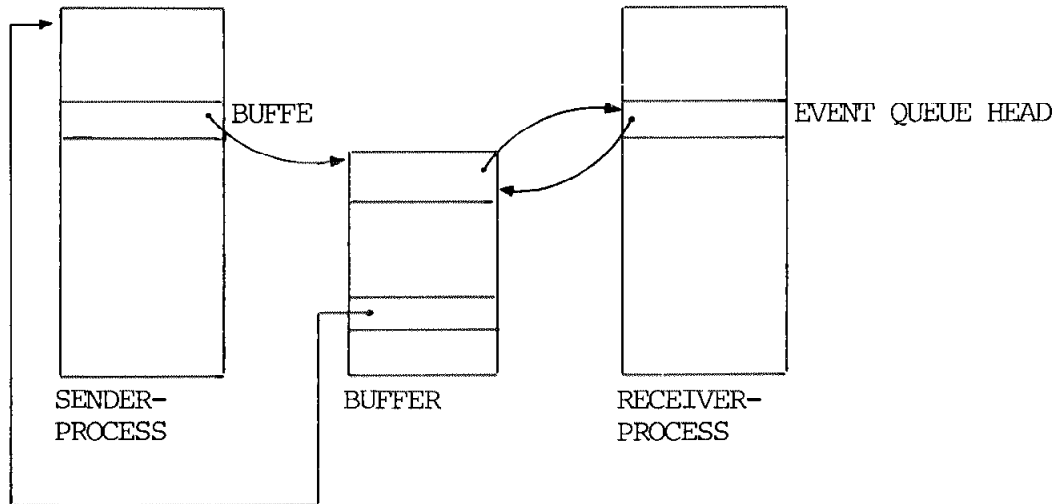


Fig. 5

### 3.2.3 Function WAITANSWER:

3.2.3

	call	return
AC0		first
AC1		second
AC2	buf	buf
AC3	link	cur

When a process has sent a message, the answer to it will be returned in the same message buffer. WAITANSWER delays the process until an answer arrives in the message buffer given as a parameter. The first two words of an answer (i.e. BUF.MESS0 and BUF.MESS1) are copied into AC0 and AC1 and the message buffer is



released.

Note that WAITANSWER is the only monitor function that releases a message buffer. This means that WAITANSWER must also be used if the process has received information about the answer by means of WAITEVENT or WAIT (see these functions).

Ex:

```

LDA      3      CUR      ;
LDA      2      BUFPE,3  ;
WAITANSWER                                ;waitanswer (1.mess buf-
                                           ;fer)
MOV#     0,0    SZR      ;if buf.mess0<>0 then
JMP      ERROR  ;goto error;
.        .      ;the message buffer
.        .      ;is now released.

```

Example 4.

After WAITANSWER:

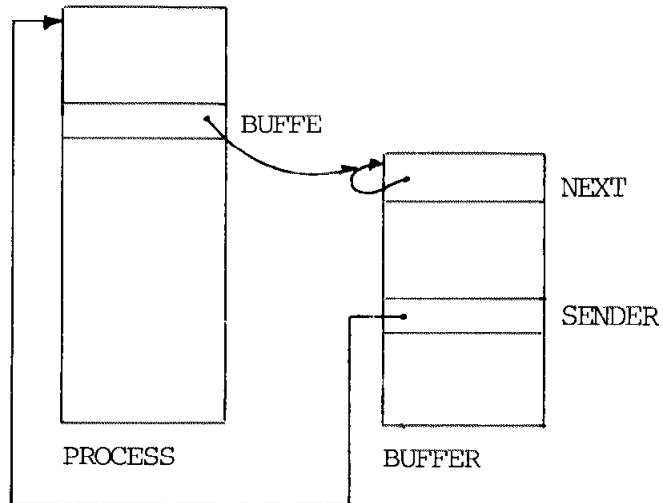


Fig. 6

#### 3.2.4 Function WAITEVENT.

3.2.4

	call	return
AC0		first
AC1		second
AC2	buf	next buf
AC3	link	cur

Link+0	answer
link+1	message

The process is delayed until an event (a message or an answer) is linked to its event queue after the message buffer given as para-

meter. If 'buf' is zero, the event queue is examined from its beginning. The calling process is supplied with the address of the new event in AC2 and with MESS0 and MESS1 from the event in AC0 and AC1. If the event arrived is an answer, the function returns to link+0; otherwise link+1 is chosen.

Note that if an answer arrives, WAITANSWER must be used to release the message buffer.

Ex:

```

      SUB      0,0          ;buf:= 0
      WAITEVENT          ;waitevent
      JMP      ANSW       ;+0: answer
      .              ;+1: message
      .
      .
      .
      .
ANSW:  WAITANSWER          ;release message buffer
      .
      .

```

Example 5

3.2.5 Function WAIT.

3.2.5

	call	return (answer or message)	return (timeout or interrupt)
AC0	delay	first	unchanged
AC1	device	second	device
AC2	buf	next buf	cur
AC3	link	cur	cur
link+0	timeout		
link+1	interrupt		
link+2	answer		
link+3	message		

This function performs the combined functions of WAITINTERRUPT and WAITEVENT. 'Delay' is inserted as timer count in the process descriptor and the process is linked to the delay queue. If 'device' is non-zero, the device table is checked for an interrupt. Then it waits, either for an interrupt, a timeout or an event after the buffer given as a parameter. If 'buf' is zero, the event queue is examined from the beginning.

The return depends on, what happens first as listed above, and the contents of the registers are as for WAITEVENT or WAITINTERRUPT depending on the return.

Note again that if an answer arrives, the message buffer should be released by means of WAITANSWER.

Ex:

```

LDA      2      CUR      ;
LDA      0      .512    ;delay
LDA      1      DEV,2   ;device
SUB      2,2     ;buf: = 0
WAIT                               ;wait (delay, device,
                                   ;buf)

JMP                               DTIM    ;+0: timeout
JMP                               DINT    ;+1: interrupt
JMP                               DANS    ;+2: answer
.                                       ;+3: message

```

Example 6

3.2.6 Function SENDANSWER.

3.2.6

	call	return
AC0	first	first
AC1	second	second
AC2	buf	buf
AC3	link	cur

'Buf' is the address at the message buffer which the calling process wants to answer. 'First' and 'second' are copied into MESS0 and MESS1 of the message buffer and the buffer is then delivered as an answer in the event queue of the original sender (i.e. the owner of the buffer). If the calling process also wants to return information in MESS2 and MESS3 of the buffer, this information must be stored directly in BUF.MESS2 and BUF.MESS3 by the calling process before SENDANSWER is called.

Ex:

```

WAITEVENT                                ;
JMP          DANS                        ; +0: answer
STA         2          BUF,3            ; +1: message
.                                                  ;
.                                                  ;handling of message

SUB         0,0                          ;buf.mess0: = 0
LDA         1          COUNT,3          ;buf.mess1: = cur.count
LDA         2          BUF,3            ;
SENDANSWER

```

Example 7

After SENDANSWER:

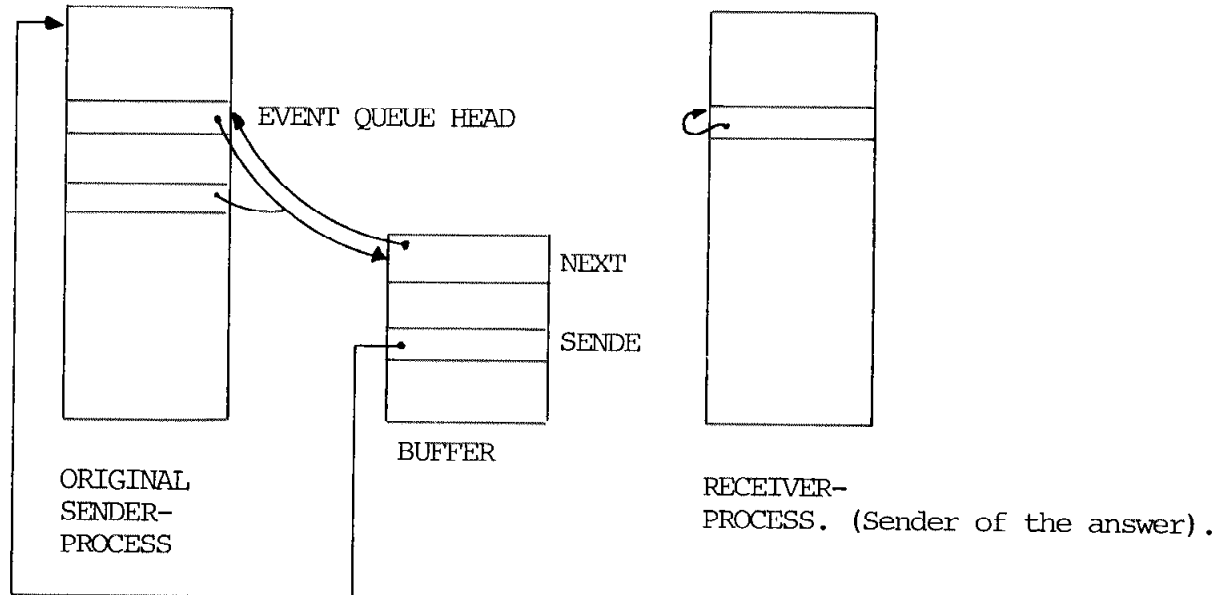


Fig. 7

### 3.2.7 Function SEARCHITEM.

3.2.7

	call	return
AC0		unchanged
AC1	chain head	chain head
AC2	name addr	item
AC3	link	cur

If the chain with the address of the chain head placed in AC1 contains an item with the name placed in 'name address' and on-

wards, the address of this item is delivered; otherwise a zero is delivered.

The address of the head of the process (program) chain is contained in the monitor parameter PROCE (PROGR). An item in the process chain can be found as shown in ex. 8.

Ex:

```

NAADR:    .+1
          .TXT .PTR<0><0>.

START:    LDA      1      PROCE    ;process chain head
          LDA      2      NAADR    ;name address
          SEARCHITEM      ;searchitem (PROCE, PTR)
          MOV#     2,2    SNR      ;if item not found
          JMP      NOTFO  ;then goto notfo.
          .           ;else continue
          .
          .
          .
          .

```

Example 8



Ex:

```

C1:      .+3                ;chain
        S100                ;size
        .TXT .P100<0>.     ;name

        .+3                ;chain
        S101                ;size
        .TXT .P101<0>.     ;name

        .+3                ;chain
        S102                ;size
        .TXT .P102<0>.     ;name

        0                   ;chain
        S103                ;size
        .TXT .P103<0>.     ;name

C1REF:   .+1-CHAIN          ;ref. to head of chain C1
        C1-CHAIN            ;head of chain C1

;

NAADR:   .+1
        .TXT .P102<0>.

START:   LDA      1         C1REF   ;chain head
        LDA      2         NAADR   ;name address
        SEARCHITEM          ;searchitem (C1, name)
        MOV#     2,2       SNR     ;if item not in C1 then
        JMP      NOTFO     ;goto notfound;

```

Example 9

3.2.8 Function BREAKPROCESS.

3.2.8

	call	return
AC0	error number	error number
AC1		unchanged
AC2	proc	proc
AC3	link	cur

The process given as parameter with process descriptor address in AC2 is started at its break address with the following accumulator contents:

AC0	error number
AC1	unchanged
AC2	proc
AC3	PSW (its old program counter)

The following error numbers are used by MUS/DOMUS system procedures:

- 4: All processes inserted in the device table are break'ed with error number -4 at power interrupt.
- 3: From the coroutine monitor procedure CSENDMESSAGE. No system operations available to send messages. [3].
- 1: The break'ed process has received a message from a process which has been cleaned after it has sent the message (see function CLEANPROCESS).
- 2: The process is break'ed by the operator.

- 3: A MUSIL program is break'ed with error number 3 if it reaches the end.
- 5: The status returned from a zone contains a hard-error-bit which is not set in the giveup mask of the zone (see WAITTRANSFER, chapter 5).
- 6: From the coroutine monitor procedures SAVELINK and RETURN. Stack over- or underflow. [3].
- 7: a) Page system break.  
b) Break from the coroutine monitor procedure WAITGEN. No system operation available for receiving messages although reserved. [3].
- 8: Reserver removed. Only reservable (driver) process that is processes with bit 7 set in the program description word.

Ex:

```

LDA      1      PROCE      ;
LDA      2      NAADR      ;searchitem (process-
SEARCHITEM                      ;chain, name)
MOV#     2,2    SNR        ;if proc not found
JMP                      ERROR ;item goto error
LDA      0      .16       ;else
BREAKPROCESS                      ;breakprocess (16, proc)
.
.
.
.
```

Example 10

3.2.9 Function CLEANPROCESS.

3.2.9

	call	return
AC0		unchanged
AC1		unchanged
AC2	proc	proc
AC3	link	cur

Messages to the process given as parameter are answered with status (= MESS0) = 0 and bytcount (= MESS1) = 0. Answers to the process are released. Messages from the process are released and the receivers are break'ed with error number = 1.

The function is used to tidy up, e.g. the operating system S applies it before killing a process.

CLEANPROCESS must be used with special care, as it may cause the processor to run in disabled mode for an unpredictable time.

Ex:

```

LDA      2      CUR      ;the process
CLEANPROCESS      ;cleans itself

```

Ex:

```

NAADR:  .+1          ;
        .TXT .P22<0><0>. ;

START:  LDA      1      PROCE ;
        LDA      2      NAADR ;
        SEARCHITEM      ;searchitem (PROCE, P22)
        MOV#     2,2    SZR   ;if P22 in processchain
        CLEANPROCESS      ;then CLEANPROCESS (P22)
        .
        .

```

Example 11

### 3.2.10 Function STOPPROCESS.

3.2.10

	call	return
AC0		unchanged
AC1		unchanged
AC2	proc	proc
AC3	link	cur

The process is set in state stopped and removed from the delay- or running queue. If it was waiting for event or answer, PSW is decreased by 1. This ensures that the monitor function is called again if STARTPROCESS is performed.

Ex:

```

        LDA      2      CUR      ;the process
        STOPPROCESS      ;stops itself.

```

3.2.11 Function STARTPROCESS.

3.2.11

	call	return
AC0		unchanged
AC1		unchanged
AC2	proc	proc
AC3	link	cur

If the process is in state stopped, it is set running. Otherwise, the function is dummy.

Ex:

```

LDA      1      PROCE      ;
LDA      2      NAADR      ;searchitem, (process-
SEARCHITEM      ;chain, name addr)
MOV#     0,0     SNR        ;if process found
JMP                      ERROR      ;
STARTPROCESS      ;then start it;

```

Example 12

3.2.12 Function RECHAIN.

3.2.12

	call	return	error return
AC0	old	old	old
AC1	new	new	new
AC2	elem	elem	-3
AC3	link	cur	cur

The parameters 'old' and 'new' are chain head addresses and 'elem' is the address of an item. This item is removed from the old chain and inserted in the new chain.

Error return: If the element does not exist in the old chain, the function returns with AC2 = -3.

Ex:

```

C1:      .+3                ;chain
         S100              ;size
         .TXT .P100<0>.    ;name

         .+3                ;chain
         S101              ;size
         .TXT .P101<0>.    ;name

CHELM:   .+3                ;chain
         S102              ;size
         .TXT .P102<0>.    ;name

         0                  ;chain
         S103              ;size
         .TXT .P103<0>.    ;name

C2:      0                  ;empty chain

C1REF:   .+1-CHAIN          ;ref to head of chain C1
         C1-CHAIN          ;head of chain C1

C2REF:   .+1-CHAIN          ;ref to head of chain C2
         C2-CHAIN          ;head of chain C2

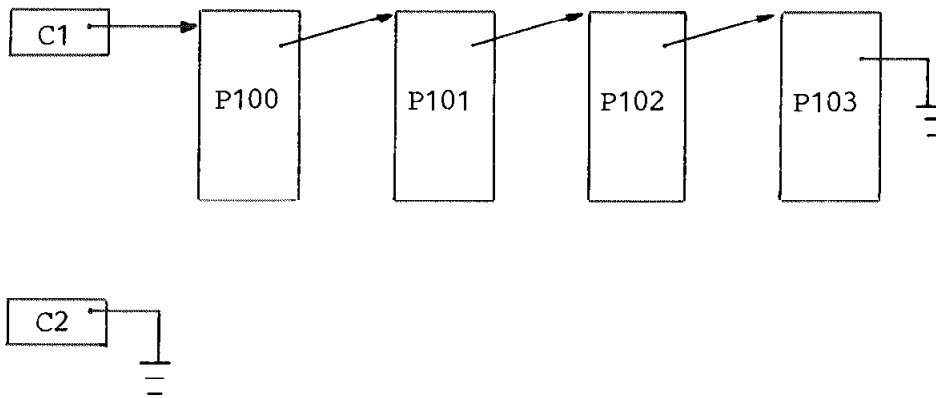
ELEM:    CHELM-CHAIN        ;P102

START:   LDA      0         C1REF  ;old: = chain C1
         LDA      1         C2REF  ;new: = chain C2

```

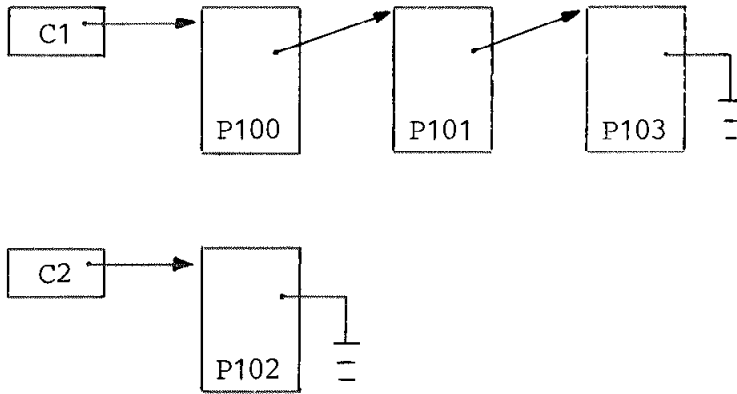
```
LDA      2      ELEM      ;elem: = P102
RECHAIN
MOVZL#   2,2    SZC       ;if elem not in chain
                        ;then
JMP      ERROR  ;goto error
.
.
```

Before RECHAIN:





After RECHAIN:



Example 13

### 3.3 Initialization of the Monitor.

3.3

When the basic system is loaded, the initialization module (MUI) is started. After an IORST has been performed and the real time clock has been started, all processes in the basic system are linked into the process chain and the running queue and their programs are inserted in the program chain. The process descriptor addresses of processes in the basic system are found from location  $402_8$  and on, where they have been inserted by system generation programs. This sequence stops with the value -1. Then the device table is initialized with the program status word (PSW) of the dummy process, which means that if an interrupt appears from

a device not supervised by a driver process, the clear interrupt routine specified in the monitor process descriptor is executed. This routine counts all such interrupts in a variable, placed just in front of its first instruction.

At last the core size is calculated and the first process in the running queue, which is S because S has the highest priority, becomes active. As MUI is always placed after S in a basic system, it is overwritten when the first process is loaded by the operator.

### 3.4 Processor Expansion.

3.4

The processor expansion system, which makes it possible to communicate between processes placed in two different processors, consists of a software module, XCOMX, a copy of which is placed in each processor. XCOMX is a normal MUS process which takes care of sending messages and data destined for processes in the other processor and returning answers the other way. The system is described in [4]. The only monitor function that takes special care of XCOMX is SENDMESSAGE. When a message is sent, SENDMESSAGE searches for the receiving process in the process chain. If it is found, the message is inserted in its event queue, but if it is not found, the function will instead search for XCOMX. If XCOMX is present, the message is linked into the event queue of XCOMX, which then takes care of sending it to the other processor. In addition the name of the receiving process and the address of the buffer are stored in a table starting in the monitor variable COMLIST. This information is needed by XCOMX. COMLIST and the length of this table, COMNO, is set by XCOMX, when it is loaded.

## 4. DRIVER PROCESSES.

4.

### 4.1 Introduction.

4.1

A driver process is a normal process seen from the monitor, but it is dedicated to communicate with a device. Under special circumstances it might take care of several devices. E.g. console input and console output driver.

The reason for introduction of processes dedicated to device control are:

- to let more than one process communicate with a device. Without a driver as interface this would demand explicit arrangements among involved processes.
- to handle devices in a more uniform way. That is, introduction of standard operations, standard status information and blocking of all input/output, also for character oriented devices.
- to realize simple and time-saving conversions of characters directly from input or output to the devices.

Other processes must then request the driver process to perform input/output by means of messages, and the driver processes are thus the only processes which actually execute I/O instructions and take care of the device dependant interrupt service.

The messages accepted by the driver process should conform to the below mentioned standards, and the answers should also be of a standard form.

As the zone procedures used by the application program expect special reactions in connection with the exception handling all

messages must be returned by the driver within a finite time. Furthermore, the driver must enter a reject state after return of a hard error status, in which all transput messages are rejected with a non-processed status.

This reject state is important and necessary when input/output is of a sequential nature, as the errormarked message must be input/output before the following transfer request are granted when multibuffered input/output is used.

This driver reaction is not used in the handling of devices with an input/output structure more complex than simple sequential input/output transfer.

The driver reject state must be cleared by special messages send by the user process.

#### 4.2 Device Handling.

4.2

Before any I/O instructions are executed, the driver process should clear the device in question by the device specific I/O instruction (normally a NIOC DEVNO).

Then the driver must insert its process descriptor address shifted one left (bit 15 = 0) in the corresponding device table entry, either directly or by a call of the procedure SETINTERRUPT, which will also execute the above mentioned NIOC instruction.

ex:

```

; DIRECT INSERTION IN DEVICE TABLE
;
LDA      3      E000      ; ADDR: = TABLE
LDA      1      E001      ;
ADD      1,3     ; ADDR: = TABLE + DEVNO;
LDA      2      CUR       ;
MOVZL    2,0     ; WORD(ADDR): = CUR
;                               SHIFT 1;
STA      0      +0,3     ;
.
.
E000:    DEVTA   ; ADDRESS OF DEVICE TABLE
E001:    DEVNO   ; DEVICENUMBER

```

Example 14A

```

; USE OF PROCEDURE SETINTERRUPT
;
LDA      1      E002      ;
SETINTERRUPT ; SETINTERRUPT (DEVNO);
.         ; NIOC DEVNO
.
E002:    DEVNO

```

Example 14B

When the device table word is initialized, all interrupts from the device will cause the call of a user defined interrupt procedure given by its address in the process descriptor with the displacement CLINT (see fig. 8).

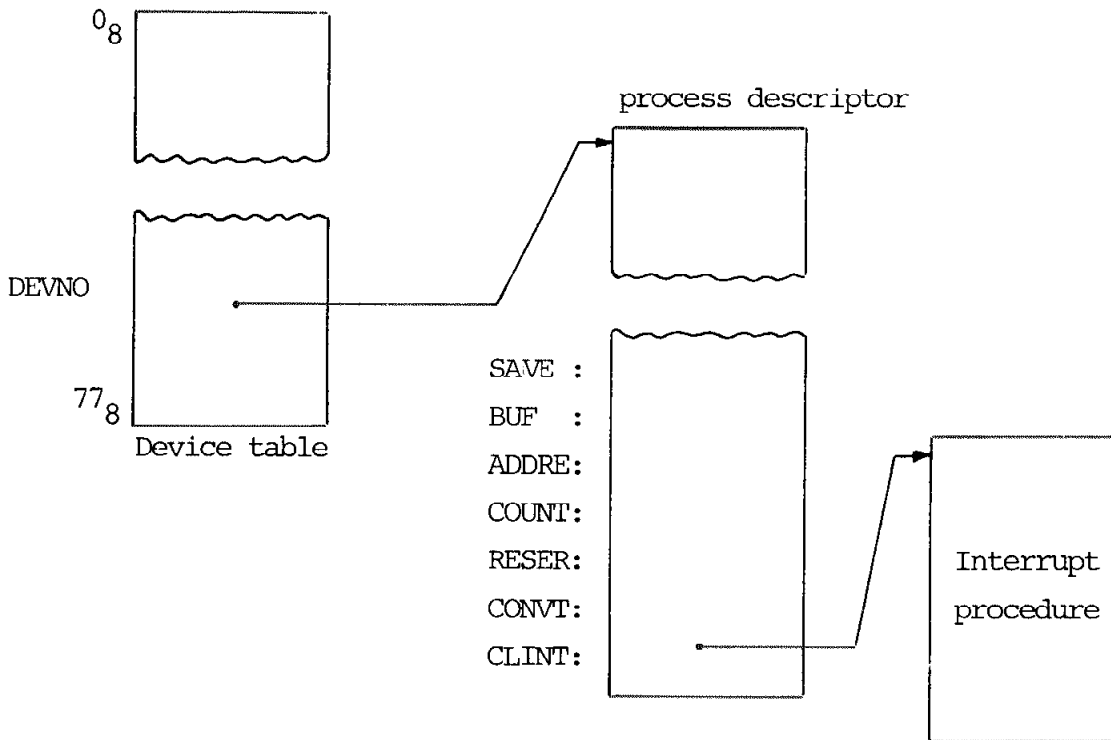


Fig. 8 Organisation of device table, process descriptor and interrupt procedure in a driver process.

The user defined interrupt procedure must obey the following conventions:

User interrupt procedure

	Call	Return
AC0	-	destroyed
AC1	device	unchanged
AC2	process descriptor addr	unchanged
AC3	link	destroyed

The procedure is called by the monitor in disabled mode and it must not change this state. It must return with the device interrupt request cleared, and it must not call other system procedures or use any system variables. (The standard variable CUR t.ex. points to the interrupted process and not to the driver process).

The amount of data processing must be as small as possible since it affects the system overhead.

Priorities in interrupt service by means of the hardware MASKOUT feature is not supported. Instead, the interrupt service time should be kept at a minimum in the disabled interrupt service routine, and the priority between different devices can then be achieved by the process priority, as active processes are assigned CPU time according to the process priorities. Data handling should also be done outside the interrupt service routine as all system variables and procedures can be used freely in this case.

The system offers a standard interrupt procedure which only executes a NIOC <device>. This procedure can be selected by the system address CLEAR.

If at return the driver process is in a state waiting for interrupt from the interrupting device, the process is set running.

If the process is in any other state, the interrupt is indicated by the monitor setting bit 15 in the device table word, thus indicating one or more interrupts from the actual device.

This bit is only cleared if the process calls the `WAITINTERRUPT` function, in which case the return from `WAITINTERRUPT` takes place immediately, or when the bit is cleared by an initialization as mentioned above, using the procedure `SETINTERRUPT`.

If the process start or interrupt indication is not wanted, the return can be made by an indirect jump to the monitor exit action (`JMP@ EXIT`).

No check is performed on the selected device code, and special care must be taken to secure that two driver processes do not use the same device code.

The device codes 0, 1, 2, 3, 4,  $14_8$  and  $77_8$  are used exclusively by the system and must not be used by any driver.

Devicecode  $14_8$  is the real time clock (RTC) and code  $77_8$  is the CPU. Other standard RC 3600 device codes can be found in the appendix.

### 4.3 Driver Interface.

4.3

User programs communicate with a driver by means of messages with two standard formats: control and transput. A control message indicates an operation which does not imply any actual input or output but performs positioning, selects different facilities such as parity, density, baud rate, conversion etc. A transput message requests input to or output from a message defined core-area.

Regardless of the message type, the answer returned when a message has been processed by the driver process contains a status



word, which describes the success of the requested operation.

#### 4.3.1 Control Messages.

4.3.1

The standard format of a control message is:

	0	14 15
MESS0	MODE	00
MESS1	SPECIAL 1	
MESS2	SPECIAL 2	
MESS3	SPECIAL 3	

The mode is any array of bits which specifies the actions to be taken.

An action is performed if the corresponding bit is one. The interpretation proceeds normally from bit 13 to bit 0. Not all actions are relevant for specific driver processes and some complex drivers will give other reactions than the standard reactions described below:

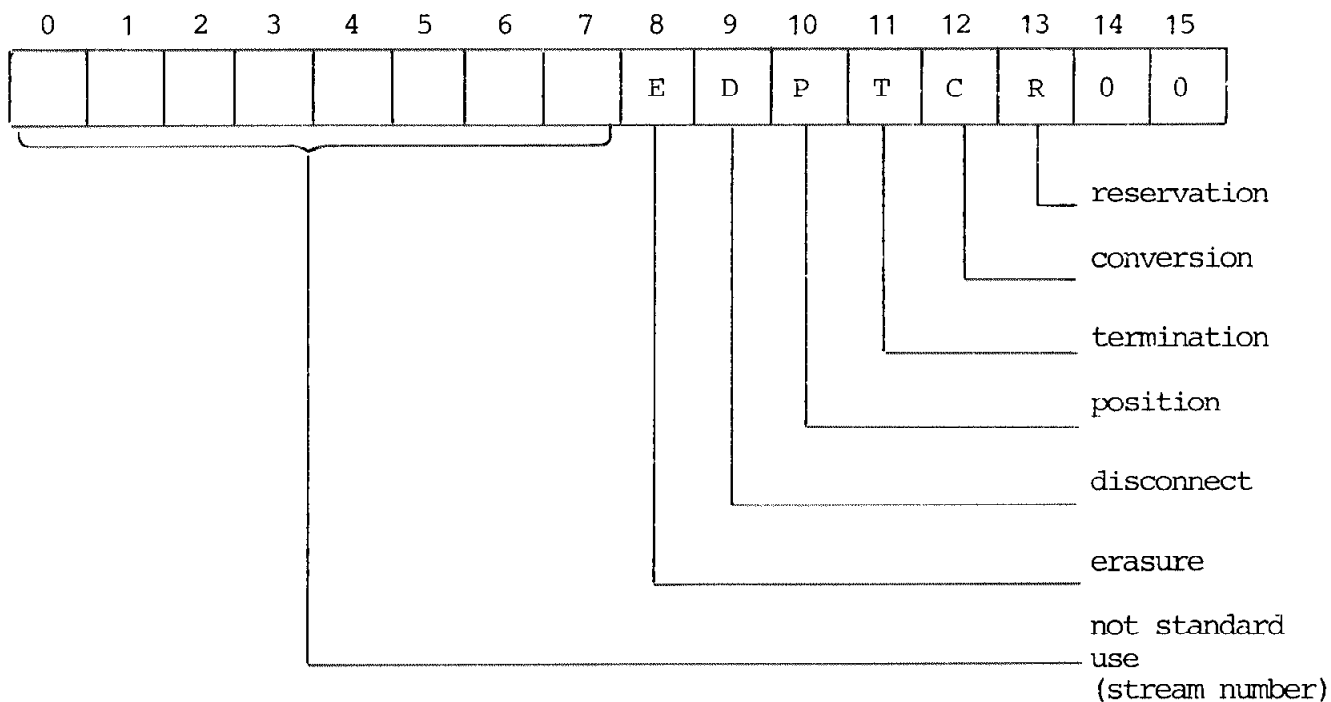


Fig. 9 Standard mode bits in RC 3600 MUS system.

The standard mode bits can be seen in the word layout in fig. 9.

### Reservation

If bit 13 is set in the MESS0 word, the action depends on the contents of MESS1 as follows:

MESS1  $\neq$  0: The sender of the message is set as reserver of the driver, i.e. the sender gains exclusive access to the driver process, and messages from other processes are rejected.

MESS1 = 0: The gained reservation is cancelled and the driver process will accept any message again.

### Conversion

If bit 12 is set in the MESS0 word, the MESS2 part (special 2) is taken as a byteaddress of a conversion table.

A table address of zero specifies no conversion. Format and interpretation of the table is dependent on the driver. Note that if conversion is used, reservation ought to be done.

### Termination

Normally used only by output devices, as it indicates that the data, which has previously been output, must be terminated logically. E.g. for a magnetic tape, a file mark is written and for a paper tape punch a leader is punched.

The termination action is taken, if bit 11 of the MESS0 word is set.

### Position

The document is positioned according to the information in the message, or the special parameters are taken from the message words MESS2 and MESS3.

If the message is interpreted as a true position command, MESS2 (special 2) is taken as the wanted file position and MESS3 (special 3) is taken as the new block position.

In special cases, the maximum value of MESS3 is too small and the wanted blockposition is calculated by taking the first byte of MESS0 as the most significant part and MESS3 as the least significant part.

As the presence of bit 10 in the MESS0 part of the message normally indicates parameters in MESS2 and MESS3, this bit is nor-

mally the only one set in the word and thus it is not used together with actions indicated by other bits.

#### Disconnection

If bit 9 of the MESS0 part is set, the device in question is set local if possible. E.g. a magnetic tape is rewound and the station is set off-line.

#### Erasure

Only relevant for output devices, which are able to cancel previous output. E.g. some inches of magnetic tape are erased or a flexible disc sector is marked as a skip sector.

#### General

If all bits are zero in the MESS0 part, a sense of the device is executed by the driver.

If the driver is able to handle a number of devices connected to the same controller with a single device code, the first byte can be used as a stream number and the above mentioned actions can then be done for the stream in question. E.g. several VDU's connected to a multiplexer.

$14_8$	$40_8$	MESS0
0	-	MESS1
$17777_8$	2	MESS2
-	$10_8$	MESS3

Example a) Control message with request for reservation and conversion table address  $17777_8$ .  
(I/O procedure OPEN).

b) Control message requesting position to file 2, block  $10_8$ .  
(I/O procedure SETPOSITION).

#### 4.3.2 Transput Messages.

4.3.2

A transput message requests an operation, which involves transfer of data to or from a core area. The core area can be transferred by the driver either byte by byte with programmed input/output or directly from the core area by hardware controlled DMA transfer (DMA = Direct Memory Access), but seen from the user process there is no difference as the data is handed over as a whole block.

The standard format of a transput message is:

4.3.2.1 Input.

4.3.2.1

MESS0	operation	01
MESS1	bytecount	
MESS2	byteaddress	
MESS3	special	

4.3.2.2 Output.

4.3.2.2

MESS0	operation	11
MESS1	bytecount	
MESS2	byteaddress	
MESS3	special	

Fig. 10 Standard formats of input and output messages.

The operation field of MESS0 transmits information about the mode of transfer. E.g. parity, format or special actions.

The bytecount in MESS1 specifies the number of bytes to be transferred to or from the core area, pointed out by the first byte given in the MESS2 word of the message (byte address).

MESS3 contains special information concerning the transfer t.ex. the block number of the wanted block to be read/written on a random accessible device.

The first byte of MESS0 can be used as a stream number or as an extension to the special information in MESS3.

Bit 8 set in MESS0 is normally used to indicate that MESS2 should be interpreted as a wordaddress.

The core area to input or output must have a layout as seen in fig. 11. The bytes are packed left to right in the memory words with the first byte in the lowest memory address.

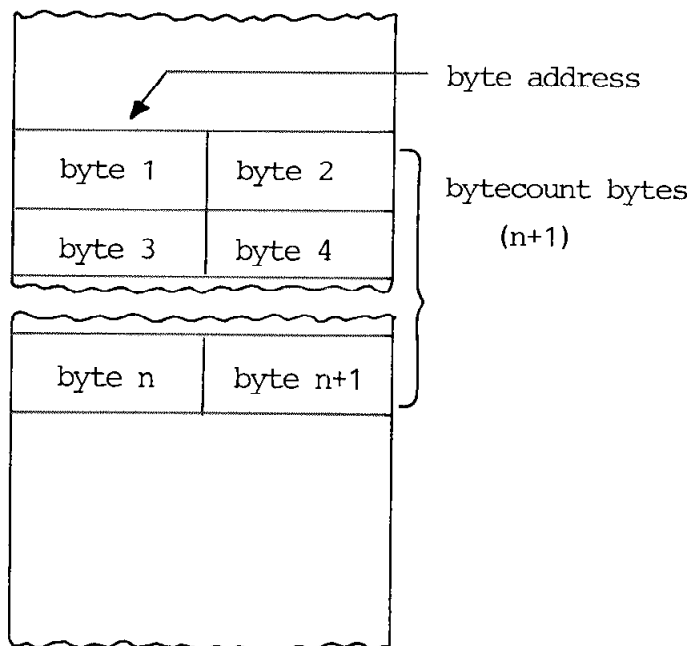


Fig. 11 Layout of core area pointed out by a transput message.

a)	5	
	$10_8$	
	$12765_8$	
		MESS0
		MESS1
		MESS2
		MESS3

Example 15: a) Input message requesting transfer of  $10_8$  bytes to address  $12765_8$ .

b) Output message requesting transfer of  $20_8$  bytes from address  $13775_8$  and on.

#### 4.3.3 Answers.

4.3.3

The answers returned in the message buffer is independent of the message type received and the format is:

#### 4.3.3.1 Answer.

4.3.3.1

MESS0	status
MESS1	bytecount
MESS2	special 1
MESS3	special 2

Fig. 12 Standard answer



The MESS0 part of the message is an array of bits called status, which conveys information about device errors or call errors and gives information about the success of the device action requested by the received message.

The different bits have been given special meanings in order to standardize error recovery in the user input/output procedures.

The status returned on a successful operation should be zero, except for the below mentioned bits 3, 4, 5 which have device dependent informative functions only.

The standard status bit interpretation is:

bit	mnemonic	clean	
0	disconnected	*	The device is not present.
1	off-line	*	The device is or has been off-line. The device is not ready.
2	device busy	*	The device was temporarily not able to execute the operation.
3	device spec 1		Device dependent, informative.
4	device spec 2		Device dependent, informative.
5	device spec 3		Device dependent, informative. Document write protected. (Returned with 'illegal' when output

			transfers are attempted)
6	illegal	*	Device reserved. Operation illegal or unknown.
7	eof	*	Logical end of document is detected. E.g. file mark read, end of transmission.
8	block error	*	The core area specified is too small to hold the input block, or the output block was too big for the document.
9	data late	*	The high speed data channel responded too late, and data was lost.
10	parity	*	One or more invalid characters were input or one or more characters were output badly on the document (device read after write feature).
11	end medium	*	Physical end of medium. E.g. end-of-tape, paper tape reader empty, paper out on printer.
12	position error	*	The requested position was nonsense or not found.

13			Not used by driver. Used by I/O procedures to indicate the absence of the driver process.
14	timeout	*	An expected interrupt was not received within a maximum driver specified time.
15	rejected	*	The message was returned without any treatment, because a previously returned answer contained a cleanbit as marked in this table.

All statusbits marked with \* are called cleanbits. This means that all the following transput messages received after the return of one or more of these bits should be rejected with status bit 15. This enables the user to reestablish the original sequence of messages in the driver event queue after an error recovery. The driver accepts transput messages again after reception of a control message.

Special care must be taken when status is returned on control messages. Logical meaningless status as block length error or parity error on position messages should be avoided.

The bytecount of answer specifies the number of bytes actually transferred.

Special 1 and special 2 can be used for special information concerning the transfer or operation, and they normally hold the filecount (special 1) and blockcount (special 2) if position information has meaning in connection with the actual device.

#### 4.4 System Utility Procedures.

4.4

##### 4.4.1 Formats.

4.4.1

As an aid to the driverprogrammer a number of actions, which frequently have to be executed in any driver, are collected as re-entrant routines. Furthermore, these routines are designed to give the standard driverinterface expected by the basic I/O procedures used for standard I/O operations.

If these procedures are used, the process descriptor should contain a number of extra words after the standard variable SAVE. These words are given by the displacements below and can be fetched relative to the process descriptor (CUR):

- BUF: Address of the current message buffer found by the procedures NEXTOPERATION and WAITOPERATION when returning. It is also used as a call parameter to these procedures set by the programmer or the procedure RETURNANSWER.
- ADDRE: Value of MESS2 of the current message buffer i.e. the first byte address of the data in case of a transput message. Used by the procedure RETURNANSWER for calculation of the bytecount in the returned message.
- COUNT: Value of MESS1 of the current message buffer, i.e. the number of bytes in case of a transput message.
- RESER: Word containing the address of the reserver-process description set and cleared by the procedure SETRESERVATION and checked by the procedures NEXTOPERATION and WAITOPERATION.

CONVT: Word containing the conversion table address fetched from the message indicated by BUF when SETCONVERSION is called.

CLINT: Address of the interrupt procedure to be called by the monitor when the device requests an interrupt. System address CLEAR can be used if only a NIOC <devno> is wanted.

STTAB: Standard conversion table address. Used by some drivers supporting standard conversion. If the user requests no conversion, this address is taken as the conversion table address. STTAB is set by special system programs.

#### 4.4.2 Procedures.

4.4.2

##### 4.4.2.1 Procedure NEXTOPERATION.

4.4.2.1

call return (+0, +1, +2):

AC0	-	mode
AC1	-	count
AC2	cur	cur
AC3	-	buf

+0: control message received.

+1: input message received.

+2: output message received.

SAVE: destroyed

BUF: set by procedure

COUNT: set by procedure

ADDRE: set by procedure

'Mode' returned in ACO is equal to MESS0 SHIFT -2. This procedure is used when the driver is ready for a new operation, and will delay the driver process until a relevant message arrives in its event queue.

If the word BUF in the process descriptor is set equal to -1 either by the programmer or the procedure RETURNANSWER, indicating return of a hard error in a previous answer, the procedure will automatically return all transput messages received, with the not processed status (1b15) and zero bytecount, until a control message is received. At the receipt of a control message, the BUF is reset and the procedure returns.

If a message is received with a sender different from a nonzero reserver word (RESER) in the process description, this message is returned automatically with illegal status, and the procedure will not return but continue to examine the event queue.

If a transput message is received with zero bytecount (MESS1 = 0) this message is returned with zero status and zero bytecount. The procedure thus saves the program from testing the special case of zero bytecount, and the standard instruction decrement and skip if zero (DSZ) can be used freely on the word COUNT in the process description.

If a control message (MESS0(15:15) = 0) is received, the message buffer address is saved in BUF, and COUNT and ADDRESS is set equal to MESS1 and MESS2 of the message buffer. Return is to the word following the call.

If a transput message (MESS0(15:15)=1) is received, the message buffer address is saved in BUF and COUNT and ADDRESS is set equal to MESS1 and MESS2. Return is to the second word following the call if the operation is input else to the third word.

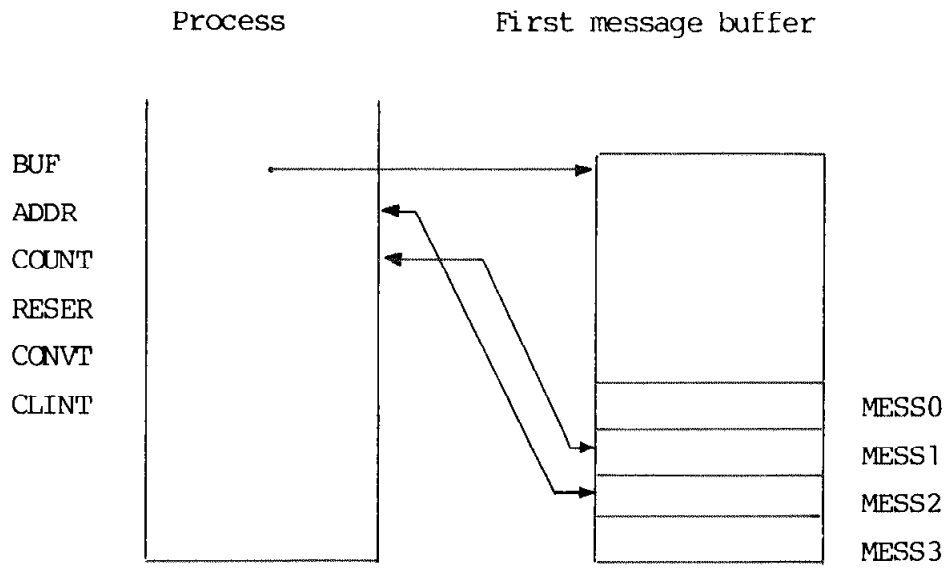


Fig. 13

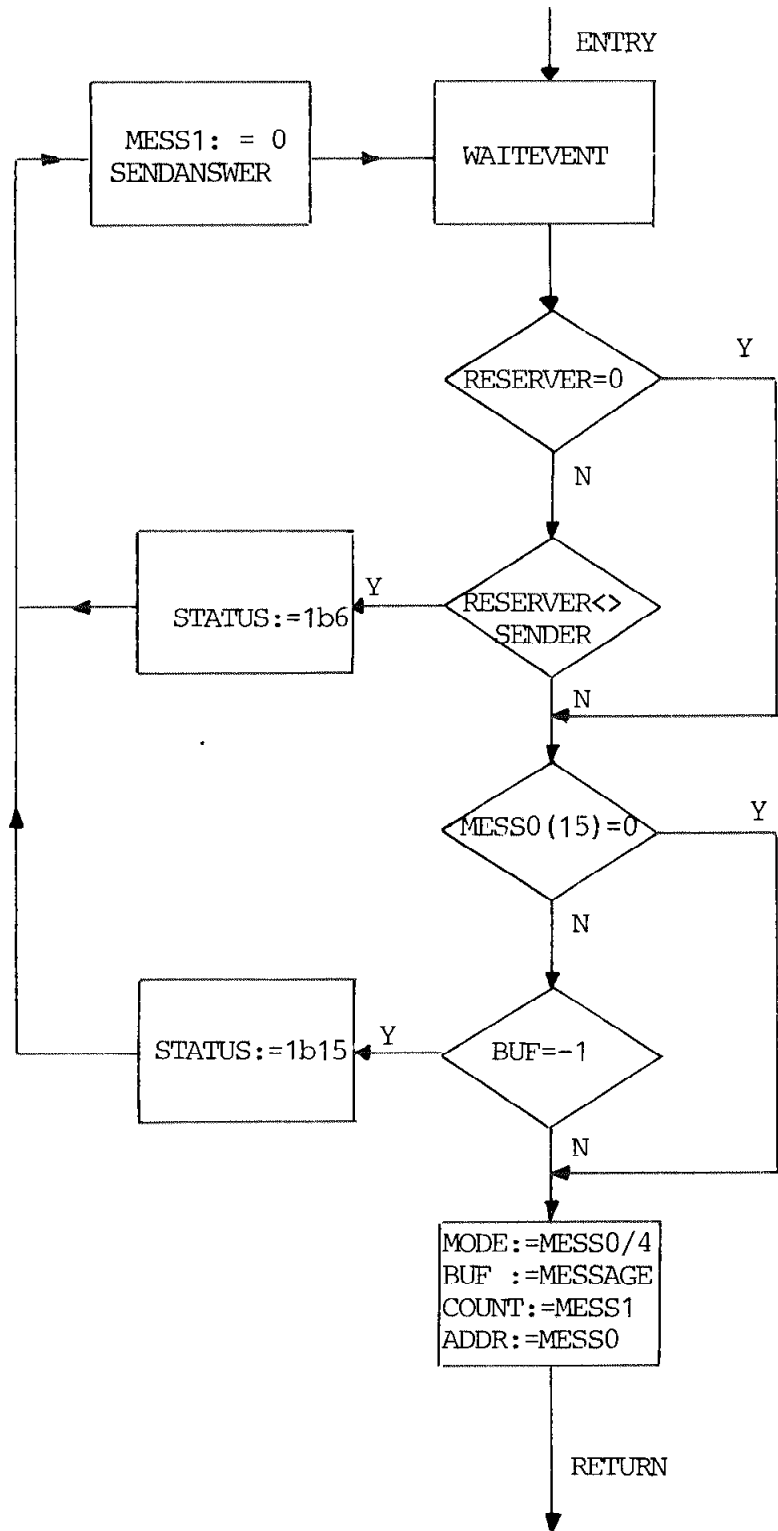


Fig. 14 Procedure NEXTOPERATION flow



4.4.2.2 Procedure WAITOPERATION.

	Call	return +0, +1	return +2	return +3, +4, +5
AC0	timer	unchanged	destroyed	mode
AC1	deviceno	unchanged	destroyed	count
AC2	cur	destroyed	destroyed	cur
AC3	-	cur	cur	buf

+0: Timer expired  
 +1: Interrupt received  
 +2: Answer received or the procedure has returned a buffer automatically.  
 +3: Control message received  
 +4: Input message received  
 +5: Output message received

SAVE: Destroyed  
 BUF: Set by procedure  
 COUNT: Set by procedure  
 ADDRES: Set by procedure

'Mode' returned in AC0 is MESS0 SHIFT -2. This procedure may be used by a driver process when it is necessary to wait for either device interrupt, timeout or a message.

The procedure will treat the received messages as described in the procedure NEXTOPERATION, i.e. check the reserver, transput-message with zero bytecount and reject action if BUF equals -1.

If the timer given as parameter in the call expires before any message or answer is received, the procedure will return to the address following the call. The procedure can be used without the interrupt return if the given devicenumber is set to zero, thus indicating a wait for the real time clock.

The timer is given in units of 20 ms.

When called, the process descriptor word BUF must be equal to 0, indicating wait for any buffer, or -1, indicating wait for any buffer but reject transput messages received.

If wait for a message following a buffer in the event queue is wanted, BUF must contain the address of this buffer.

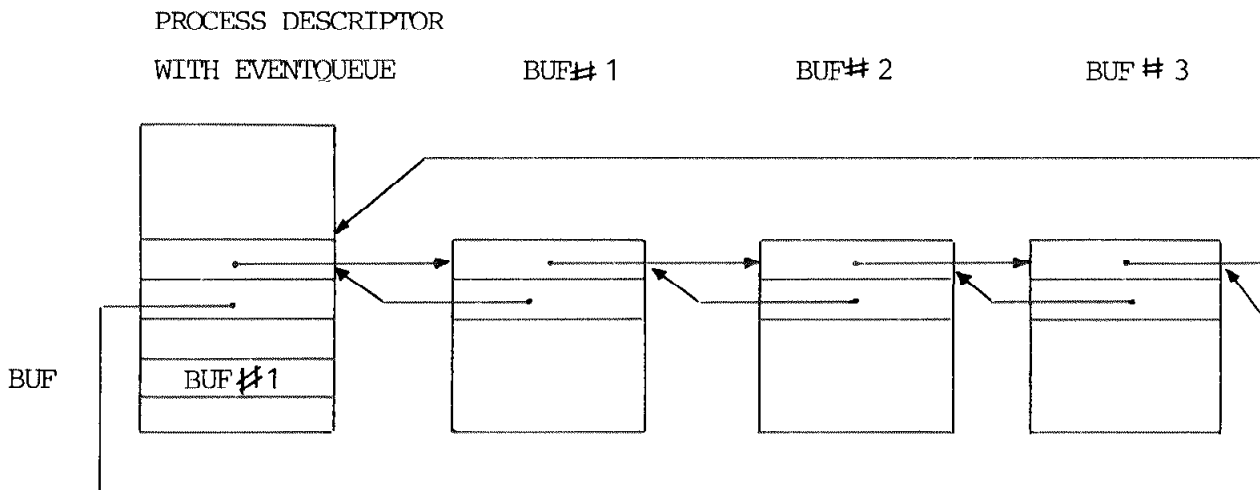


Fig. 15 If BUF in process descriptor contains the address of message buffer BUF#1, the address of message buffer BUF#2 is returned by procedure WAITOPERATION.

Return is made to the third word if the procedure returns a message with the not processed status or illegal status, or if it returns a transput message with zero bytecount.

If an answer is received, return is made to the same address and the answer can then be fetched by a call of the procedure WAIT-ANSWER on the relevant message buffer.

The address of the answered message buffer is not returned by the procedure.

Note that when the procedure is called, BUF must be equal to 0 or -1 or point to a message buffer in the event queue, otherwise a system breakdown may occur as there is no check on the parameters.

#### 4.4.2.3 Procedure RETURNANSWER.

4.4.2.3

	Call	Return
AC0	status	status
AC1	MESS2 to buf	destroyed
AC2	cur	cur
AC3	-	destroyed
SAVE:	Destroyed	
BUF:	Set to zero if no hardbits present in status (bit 0-2, bit 6-14) else -1.	
COUNT:	Undefined	
ADDRE:	Undefined	

The parameter status is set into MESS0 of the message buffer pointed out by BUF.

The number of bytes (MESS1) is calculated by subtracting the original byte address still remaining in the message buffer (MESS2) from the updated byte address found in ADDRE in the process description.

If one of the clean bits is set in status, the BUF word is set to -1, indicating that following transput messages must be returned with the not processed status (1B15).

The message buffer is returned to the sender process by means of the SENDANSWER procedure. The word BUF must point to a message buffer in the driver event queue, otherwise the call will cause system break down.

#### 4.4.2.4 Procedure SETRESERVATION.

4.4.2.4

	Call	Return
AC0	operation	operation/2
AC1	-	destroyed
AC2	cur	cur
AC3	-	destroyed

RESER: Set by the procedure.

If bit 15 of the operation (reservation bit of MESS0 when shifted 2 to the left by NEXTOPERATION) is nonzero, MESS1 of the message buffer given in BUF is examined. If this word is nonzero, the sender of the message is inserted as reserver of the process (RESER word), otherwise the word RESERVER is set to zero, indicating no reserver process.

4.4.2.5 Procedure SETCONVERSION.

4.4.2.5

	Call	Return
AC0	operation	operation/2
AC1	-	destroyed
AC2	cur	cur
AC3	-	destroyed

CONVT: Set by the procedure.

If bit 15 of the parameter operation (conversion bit of MESS0 when shifted 3 left by NEXTOPERATION and SETRESERVATION) is non-zero, the CONVT is set equal to MESS2 of the message buffer given in BUF.

4.4.2.6 Procedure CONBYTE.

4.4.2.6

	Call	Return
AC0	byte	newbyte
AC1	-	destroyed
AC2	cur	cur
AC3	-	destroyed

The returnvalue 'newbyte' is the bytevalue found at relative location 'byte' in the conversion table specified by the process descriptor word CONVT. The conversion table address must be a byteaddress, and the new bytevalue is fetched as:

```

.
.
LDA      1      CONV1,2  ;
ADD      0,1    ;
GETBYTE  ;
.

```

If the conversion table address is zero, the procedure is dummy and the byte returned is the same as given in the call.

#### 4.4.2.7 Procedure GETBYTE.

4.4.2.7

	Call	Return
AC0	-	byte
AC1	byte addr	byte addr
AC2	-	cur
AC3	-	destroyed

Fetch the byte at the given byte address. The wordaddress of the word containing the byte is taken as  $(\text{byte addr})/2$ , and if bit 15 is set in the byte address, the rightmost byte is delivered, else the leftmost.

#### 4.4.2.8 Procedure PUTBYTE.

4.4.2.8

	Call	Return
AC0	byte	unchanged
AC1	byteaddr	byteaddr
AC2	-	cur
AC3	-	destroyed

Store the byte value given as parameter at the given byteaddress. The byteaddress is interpreted as in the GETBYTE procedure. The remaining part of the word containing the byte is unchanged if the byte given is in the range 0 to 255.

```
; Procedure MOVEBYTES
```

```
;      call          return
```

```
; AC0    count      destroyed
```

```
; AC1    to-addr    destroyed
```

```
; AC2    from-addr  cur
```

```
; AC3    -          destroyed
```

```
; The procedure moves 'count' byte to the byte address to-addr
```

```
; from the byte address from-addr
```

```
;
```

```
MBYTE:  STA      3      LINK      ; MOVEBYTES:
        STA      1      TOADDR    ;
        STA      2      FRADDR    ;
        STA      0      BCOUNT   ; REPEAT
LOOP:   LDA      1      FRADDR    ;
        GETBYTE                                ; GETBYTE (FRADDR, BYTE);
        LDA      1      TOADDR    ;
        PUTBYTE                                ; PUTBYTE (TOADDR, BYTE);
        ISZ                                ; TOADDR:=TOADDR+1;
```

```

ISZ          FRADDR    ; FRADDR:=FRADDR+1;
DSZ          BCOUNT    ; COUNT:=COUNT-1
JMP         LOOP      ; UNTIL COUNT = 0
JMP         LINK      ; RETURN

```

```

LINK:      0          ;
BCOUNT:    0          ;
TCADDR:    0          ;
FRADDR:    0          ;

```

Example 16 Use of PUTBYTE and GETBYTE

#### 4.4.2.9 Procedure MULTIPLY.

4.4.2.9

	Call	Return
AC0	op1	result(0:15) high part
AC1	op2	result(16:31) low part
AC2	-	cur
AC3	-	destroyed

SAVE: Destroyed

Computes the unsigned double length product of the two single length operands, result:=op1\*op2. The result is 32 bits long.



4.4.2.10 Procedure DIVIDE.

4.4.2.10

	Call	Return
AC0	dividend	quotient
AC1	divisor	divisor
AC2	-	cur
AC3	-	remainder
SAVE:	Destroyed	

Performs a short division of the 16 bit dividend extended with zeroes by the divisor, giving single length quotient and remainder.

```
quotient:= dividend//divisor
remainder:= dividend REM divisor
```

Division with a zero divisor is not checked and delivers unpredictable results.

4.4.2.11 Procedure SETINTERRUPT.

4.4.2.11

	Call	Return
AC0	-	destroyed
AC1	deviceno	deviceno
AC2	-	unchanged
AC3	-	destroyed

The procedure is executed with interrupt disabled. It includes the process as user of the device. The device given by 'deviceno' is cleared by a NIOC <deviceno> instruction.

Any interrupt request from deviceno will cause a call of the interrupt procedure defined by CLINT in the process descriptor.

A standard system procedure is CLEAR, which executes a NIOC instruction.

```

LOOP:      .           ; LOOP:
           .
           .
           .
           .
NEXTOPERATION      ; NEXTOPERATION (MODE);
JMP      CONTR      ; +0: GOTO CONTROL;
JMP      ILLEGAL     ; +1: GOTO ILLEGAL;
.         ; +2: OUTPUT
.
.         ;
.
.         ;
JMP      OK          ; GOTO OKSTATUS;

CONTR:     SETRESERVATION      ; CONTROL:
           SETCONVERSION      ; SETRESERVATION;
           MOVZR      0,0      SNC      ; SETCONVERSION;
           JMP      OK          ; IF TERMINATION THEN
           .         ; BEGIN
           .         ; .
           .         ; .
           .         ; .
           .         ; END;
           .         ; GOTO OKSTATUS

ILLEGAL:   LDA      0          SILLE  ; ILLEGAL:
           JMP      RET        ; STATUS:= ILLEGAL

OK   :     LDA      0          .0     ; OKSTATUS:
RET   :     RETURNANSWER      ; STATUS:= 0
           JMP      LOOP      ; RETURNANSWER;

```

## 4.5 Driver Requirements.

4.5

In the previous sections a number of driver standards have been mentioned. These standards are fulfilled if the MUS utility procedures are used and if the messageformats described are accepted by the drivers and the answers returned are of standard layout.

When drivers are designed, it should be noted that I/O instructions are not checked by the system and it is then the programmers responsibility that other system components are left untouched in all situations. Therefore the use of IORST,MSKO instructions is not allowed and special care must be taken if INTDS and INTEN instructions are used, as the system performance can decrease rapidly when too much driver code is executed in disabled mode.

It should also be noted that the system offers no memory protection and even small programming errors can have disastrous effects on the whole system.

### 4.5.1 Break Action.

4.5.1

The driver can, as all processes, be breaked with a number of causes, e.g. if a message buffer is removed from its event queue by the monitor after a process kill or at system start up after power failure.

Whenever the driver is breaked it must stop all operations in progress on the device and clear the device table entry for interrupt indications. This can normally be done by call of the procedure SETINTERRUPT, which will clear both the device and the device table.

If the break cause is not power failure, the reserver must be cleared, hereby releasing the driver e.g. on request from the operator.

If continued operation on the device is impossible without loss of data, this must be indicated in the answer to the next message received, and the user process must then take care of the error recovery. Continued operation is normally only possible on devices, which supports repetition of the previous operation, and can not be done on other devicetypes, as the process state before the break can not be reestablished.

#### 4.5.2 Device handling.

4.5.2

The device handling depends heavily on the actual device, but all synchronization with the device must be achieved by means of interrupts, as busy waiting will use too much CPU-time in a multi-programmed system.

The devices are normally designed such that interrupts are only requested after the device has been started by the programmer with a S or P pulse.

It is a general rule that interrupt requests are awaited only for a maximum device specific time, and if it is not received within this time, the status timeout is indicated and the device is cleared to prevent an interrupt after the timeout (procedure SETINTERRUPT).

The normal procedure for device handling is then:

```

      .                               ; SETUP DEVICE PARAMETERS
      .
      .                               ;
      .                               ;
      NIOS                DEVNO      ; START (DEVNO)
      .                               ;
      LDA      1          .10        ;
      LDA      2          .DEVNO     ;
      WAITINTERRUPT                ; WAITINTERRUPT (DEVNO,
      .                               ; 10*20 ms)
      JMP      TIMO       ; +0: GOTO TIMEOUT
      .                               ;
      .
      .DEVNO:  DEVNO                ;
      .                               ; TIMEOUT:
      TIMO:    SETINTERRUPT          ; SETINTERRUPT (DEVNO)
      LDA      0          STIMER     ; STATUS:= TIMEOUT;
      .
      .
      .
      .

```

Example 17

#### 4.6 Driver Incarnations.

4.6

If a number of devices of the same type are connected to the system, the code to handle the devices will only differ on few points. To save memory space, the driver for the first incarnation can be coded reentrant, which means that other incarnations of the same driver need only to be the process descriptor executing the same code as the first driver.

The driver can be coded reentrant if all device specific variables, constants and instructions are fetched relative to the pro-

cess description, and placed in the process descriptor following the words used by the utility procedures. This can be seen in example 18. (Note that I/O instructions must also be placed in the process descriptor, as the device number is an integral part of the instruction).

The connection of a second process description to the program in question can be done using a small program, which searches in the program chain for the reentrant program and, when found, reassigns the process break address, interrupt procedure address and jumps to the program start.

Appendix D contains as an example a listing of the paper tape punch driver and the second paper tape punch driver.

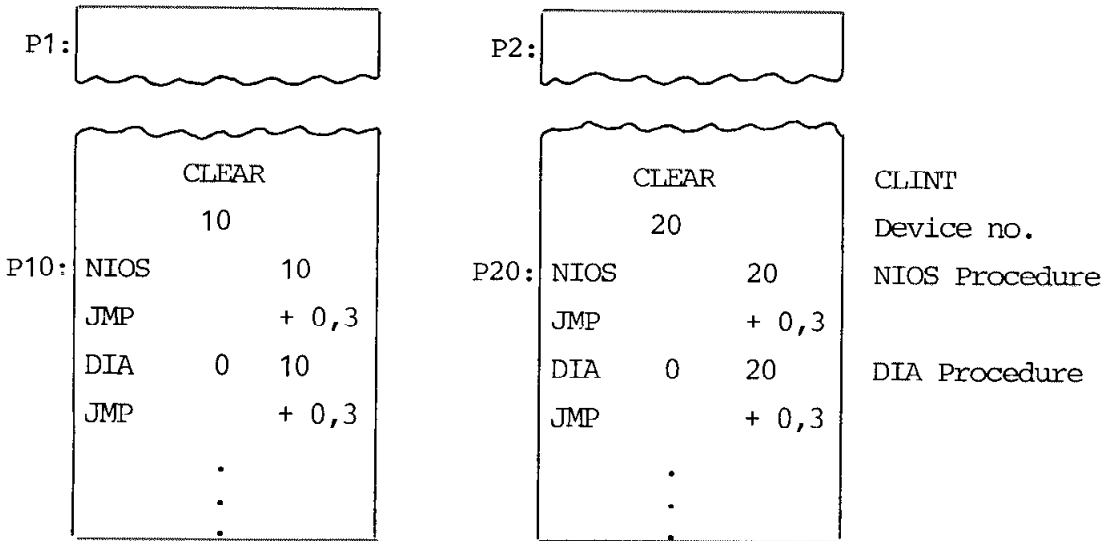
Reentrant  
driver program

use of i/o instructions

```

DNIOS = P10 - P1      ;
LDA     2    CUR      ;
JSR          DNIOS,2  ;
    
```

First process, Process name P1. Second process, Process name P2 .



Example 18 Process incarnations.

## 5. BASIC I/O HANDLING.

5.

### 5.1 General Description.

5.1

The I/O operations in the MUS system is based on dedicated device handling processes, called drivers.

All user requests for operations on a specific device are transferred to the corresponding driver process by means of the fundamental monitor functions SENDMESSAGE and WAITANSWER. As previous described, the information to the driver is transferred in a message buffer containing 4 16 bits words, which is interpreted by the driver process.

In order to ease the use of the driver process and to simplify the possibility of using multibuffered input/output, a number of reentrant I/O procedures have been included in the MUS-system. The procedures work on a data structure called zone or filedescriptor.

The fundamental concept that all communication is done by means of messages, is not broken, and the basic I/O procedures must be viewed as the first level on top of the monitor functions. Correspondingly, the character and record input/output procedures are the next level on top of the basic I/O procedures.



## Basic I/O procedures.

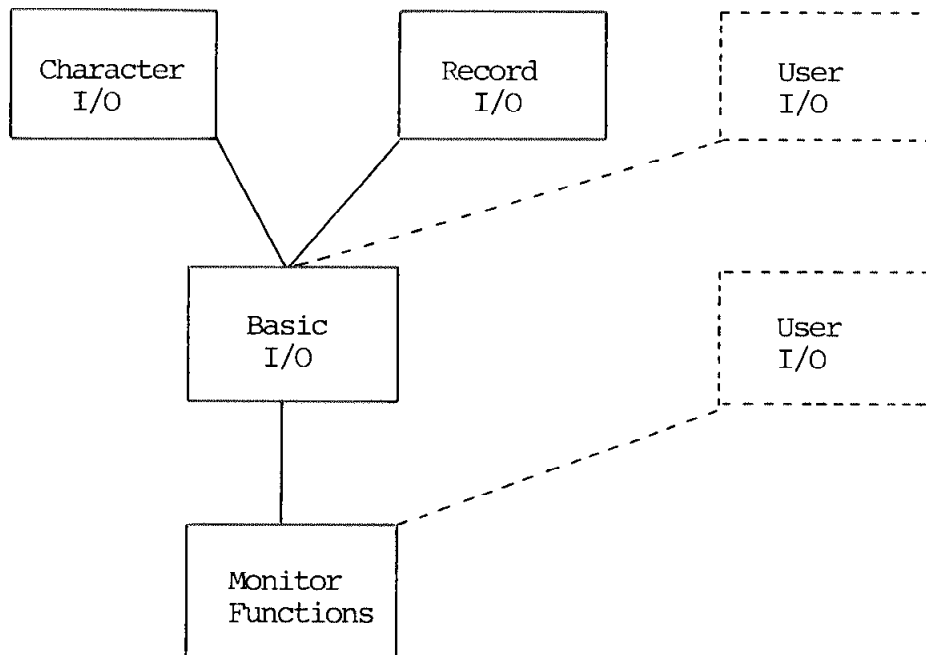


Fig. 16 The procedure hierarchy in the MUS system.

It is then possible for the user to build his own dedicated procedures at all levels over the monitor functions, if the offered procedures do not fit the actual usage.

All MUS high level procedures work on a zone, which is a collection of information and data buffers, normally pointed out by the programmer by its wordaddress.

The zone contains three parts:

1) The zone descriptor

It contains informations about the document and the device that holds it. It holds furthermore information concerning the data accessible to the user, i.e. the storage area which

is not involved in device input/output and thus can be fetched or filled by the program.

2) The share descriptors

The share descriptor holds information about the data buffer associated with it, and the state of data, i.e. if the data is or has been processed by the driver. It holds moreover a copy of the message, which requested the data transfer, consequently repetition of the transfer is possible if it was rejected because of an error.

3) The buffer area

The area from/to which the actual data transfer is done by the driver on request from the I/O procedures.

The memory layout of a zone with two shares is then:

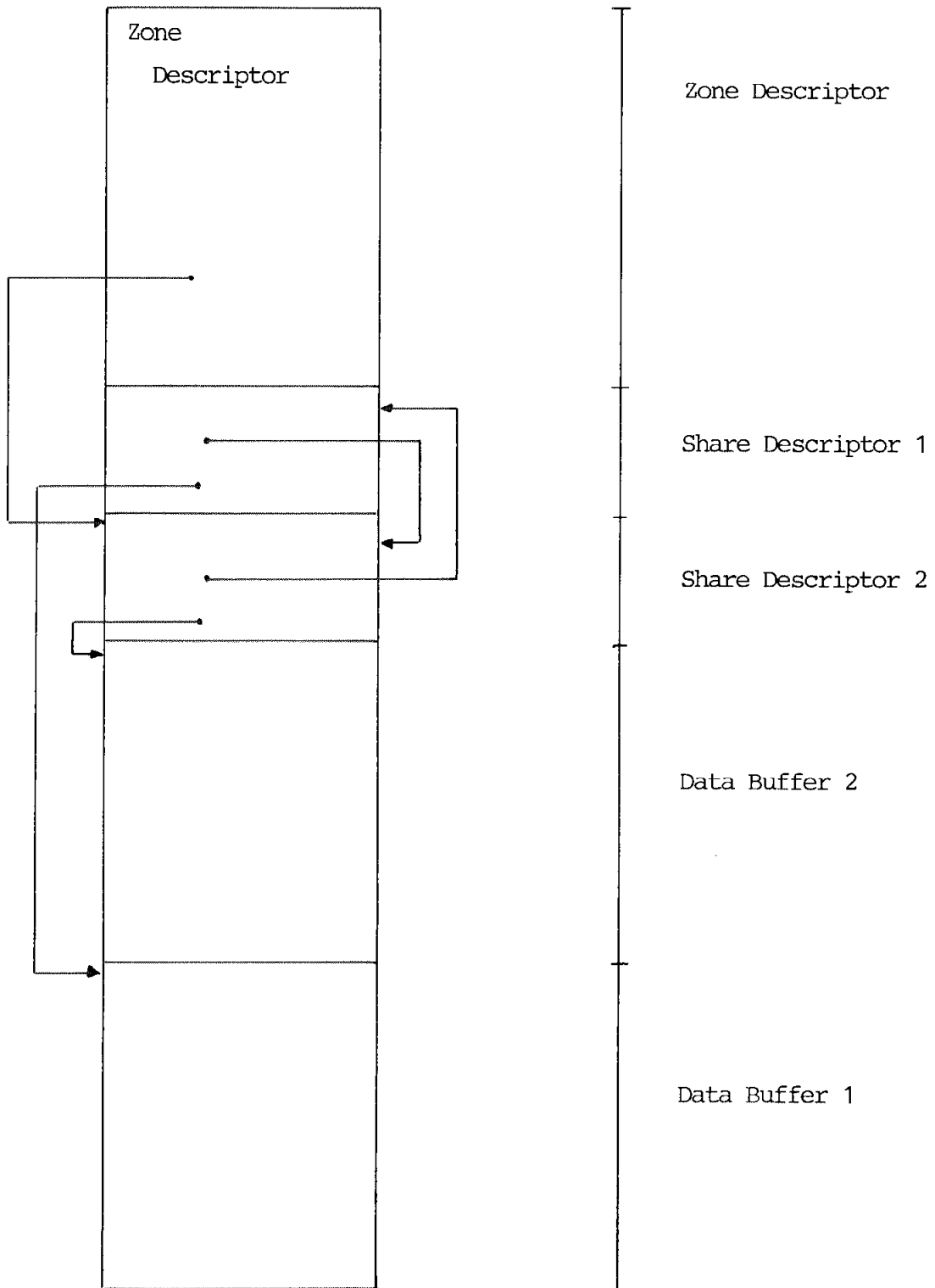


Fig. 17

On fig. 17 the following references can be seen:

- 1) In the file descriptor a pointer to the share involved in user I/O. The pointer is updated by the I/O procedures.
- 2) In the share descriptor a pointer to the data buffer connected to the share descriptor. The pointer is static.
- 3) The share descriptors are linked cyclically. In this way the next share descriptor and its associated data buffer can be selected by the pointer mentioned in 1), when the data buffer has been filled with data and a message has been sent to the driver requesting the transfer to the document. The pointers are static.

## 5.2 Zone format.

5.2

The zone is in all procedure calls identified by the word address of its first location, and the words in the zone descriptor can be fetched by the user by means of this address and the displacements below recognized by the assembler:

ZNAME      The document name (drivename) 6 characters, 3 words.

This process will receive all messages generated by the I/O procedures.

SIZE        Size of zone descriptor area.

ZMODE      The operation.

This word is used as operation code in all transput messages to the driver. All central procedures transfer the first byte of this word as the first byte in the operation code to the driver.

If the last two bits are equal to 11 the operation is output. If the are equal to 01 the operation is input.

The word is set by the procedure OPEN.

ZKIND The kind used for errorhandling and initialization actions (see later).

ZMASK The giveupmask.

This mask is compared with the status received from the driver. If common bits are set, the defined giveup procedure is called.

ZGIVE The address of the user giveup procedure. The procedure is called, if an error is returned and the errorbit is present in the giveupmask

ZFILE Used for file position with some document kinds.

ZBLOCK Used for block position with some document kinds.

ZCONV Conversion table address.

This address is transferred to the driver in an OPEN call.

ZBUFF Buffer address.

Used by MUSIL interpreter.

ZSIZE Size of buffer.

Used by MUSIL interpreter.

ZFORM Format code for records.

Used by Record I/O procedures.

ZLENG Length of current record.

Used by Record I/O procedures.

ZFIRS First of record.

Used by record and Character I/O procedures. The byte address of the first byte in the current record.

ZTOP Top of record.

Byteaddress of the first byte after the current record.

ZUSED Used share.

Word address of the currently used share descriptor.  
See fig. 17.

ZSHAR Share length.

The number of bytes in each data buffer.

ZREM The number of bytes in the current data buffer not yet processed, or the number of bytes left in the data buffer to output.

Z0 Status.

The status returned on the last checked transfer. Only defined in the giveup procedure.

The zone contains a number of auxiliary words, exclusively used by the procedures.

The number of these are given by the assembly constant ZAUX, which also includes the above mentioned Z0 word.

The total size of a standard zone descriptor is given by the

assembly constant Z.

### 5.3 Share Descriptor Format.

5.3

A share descriptor is identified by the word address of its first location. The share descriptor of the current share can be found in the zone descriptor word ZUSED.

The share state can then be found as:

```

; AC2 = zone address

LDA      3      ZUSED,2  ;
LDA      0      SSTAT,3  ;

```

SOPER    Operation to the driver  
         MESS0 in the message generated

SCOUN    Number of bytes transferred  
         MESS1 in the message generated

SADDR    Address of the data buffer  
         MESS2 in the message generated

SSPEC    Special  
         MESS3 in the message generated

These four words are used as the message to the document and are unaltered when the answer is received. In this way it is possible to repeat the operation if the transfer was unsuccessful.

SNEXT    Next share  
         The wordaddress of the next share descriptor. All share descriptors are linked cyclically.

SSTAT State of share

Value

0 The data buffer can be used.

<>0 The data buffer is involved in transfer to the device. The word contains the address of the message buffer carrying the transfer request.

SFIRS First byte address.

The byte address of the first byte in the associated data buffer. The address is transferred to ZTOP in zone descriptor, when the next share is made available to the user after input of data or before output of data.

The size of the standard share descriptor can be found as the standard assembler constant SSIZE.



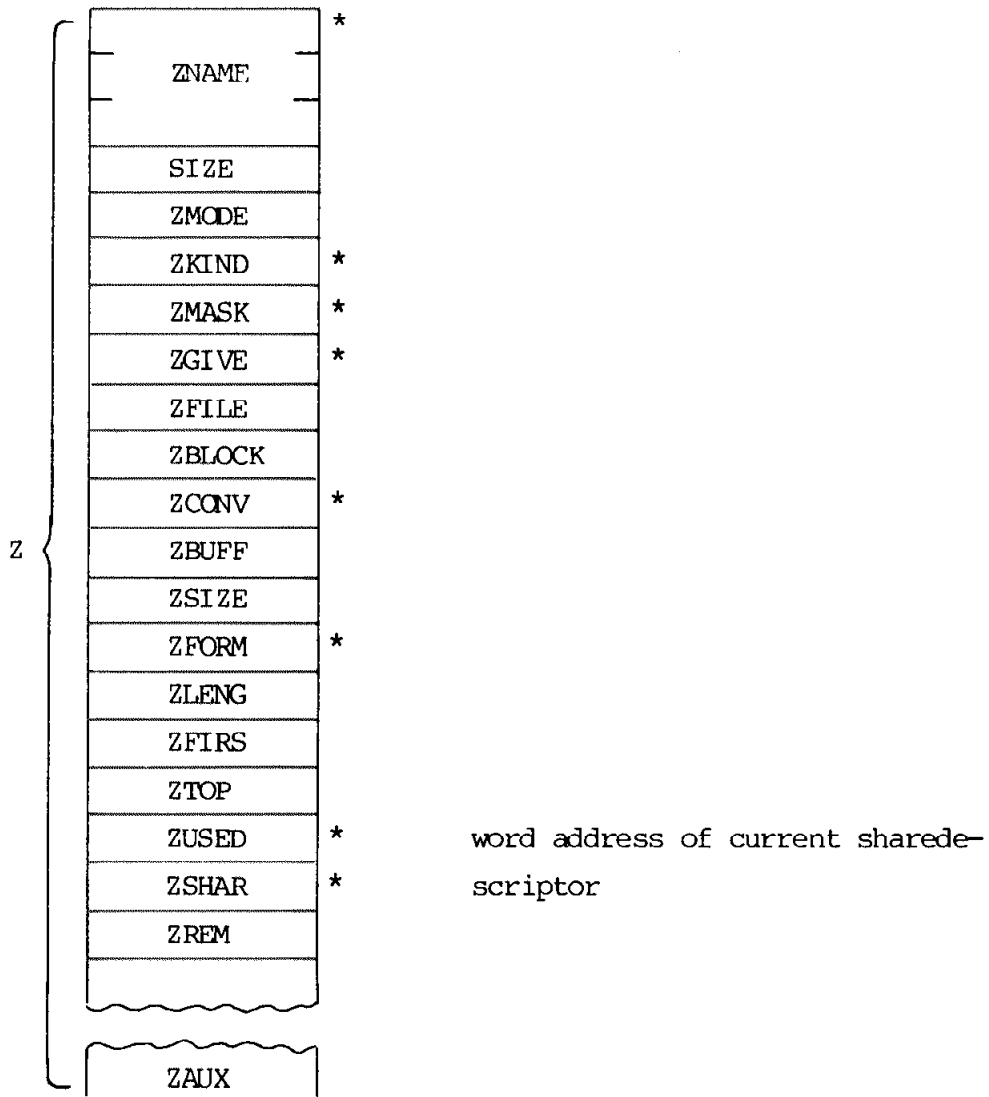


Fig. 18 zone layout. All words marked with \* must be set by the programmer before use.

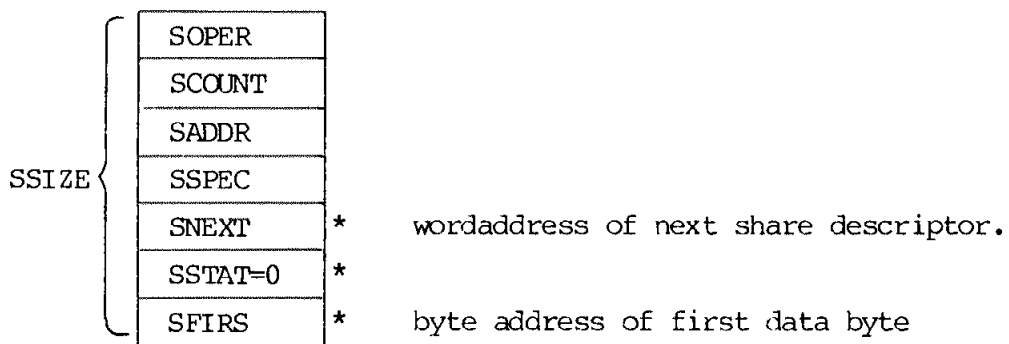


Fig. 19 share descriptor layout. All words marked with \* must be set by the programmer before use.

#### 5.4 Zone Setup.

5.4

When the zone data structure is used in the assembler, a number of words in the zone descriptor and share descriptor must be pre-defined by the programmer.

##### 5.4.1 Zone Descriptor.

5.4.1

The document name must be initialized.

The kind must be set. (See chapter 5.5)

The giveupmask must be set.

The giveupprocedure address must be defined.

The conversion table must be defined as a byteaddress (0 if no conversion).

The record format must be defined. (see chapter 6.)

The record length must be defined if fixedlength formats are going to be used.

The address of the used share descriptor must be defined.

The share length (data buffer size) must be set as a number of bytes.

5.4.2 Share Descriptor.

5.4.2

All sharedescriptors must be linked cyclically in the words SNEXT.

Sharedescriptor status must be set to 0.

The first byte in the associated data buffer must be set in the sharedescriptor word SFIRS.

5.4.3 Message Buffer Pool Size.

5.4.3

A number of message buffers must be set up too. If the process uses  $n$  zones, each with  $N_i$  shares, the number of message buffers must be

$$1 + \sum_{i=1}^n (N_i - 1)$$

This number is sufficient if all I/O is done with the Basic I/O procedures. Else all user generated messages must be added.

If the zones are used in a coroutine environment (see [3]), the number of message buffers must be

$$\sum_{i=1}^n N_i$$

```

T0001:                                ; LIST ZONE DESCRIPTOR
.TXT  .$.LPT <0>.                    ; NAME
                                         ; SIZE
                                         ; MODE
                                         ; KIND
.TDX  2                                ; MASK
      1110001111111110
.TDX 10
      P0099                            ; GIVEUP
      0                                ; FILE
      1                                ; BLOCK
      0                                ; CONVERSION
      T0061                             ; BUFFER
      T0069-T0061                       ; SIZE OF BUFFER
      0                                ; FORMAT
      0                                ; LENGTH
      T0062                             ; FIRST
      T0062                             ; TOP
      T0061                             ; USED SHARE
      512                               ; SHARE LENGTH
      512                               ; REMAINING
.BLK  ZAUX                             ; AUXILIARY
T0061:                                ;LIST SHARE DESCRIPTOR
      0                                ; OPERATION
      0                                ; COUNT
      0                                ; ADDRESS
      0                                ; SPECIAL
      T0061                             ; NEXT SHARE
      0                                ; STATE
      T0062                             ; FIRST SHARED
T0062=  .*2                            ;FIRST SHARED:
.BLK   512/2                           ; MAKE ROOM FOR SHARE
T0069:                                ;TOP OF BUFFER:

```

Example 19.

5.5 Document identification.

5.5

A document is a physical medium, which is able to contain data and which is mounted on a device.

In the zone descriptor, the document in question is described by:

- 1) The name of the driver process, which controls the device.
- 2) The mode, which describes the operation requested at the driver when data is transferred to the document. E.g. if characters should be output with or without parity check information.
- 3) The kind of the device.  
An array of bits, which describes, how transfer errors should be handled and special actions on each transfer or control operation.

The following bits are at present defined:

Bit 15:           character oriented

Set if the device transfers characters one by one. If an error is returned before output of the whole data block and the transfer is repeated in the users giveup procedure, only the characters not yet output are requested to be transferred. Examples are papertape punches and lineprinters used in unformatted mode.

bit 14: Block Oriented.

Used, if the datablocks are transferred as a unit to/from the device. If repetition is necessary, the whole block must be transferred again.

Examples are cardreaders and punches, magnetic tapes, discs and line printers used in formatted mode.

bit 13: Positionable.

Set, if positioning has any effect. Must be set, if the document has to be repositioned before a datatransfer is repeated.

Examples: magnetic tapes and discs.

bit 12: Repeatable.

Set, if automatic repetition can be performed by the system on special status bits.

Examples: magnetic tapes and discs.

bit 11: Catalog file.

Must be set, if the filesystem is used. Automatic creation and removal of catalog area processes is performed and any input/output is requested with 512 bytes buffer size. (See chapter 8).

bit 0: Coroutine bit.

Set, if the process uses the RC 3600 Coroutine Monitor (see [3]). Call of Basic I/O procedures will not delay the whole process but only the calling coroutine by a call of

the procedure CWANSWER.

## 5.6 Exception Handling.

5.6

In the input/output procedures the user may select certain status bits which, if set in the answer to a message to the driver, will transfer control to a user defined procedure.

Before the user procedure is called, a number of automatic error recovery actions are performed, depending on the zone kind.

The statusbits returned from the driver are interpreted as:

	name	error type
bit 0	disconnected	hard error
bit 1	offline	hard error
bit 2	device busy	repeat error
bit 3	device bit 1	informative
bit 4	device bit 2	informative
bit 5	device bit 3	informative
bit 6	illegal	hard error
bit 7	end of file	hard error
bit 8	block length error	hard error
bit 9	data late	repeat error
bit 10	parity error	repeat error
bit 11	end medium	hard error
bit 12	position error	hard error
bit 13	driver missing	hard error
bit 14	timeout	hard error

If the error is classified as a repeat error and the zone kind is repeatable, the operation is repeated up to 5 times, before the users giveup procedure is called. If the kind word has the positionable bit set, the position is adjusted to the file count specified in ZFILE and the blocknumber ZBLOCK-1 before the

repetition is performed.

When repetition takes place, the whole data block is requested to be input/output again, but if the kind is characteroriented, only the part of the buffer not yet output is requested to be transferred.

An operation is repeated a maximum of 5 times. If it is still erroneous, it is classified as having a hard error.

When a hard error is detected, the users giveup mask is compared with the status received. If one of the hard error bits present in status is not set in the giveup mask, the process is broken with errorcode = 5, and the status in accumulator 1.

If however the giveup mask bit 15 is set, the giveup procedure is called unconditionally.

After the standard check, the status is compared with the user giveup mask. If any bits are common, the giveup procedure is called.

#### 5.6.1 User Giveup.

5.6.1

	call	return (normal return, repeat share)
AC0	-	-
AC1	status	status
AC2	zone	zone
AC3	return address	return address

The Z0 word in the zone is the status of the call.

The ZREM word is the actual number of bytes input or the share-length, if the operation was output.

The ZTOP word contains the byteaddress of the first byte trans-



ferred.

Depending on the giveup action, return can take place in three different ways:

1) A jump to the main program.

The error is accepted and the transfer is unsuccessful.

2) Return to the given return address.

The error is accepted and the transfer is unsuccessful. Return is made to the procedure which caused the transfer.

Please note that character and record I/O procedures will not return with zero bytes transferred, i.e. the procedure call is repeated until any bytes have been transferred, and this can easily cause loops in the program.

3) The operation can be repeated by a jump to the repeataction.

This is done with the predefined instruction `.REPEATSHARE`.

Please note that if the repetition of the operation is successful, the final return is made to the procedure which caused the faulty transfer.

I/O effecting procedures on the same zone must not be called in the giveup action, if return to the calling I/O procedure is wanted by means of `.REPEATSHARE` or via the given return address.

When the basic procedures handle an answer, the status word is augmented with the following bits:

bit 13:           Set, if the receiver process is not loaded.

bit 13:           Set, if a control operation with command  
(14:15) = 10 is checked.

5.6.2 Giveup Procedure Example.

5.6.2

The giveup procedure is used with a zone, to which input is made by the INCHAR procedure. Errors indicating end of file or end of medium are converted to input of the character EM (25):

```

GIVE:   LDA      0      ZREM,2   ; IF ZONE.ZREM <>0 THEN
        MOV      0,0    SZR      ;          RETURN
        JMP                      +0,3 ;
        LDA      0      EMSTAT   ; IF STATUS AND 1b7+1b11
        AND      1,0    SNR      ; <>0 THEN
        JMP                      HARDERROR ; BEGIN
        STA      3      LINK      ;
        STA      2      ZONE      ; PUTBYTE (EM,ZONE.ZTOP);
        LDA      1      ZTOP,2   ;
        LDA      0      .EM      ;
        PUTBYTE                      ;
        LDA      0      .1      ; ZONE.ZREM:=1;
        LDA      2      ZONE      ;
        STA      0      ZREM,2   ; STATUS:=0;
        LDA      3      LINK      ;
        SUB      1,1    .        ; RETURN;
        JMP                      +0,3 ; END;

HARDER: .
        .

LINK:   0          ; SAVE LINK
ZONE:   0          ; SAVE ZONE
EMSTAT: 1b7+1b11  ; EOF, EM STATUS
.EM:    25         ; CHARACTER EM

```

5.7 Repeat Actions.

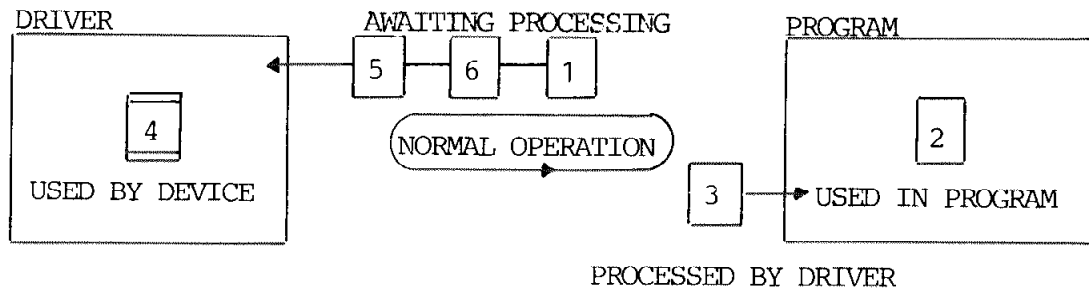
5.7

When a driver returns a harderror, it enters a reject state, in which all transput messages are returned with the non-processed status (1b15). This state is only changed, if a control operation is received. The basic I/O procedures will repeat a message with

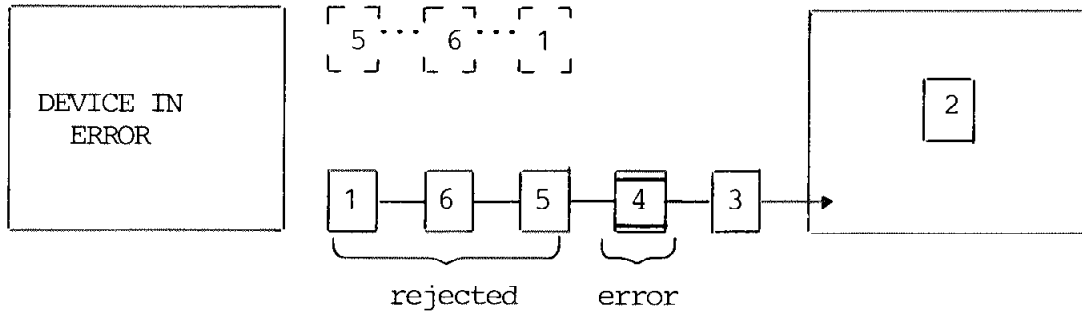
the nonprocessed status automatically after a harderror status. These procedures concerning harderrors, guaranties that the right sequence of messages is maintained in the drivers event queue.

As an example, see the following drawings of multibuffered error recovery:

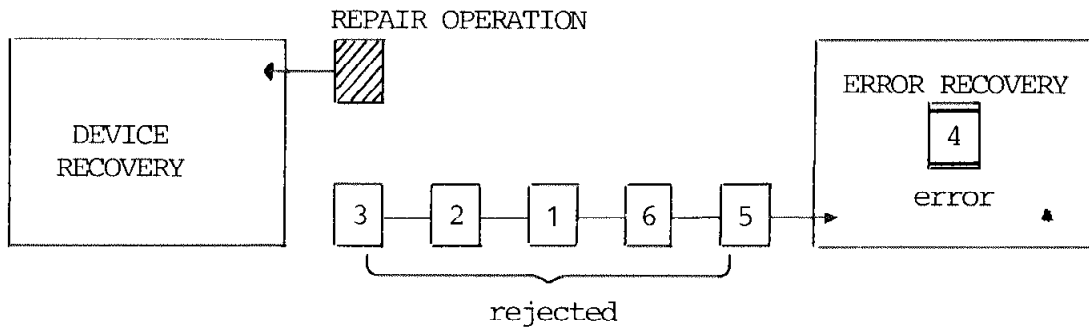
ERROR IS DETECTED AT THE DRIVER



AFTER ERROR



ERROR RECOVERY



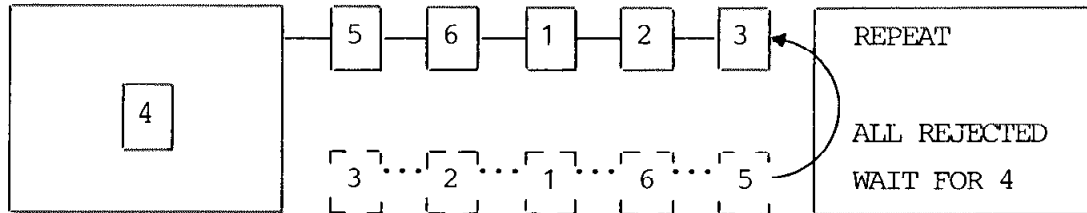
DEVICE READY

Fig. 20.

5.8 Basic I/O Procedures.

5.8

The basic I/O procedure calls are predefined in the system assembler. The procedures are thus called by their names, and the return address is then given by the CPU in AC3.

If procedures must be called with another return address, the same names can be used with a leading character '.'.

Example:

Normal use:

```
LDA      0      .3      ;
LDA      2      ZONE    ;
OPEN                                           ;
                                           ; RETURN HERE
```

Special use:

```
LDA      0      .3      ;
LDA      2      ZONE    ;
LDA      3      RETAD   ;
.OPEN                                         ;
RETAD:   LABEL
        .
        .
        .
LABEL:                                       ; RETURN HERE
```

When basic I/O transfer procedures return, the following values have been updated:

ZBLOCK is set equal to MESS3 of the answer if the zone is set to be positionable and if status <1b6, 1b13 and 1b15.

If the zone is not defined to be positionable, ZBLOCK is incremented with one for each block successfully input/output.

ZFILE is set equal to MESS2 of the answer if the zone is set to be positionable and the status <1b6, 1b13 and 1b15.

ZREM if the mode is input, ZREM is set equal to the number of bytes input or, if mode is output, equal to ZSHAREL.

ZTOP is set to the byteaddress of the first byte in the current data buffer. The address is fetched from SFIRST in the used share.

### 5.8.1 Initialization Procedures.

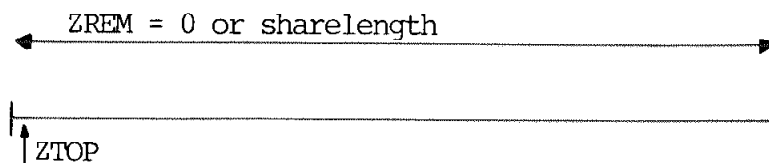
5.8.1

#### 5.8.1.1 Procedure OPEN.

5.8.1.1

	Call	return
AC0	operation	destroyed
AC1	-	destroyed
AC2	zone	zone
AC3	-	destroyed

The operation is placed in the modeword (ZMODE) of the zone descriptor. The remaining bytes of the zone (ZREM) is initialized either to zero, if the operation is input, or to sharelength (ZSHAREL) if operation is output. The top of the zone points to the first byte in the data buffer.



To initialize the transfer, a control message is sent to the process specified by ZNAME. The message requests reservation and conversion.

Message generated:

MESS0	ZMODE (0:7)	14 <sub>g</sub>
MESS1	<> 0	
MESS2	ZCONV	
MESS3	-	

5.8.1.2 Procedure SETPOSITION.

5.8.1.2

	Call	Return
AC0	block	destroyed
AC1	file	destroyed
AC2	zone	zone
AC3	-	destroyed

This procedure first waits for all transfers by means of the procedure WAITZONE. Then a position message is sent to the process with the name given in ZNAME.

ZREM and ZTOP are defined as for OPEN.

Message generated:

MESS0	ZMODE(0:7)	40 <sub>g</sub>
MESS1	-	
MESS2	FILE	
MESS3	BLOCK	



5.8.1.3 Procedure WAITZONE.

5.8.1.3

	Call	Return
AC0	-	unchanged
AC1	-	unchanged
AC2	zone	zone
AC3	-	destroyed

Terminates the current activities of the zone. If the zone is opened for output, the last block is output as if OUTBLOCK was called, and then all pending transfers are awaited and checked, i.e. the giveupprocedure may be called.

If the zone is opened for input, all transfers are awaited, but not checked.

If the last block is output, the message generated is:

MESS0	ZMODE	
MESS1	ZSHAREL - ZREM *	* ZSHAREL if
MESS2	ZUSED.SFIRST	ZKIND(11)=1
MESS3	ZBLOCK	

ZREM and ZTOP are defined as for OPEN.

5.8.1.4 Procedure CLOSE.

5.8.1.4

	Call	Return
AC0	-	destroyed
AC1	release	destroyed
AC2	zone	zone
AC3	-	destroyed

The zone is first set neutral by means of WAITZONE.

If the zone is opened for output, a termination, and if 'release' is nonzero also a release-reservation and disconnection control message is sent to the process with the name given in ZNAME.

If the zone is opened for input, a sense, and if 'release' is nonzero a release-reservation and disconnection control message is sent.

ZMODE is then set to zero and the control message is awaited answered by the WAITZONE procedure.

ZMODE \ RELEASE	output	input
0	termination	sense
1	release termination disconnect	release disconnect

Message generated:

MESS0	ZMODE(0:7)	OP
MESS1	0	
MESS2	-	
MESS3	ZBLOCK	

OP is specified in the table above.

All zonepointers and values are undefined.

Example:

```

LDA      2      ZONE      ;
LDA      0      .3       ;
OPEN                                ; OPEN(ZONE 3);
LOOP:   LDA      0      BLOCK ; REPEAT
LDA      1      FILE     ; SETPOSITION(ZONE,FILE,
                          ; BLOCK);
SETPOSITION                          ;
.                                     ; DATA PROCESSING
.                                     ;
                          ; FILE:=FILE+1;
SUB      1,1    ;
CLOSE                                ; CLOSE(ZONE,FALSE);
LDA      0      .3       ; OPEN(ZONE, 3)
OPEN                                ;UNTIL FILE=10
LDA      0      ZFILE,2  ;
STA      0      FILE     ; CLOSE(ZONE,TRUE);
LDA      1      .10     ;
SUB      1,0    SZR      ;
JMP                                ;
CLOSE                                ;

```

Example 20.

5.8.2 INPUT/OUTPUT Procedures.

5.8.2

5.8.2.1 Procedure TRANSFER.

5.8.2.1

	Call	Return
AC0	operation	destroyed
AC1	length	destroyed
AC2	zone	zone
AC3	-	destroyed

Initiates a transfer operation given by 'operation' and with bytecount 'length' to/from the databuffer pointed to by the used sharedescrptor.

After a call, the state of the sharedescrptor is set to the address of the messagebuffer, which was sent to the process given by ZNAME.

The used share is updated to the next sharedescrptor in the sharedescrptor chain.

The procedure does not check the state of the currently used share. If the state is nonzero, the messagebuffer address saved is lost and the message buffer is never released for use.

Message generated:

MESS0	operation
MESS1	length
MESS2	ZUSED.SFIRST
MESS3	ZBLOCK

All zone pointers and values are undefined.

## 5.8.2.2 Procedure WAITTRANSFER.

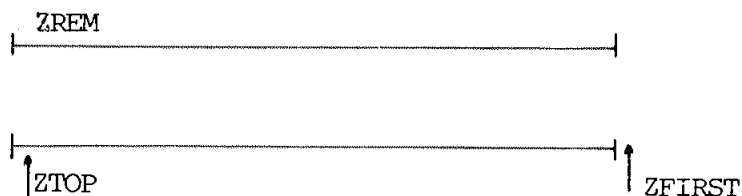
5.8.2.2

	Call	Return
AC0	-	destroyed
AC1	-	destroyed
AC2	zone	zone
AC3	-	destroyed

If state in the used shared descriptor is free the procedure returns immediately, otherwise it waits for an answer to the message placed in the message buffer given by state and when the answer is received, it sets the state to zero (free).

When the answer arrives, the status is checked as described in Exception Handling and the user giveup procedure can then be called.

After return, the pointers are updated as:



If a repetition of the message is generated, either by the user or by the system, the message sent is:

ZMODE(0:7)	OP
-	
ZFILE	
ZBLOCK	

OP = +1b10 (position) if ZKIND (13) = 1  
 +1b8 (erasure) if status (10) = 1 (parity) and zone is  
 opened for output.

Example:

The procedure INBLOCK can only be coded with the fundamental procedures TRANSFER and WAITTRANSFER.

All data buffers are sent to the input driver and then the first one sent is awaited.

```

ANEXT:  LDA      3      ZUSED,2  ;
        LDA      3      SSTATE,3 ; WHILE ZONE.USED.STATE
        ;              = 0 DO
        MOV      3,3    SZR      ;
        JMP      AWAIT  ;
        LDA      0      ZMODE,2  ; TRANSFER (ZONE,ZSHAREL,
        ;              ZMODE);
        LDA      1      ZSHAREL,2 ;
        TRANSFER ;
        JMP      ANEXT  ;
AWAIT:  WAITTRANSFER ; WAITTRANSFER (ZONE);
  
```

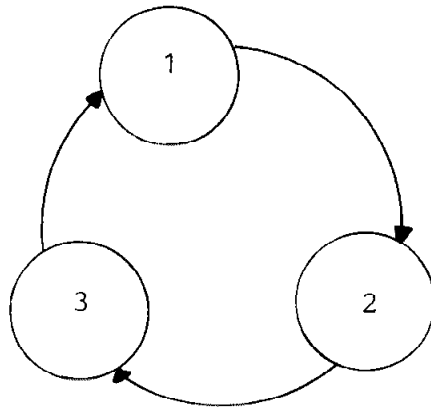
### 5.8.2.3 Procedure INBLOCK.

5.8.2.3

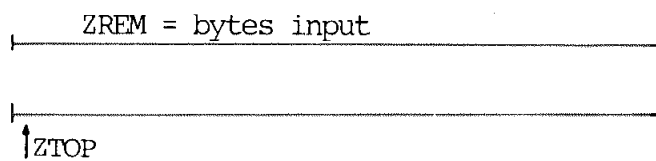
	Call	Return
AC0	-	destroyed
AC1	-	destroyed
AC2	zone	zone
AC3	-	destroyed

Administrates the basic cyclic buffering strategy for input procedures like INCHAR and GETREC.

The databuffers are represented by circles. INBLOCK starts the transfer of 1, 2 and 3. Then it waits for buffer 1, after which the data is ready for processing. The second call will request transfer to buffer 1 and wait for 2.



After return ZREM and ZTOP are defined as:



The messages generated are:

MESS0	ZMODE
MESS1	ZSHARE
MESS2	ZUSED.SFIRST
MESS3	ZBLOCK

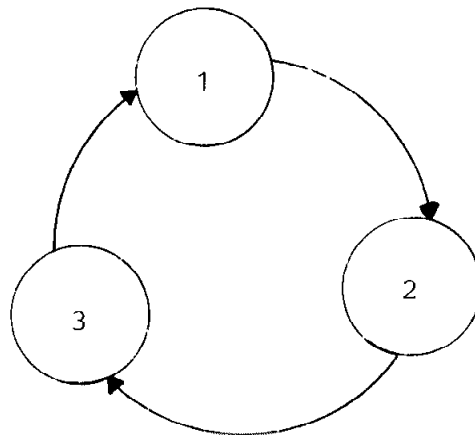
#### 5.8.2.4 Procedure OUTBLOCK.

5.8.2.4

	Call	Return
AC0	-	destroyed
AC1	-	destroyed
AC2	zone	zone
AC3	-	-

Administrates the basic cyclic buffering of output.

At first, buffer 1 is filled with output and after a call of OUTBLOCK, the first buffer is sent for processing. Buffer 2 is then ready to be filled. When buffer 1 is reached again, the answer is awaited and checked before the data buffer is ready to receive data again.



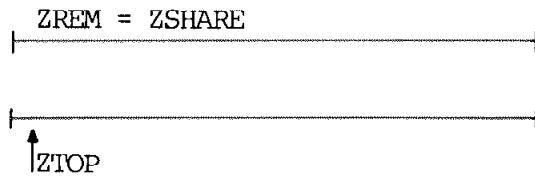


The message generated is:

ZMODE
ZSHAREL - ZREM *
ZUSED.SFIRST
ZBLOCK

\* ZSHAREL if ZKIND(11)=1

After return, ZREM and ZTOP are defined as:



6. RECORD I/O.

6.

6.1 Physical/logical Datablock.

6.1

When dealing with a physical data transfer to/from a certain device mounted document as to example a magnetic tape roll, the data are gathered in groups of data, called physical datablocks.

From a MUS-system point of view, interchange of data between documents is handled as a transport of physical datablocks between so-called zones (chapter 5). A zone contains information about the document in use and about data to be processed. To establish a link between the zone and physical datablock concept, the document to use and the process-buffer concept of the MUS-MONITOR system, the basic I/O-procedures are implemented (chapter 5).

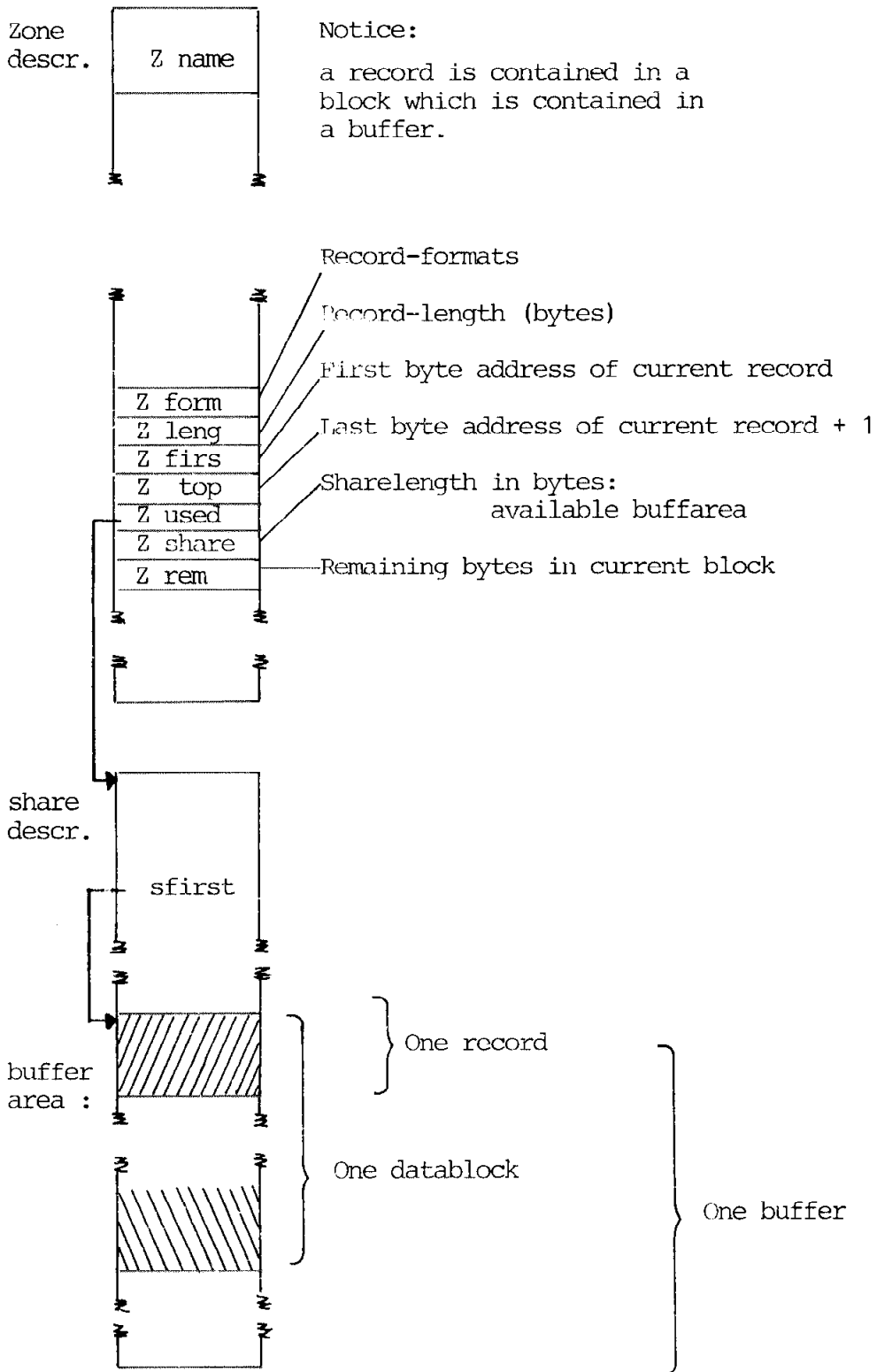
From a programmers point of view, a physical datablock should not always be looked upon as a logically indivisible whole of data. Often, a logical datablock surrounded by some header or tail information, and after some documents/device dependant physical features calls for unmatching moving information between different documents. To example:

Ex. I: Information hold on punched cards is to be transferred to disc files for future processing. This calls for reading datablocks of 80 bytes in length, transforming them into datablocks of 512 bytes in length. To easily access the information transferred, a logical block concept could be of great help.

Ex. II: Skipping information parts, according to some identifier-part of the physical datablock. If the third logical block is false, then skip the physical block, reading the next logical third block.

Thus to easily take care of handling logical information parts not necessarily fitting the physical blocklength, a logical data block concept a record is introduced. In the following, the physical datablock is referred to as a datablock or simply as a block.

LAYOUT of zonebuffer.  
Datablock and record.



Example 21

As mentioned earlier, the physical datablock (block) is used by the basic I/O procedures, f.ex. INBLOCK and OUTBLOCK, as information interchange element. Therefore, to introduce a logical datablock (record) information element, the record I/O procedures GETREC and PUTREC are developed as a second I/O procedure level, interfacing the basic I/O procedures to care for transfer of logical datablocks between a zone and its document.

GETREC reads in the next record to process here after defining this record as the current record, thus stepping through the records in the current block reading in a new block if necessary or so wanted.

PUTREC makes the next record to process available as the current record in reserving space for it in the current block. If no more space is left or the rest is not to be used, the current record and the records behind it, if any, belonging to the current block, are output and space is reserved in a new block. So, PUTREC does not actually care about how data is transferred to the current record. Current record is only made available for processing.

If a record is to be transferred from an inputzone to an outputzone, you must call GETREC on the inputzone to access the record. Then PUTREC must be called on the output to reserve space for the record and hereafter the record must be interchanged, calling f. ex. MOVE (chapter 6) on current records.

The current record is printed out according to ZFIRST, ZTOP and ZLENGTH in the zone (see ex. 21). The current record length could be initially set up or handled over as call value.

When talking about a block being processed, you should notice that:

Using GETREC sharelength of bytes is asked for in input using INBLOCK. Bytecount of answer is put to ZREM, which then defines current block length in use.

Using PUTREC only the record areas actually reserved are output to the document as a physical datablock, except for a document of disc kind. In this case, 512 bytes are transferred.

So when creating a zone, the block-length is not in all future equal to ZSHAREL. ZSHAREL actually defines the maximal block-length to be used.

Using the concept block-transfer, current block is found running the buffer wheel, if multibuffering is used (see chapter 5).

## 6.2 Record-Formats.

6.2

A variety of record formats exist as listed below, common to GETREC and PUTREC.

zone.ZFORM	Mnemonic	Description
0 :	U ;	Undefined recordlength, unblocked
1 :	UB ;	Undefined recordlength, blocked
2 :	F ;	Fixed recordlength, unblocked
3 :	FB ;	Fixed recordlength, blocked
4 :	V ;	Variable recordlength, unblocked
5 :	VB ;	Variable recordlength, blocked

Table 21.

Unblocked means: only one record is contained in one block.

Blocked means: one or more records are contained in one block.

Below an illustrating treatment of the various formats are given. For details about the block, record and unused buffer space relationship, refer to section 6.3 and 6.4,

6.2.1 Undefined,Unblocked.

6.2.1

NB R NB R NB R NB R NB

R = Record

NB = New Block is called.

One record equals one block. Each time GETREC/PUTREC is called, a block is transferred from or to the document. This format is suitable simple to use, when data processing is f.ex. stright forward datatransfer and the document is not known in advance.

Ex. I: Reading from a card reader, one card is transferred to the zone, if the length in bytes of one card  $\leq$  ZSHAREL.

Ex. II: Reading from a paper tape reader, share length of bytes are transferred.

Ex. III: Reading from magnetic tape, one block of data is transferred, if  $ZREM \leq ZSHARELENGTH$ .

GETREC returns the current recordlength found (no. of bytes transferred).

The number of bytes to be transferred using PUTREC, is given as call parameter and should not exceed the value of ZSHAREL. If this is the case, you will get an error indication, reading the status 1b8 + 1b15 in your GIVEUP procedure. If this status is accepted in this procedure or if no GIVEUP procedure is defined (a zero in the zone-entry), then return is made from PUTREC with ZSHAREL put into the call parameter, and thus space is made available for a record of length ZSHAREL.

Notice: This error information is only delivered when dealing with a block (1b8: block error) but is valid for any

format in use calling PUTREC. As it is to be seen on the next pages, a somewhat different error handling is made in case of GETREC-call.

### 6.2.2 Undefined, Blocked.

6.2.2

NB 

R	R	R
---	---	---

 NB 

R
---

 NB 

R	R	R
---	---	---

 NB

One block contains one or more records, not necessarily of the same record length, which is set up using GETREC and PUTREC in this case. Each time of call of GETREC/PUTREC recordlength of bytes are made available reading in/putting out the current block, if necessary.

This format could be used, when the data transfer is not simple sequential, f.ex. if data are augmented with head and tail records not to be processed or if data error indication marks, causing you to skip the next record, are given.

### 6.2.3 Fixed, Unblocked.

6.2.3

NB 

R
---

 NB 

R
---

 NB 

R
---

 NB

Each block contains one record. Each record is of equal length. The record length is initially set up in ZONE.ZLENGTH. This format is to be used, if the record-length is known in advance. It provides you with an error indication, whenever GETREC is called and a record of the right size is not found, standard error lb8+lb15. lb15 is set too, to indicate that lb8 is not driver set.

If the error is accepted, the record is delivered as a short record: the block is not filled up. Using PUTREC the current block is simply output and space reserved in the next block for a record of length ZONE.ZREM if possible.



Notice: error handling should be defined in a user defined GIVEUP procedure (chapter 5).

#### 6.2.4 Fixed, Blocked.

6.2.4



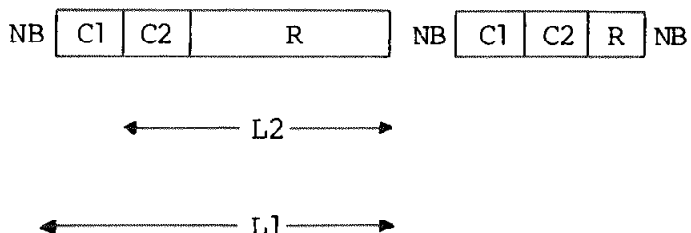
The block size should be an integral multiplum of the record length leaving no space behind. The record length is initially set up in ZLENGTH and all records should be of equal length. This format is very useful in handling wellknown documents, say to example a magnetic tape with a file containing card-image-records or a discfile with some database sort of linking of the records. As compared with the format F, a better utilization of the zone-concept is provided for here, especially in case of very small records, saving I/O overhead.

As for F-formats, a record-length check is performed on each record processed. Again a short record can be delivered - if the error is accepted - at return from GETREC.

Using PUTREC space is simply reserved for next record. If not enough space is left, no error is given, but space is reserved in a new block. If this is not possible, error is given.

#### 6.2.5 Variable, Unblocked.

6.2.5



IBM's record format V. One block contains one record. Block length and record length are not to be known in advance, but supplied to

the user at call of GETREC. Call of PUTREC results in creation of the fields C1 and C2 and reservation of space for a record. The record length (and hereby information about L1, L2) is delivered as call value.

Incorrect block size: 1b8+1b15 is set in status.

C1: 4 bytes in length.  
The first 2 bytes contain the value of L1, thus defining a block length. The value of the two last bytes is zero.

C2: 4 bytes in length.  
The first two bytes contain the value of L2, thus defining a pseudo recordlength, the value of the 2 last bytes is zero.

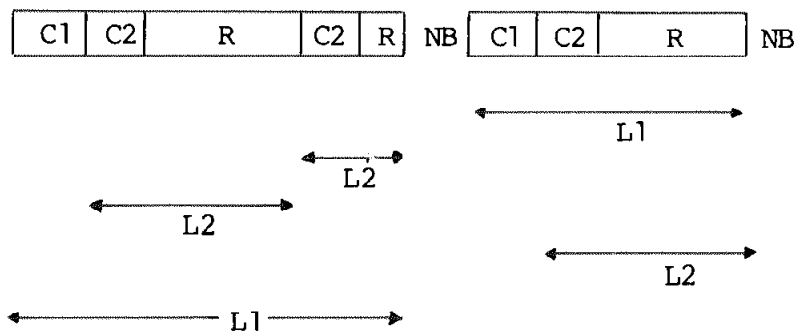
The recordlength equals (L2-4) bytes, which is the value delivered from GETREC and the value to deliver, using PUTREC.

NOTICE: Errorhandling is performed as described in the section dealing with F formats with the very one exception:

If using GETREC, having in progress a short record, then this situation is not reported as an error situation, if discfile is in use. That is: a short block is delivered without status indication. Notice that the transferred number of bytes always can be found in the return parameter of GETREC.

## 6.2.6 Variable, Blocked.

6.2.6



IBM's recordformat VB. One or more records contained in one block. The records must fill up one block (no free space left). In case of incorrect block length, a short block is delivered, if the error is accepted.

Call of PUTREC results in creation of the field C2, if current block must be changed, the fields C1 and C2 are created.

Notice: C1 is updated each time PUTREC is called, to provide for the increasing record number. The record length is delivered as call value using PUTREC. C1 and C2 are defined as illustrated in section 6.2.5

Blockerror: 1b8+1b15 is set, refer to the section dealing with FB formats. Again in case of dischandling using GETREC an exception as the one mentioned above is made.

## 6.3 GETREC.

6.3

## 6.3.1 Procedure GETREC (Zone, Addr., Bytes).

6.3.1

	Call	Return
AC0	bytes	bytes
AC1		addr
AC2	zone	zone
AC3	link	destroyed

Next record is made available in the current input buffer. IN-BLOCK is called, if current block covers the demand for a record insufficiently according to the various formats.

Parameters in use:

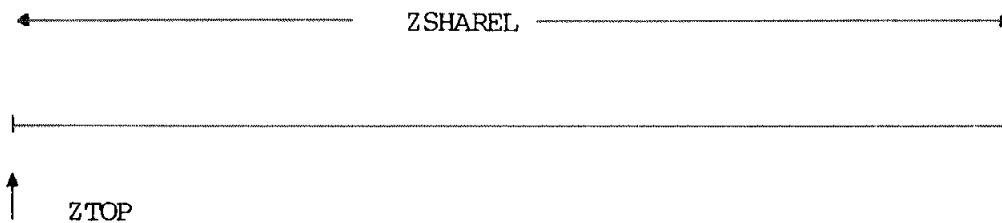
bytes: call value is used in case of UB format only to set up zlength. At return, the value of bytes is set to zlength or blockerror - return value:

Non error return:	U:	bytes:= zlength:= zrem
	UB:	bytes:= zlength:= bytes
	F, FB:	bytes:= zlength (constant, initially set up)
	V, VB:	current record length in bytes is set up in zlength using C2. Bytes: = zlength.

Error return: bytes:= zrem;

addr:	ZFIRST	(byteaddress)
zone:	ZONEADDRESS	(wordaddress)

After call of OPEN (zone, operation) the bufferarea of used share is pointed out as shown. Operation = input:



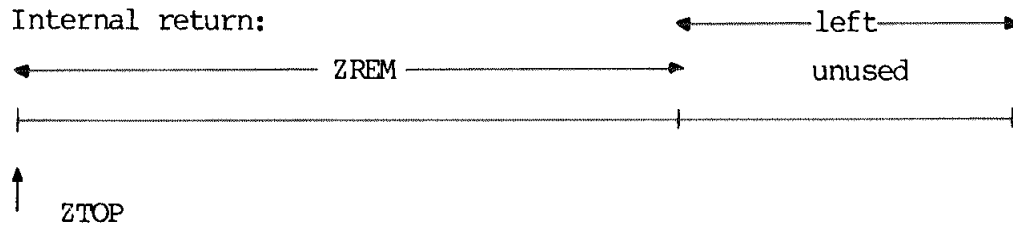
ZREM: = 0, ZFIRST undefined. No data transfer.

Fig. 22.

First call of GETREC (zone, bytes):

INBLOCK is called.

At INBLOCK-return ZREM:= bytecount of answer.



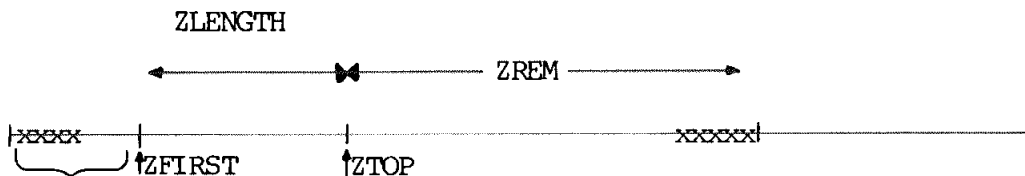
ZFIRST is undefined. Datatransfer is done

Fig. 23.

Hereafter the current record is pointed out according to the various formats in use:

Return from GETREC:

External GETREC return.



8 bytes are left behind in case of V and VB format.

Fig. 24.

Succeeding calls of GETREC (zone, bytes):

If ZREM covers the demand for a record insufficiently INBLOCK is called and the action illustrated above in fig. 23 and fig. 24 is repeated for the next buffer to use. If this is not the case, refer to fig. 25.

(UB, FB, VB)

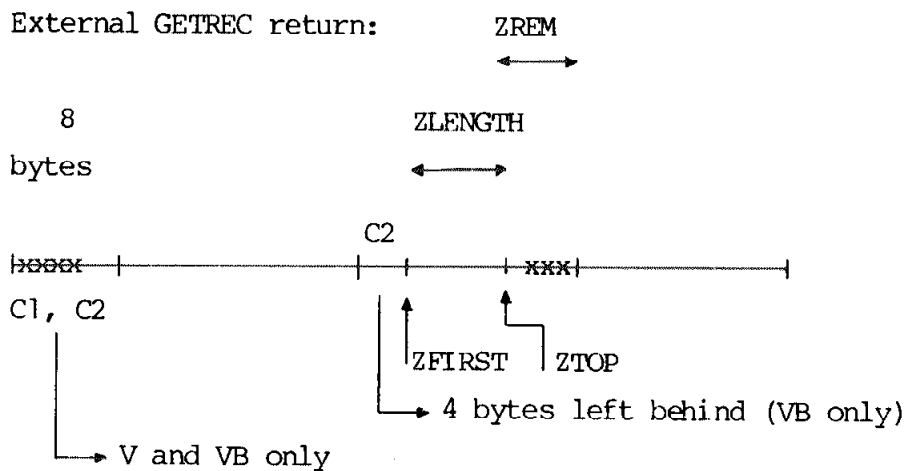


Fig. 25.

6.3.2 Pseudo-Algol Description.

6.3.2

In the following, a detailed pseudo-algol-description of GETREC is given according to the various formats:

Local procedures: BLOCKERROR and UPDATE and GETHEAD.

U: A buffer is made available for processing containing a datablock. In case of let us say paper tape in usage as input medium (zone.zname = PTR) , the datablock fills up the buffer where as usage of card as input medium results in one card being read defining one datablock not necessarily filling up the buffer.

```

repeat INBLOCK until Zrem > 0;
bytes := Zlength := Zrem;
UPDATE;
0: ;GETREC return

```

UB: Bytes characters are made available for processing. If INBLOCK is called, refer to comments given above.

```
Zlength:= bytes;
If bytes <> Zrem, then repeat INBLOCK until Zrem ≥ Zlength;
bytes:= Zlength;
UPDATE;

0: ;GETREC return.
```

F: Notice: Zlength should be initially defined.  
Zlength of bytes are made available for processing.  
Comments as for U except that data read left unaccessed results in blockerror, which is of interest using discfiles as input medium in that INBLOCK results in 512 bytes being read and therefore requires Zlength:= 512.

```
1: If Zrem <> 0, then BLOCKERROR;
   repeat INBLOCK until Zrem > 0;
   bytes:= Zlength;
   if bytes > Zrem
   then goto 1
   UPDATE;
0: ;GETREC return
```

FB: If INBLOCK is called, refer to comments given above. If this is not the case, notice that zlength of bytes is being accessed:

```
   bytes:= Zlength;
   if bytes > Zrem
   then goto 1;
   UPDATE;
0: ; GETREC return
```

V: This format equals the IBM V-format. Incorrect format length is handled as for F-formats. Please notice that in case of discfile as input medium, rounding off input block is provided for. Decoding of block- and recordlength is made acc. to the description given earlier (GETREC/PUTREC - format) and accomplished for in the local procedure GETHEAD:

```

2: repeat INBLOCK until zrem >0;
   GETHEAD; !a local variable l.bytes is declared.!
   If l.bytes <> zrem then
     Begin
       If zone.zkind (11) <> 1 then BLOCKERROR;
       Zrem:= l.bytes !a short block is delivered!
     end;

5: If Zrem = 0 then goto 2;
   GETHEAD;
   bytes:= Zlength;
   UPDATE;

0: GETREC return

```

VB: This format equals the IBM VB-format. If INBLOCK is called, refer to comments given above:

```

if Zrem = 0 then goto 2;
GETHEAD;
bytes:= Zlength;
UPDATE;
0: ;GETREC return.

```



```
PROCEDURE GETHEAD;
```

```
  if zrem < 4 then BLOCKERROR;
```

```
  zrem:= zrem -4;
```

```
  ztop:= ztop +4;
```

```
  zfirst:= ztop -4;
```

```
  l.bytes:=C;! C equals C1 OR C2!
```

```
  l.bytes:=C - 4;
```

```
  zlength:= l.bytes;
```

```
  if l.bytes >zrem then BLOCKERROR:
```

```
END GETHEAD;
```

```
PROCEDURE UPDATE (bytes);
```

```
BEGIN: zrem:= zrem - bytes;
```

```
        ztop:= ztop + bytes;
```

```
        zfirst:= ztop - bytes;
```

```
END;
```

```
END UPDATE;
```

```
PROCEDURE BLOCKERROR
```

```
BEGIN
```

```
  z0:= lb8 + lb15;
```

```
  IF Zgiveup <> 0 then go to GIVEUP;
```

```
    bytes:= zrem;
```

```
    UPDATE;
```

```
    GOTO 0    ;
```

```
END;
```

```
END BLOCKERROR;
```

Notice that if a GIVEUP procedure is called, ZREM must differ in value from zero at GIVEUP bottom-out return. If ZREM equals zero, the GIVEUP procedure is called again and a loop is established in your program.

## 6.3.3 Programming Example.

6.3.3

```

; Use of GETREC, no. of bytes on tape:600.
LDA      0      .1      ;AC0:= 1
LDA      2      UZONE   ;
OPEN                                           ;UZONE OPENED FOR INPUT,
                                           ;ZMODE is defined
GETREC                                       ;512 bytes are asked for
                                           ;
STA      0      GNORW   ;GNORW:= no delivered
                                           ;The first byte of cur-
                                           ;rent record is addres-
                                           ;sed via AC1.
GETREC                                       ;Notice:
                                           ;88 bytes are delivered
                                           ;OK but status EM is
                                           ;found. Return is made to
                                           ;GIVEUP. The CPU is
                                           ;stopped.
GNORW:   0                                           ;No of bytes read
UZONE:   .+1                                         ;Zone in use.
.TXT.PTR<0><0>.                                     ;Document: 6 bytes name
0                                               ;ZMODE: undefined
1                                               ;ZKIND: char oriented
65535                                         ;ZMASK: errormask for
                                           ;giveup
UGIVE                                         ;ZGIVEUP address
.LOC     UZONE + ZCONV                           ;
0                                               ;No conversiontable
.LOC     UZONE + ZFORM                             ;
0                                               ;Undefined, unblocked
.LOC     UZONE + ZSHAR                             ;Bufferarea in bytes
512                                           ;
                                           ;For a fully set up of
                                           ;zonedescr., sharedescr.
UGIVE:   .                                           ;+ buffers refer to ch.5
.
HALT                                           ;Stop CPU.

```

Example 22.

6.4 PUTREC.

6.4

6.4.1 Procedure PUTREC (zone, addr., bytes).

6.4.1

	Call	return
AC0	bytes	bytes
AC1	-	addr
AC2	zone	zone
AC3	link	destroyed

In current output buffer, space for next record is reserved. If no space is left or format is unblocked, then current block is output using OUTBLOCK, and hereafter space is reserved in next output buffer of the bufferwheel.

Parameters in use.

bytes: call value is used in case of U, UB, V, VB to set up zlength. In case of F and FB-format, zlength should be initially defined.

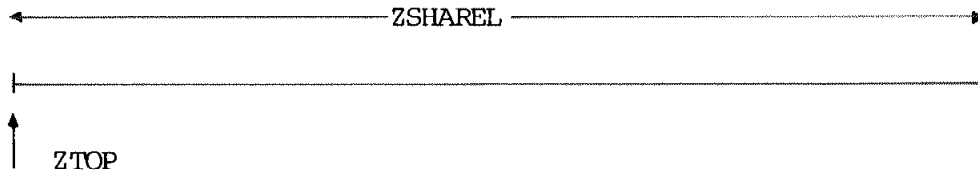
Return value equals zlength in all cases of non block error return else zrem.

addr: zfirst (byteaddress).

zone: Zoneaddress (wordaddress).

After call of OPEN (zone, operation), the buffer area of used share is pointed out as shown.

Operation = output.



ZREM:= ZSHAREL.

ZFIRST is undefined. No datatransfer.

Fig. 26

First call of PUTREC (zone, bytes).

External PUTREC return.

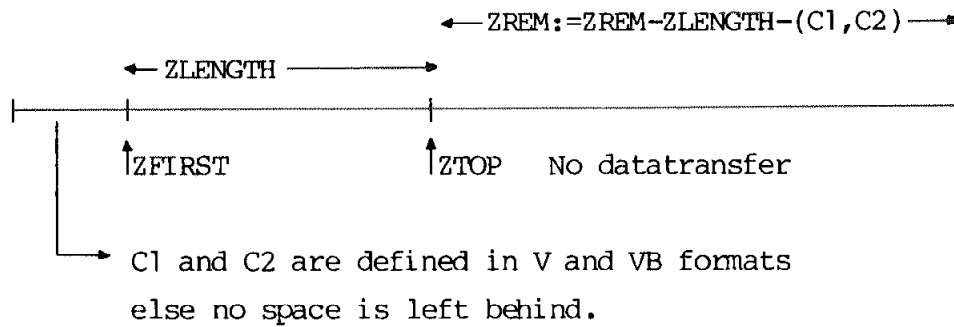


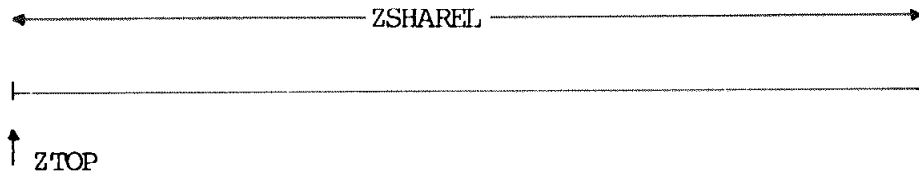
Fig. 27.

Next call of PUTREC (zone, bytes).

In case of unblocked formats, OUTBLOCK is called with bytecount equal to ZSHAREL-ZREM. In case of discfiles in use, ZREM is put to zero unconditionally.

At OUTBLOCK-return ZREM:= ZSHAREL:

Internal return.



ZREM:= ZSHAREL.

ZFIRST is undefined. Datatransfer is done.

Fig. 28.

Return from PUTREC:

External return:

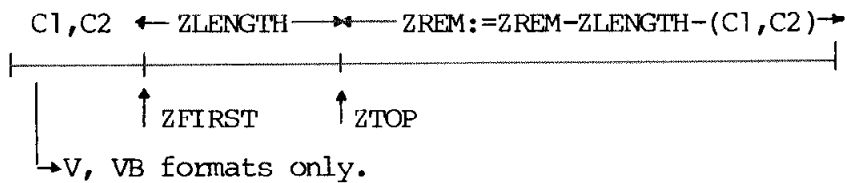
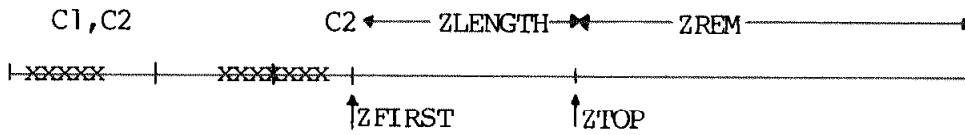


Fig. 29

In case of blocked formats, next call of PUTREC is done according to fig. 28 and fig. 29, if OUTBLOCK is called. If this is not the case (zrem covers the demand for a record), refer to fig. 30.

Return from PUTREC, no datatransfer.

External return:



ZREM:= ZREM-ZLENGTH-C2;

Fig. 30.

#### 6.4.2 Pseudo-Algol Description.

6.4.2

In the following, a detailed pseudo-algol-description of PUTREC is given acc. to the various formats.

local procedures: UPDATE and BLOCKERROR

Call of PUTREC (zone, bytes):

U: Current block of data is put out. Then space is reserved in the next buffer for the record to be output next time PUTREC is called.

```

    zlength:= bytes;
    OUTBLOCK;
    if zrem <zlength then BLOCKERROR;
    UPDATE (0); !init. of zfirst.!
    UPDATE (zlength)
0: ;PUTREC return

```

UB: If no room for next record of length 'bytes' OUTBLOCK is called.

```

    zlength:= bytes;
    if zrem < bytes then OUTBLOCK;
    if zrem < zlength then BLOCKERROR;
    if ztop = sfirst then UPDATE (0);
    !after call of outblock.!
    UPDATE (zlength);
0:   ;PUTREC return.

```

F: Comments as for U-formats. Notice that zlength should be initially defined.

```

    bytes:= zlength;
    zlength:= bytes;
    OUTBLOCK;
    if zrem < zlength then BLOCKERROR;
    UPDATE (0);
0:   ;PUTREC return

```

FB: Comments as for UB-format. Notice that zlength should be initially defined.

```

    bytes:= zlength;
    zlength:= bytes;
    if zrem < bytes then OUTBLOCK;
    if zrem < zlength then BLOCKERROR;
    if ztop = sfirst then UPDATE (0);
    UPDATE (zlength);
0:   ;PUTREC return

```



V: IBM-V-format. C1 and C2 are defined. Comments as for U-format.

```

60  bytes:= bytes +4;
    zlength:= bytes;
    OUTBLOCK;
    is zrem < zlength then BLOCKERROR;
    UPDATE (4); !reservation of space for C1!
1:  UPDATE (zlength); !reservation of space for C2 and the
        record.!
    insert (C2, zlength); !local production: C2:= zlength!
    zlength:= zlength - 4; !record alone defined.!
    zfirst:= zfirst + 4;
    l.zlength:= ztop - sfirst;
    evaluate (C1, l.zlength);!C1 is updated acc to current
        length of current block.
        Note: sfirst is used.!

0:  ;PUTREC return

```

VB: IBM-VB-format. C1 and C2 are defined. Comments as for UB format.

```

    bytes:= bytes + 4;
    zlength:= bytes;
    if zrem < bytes then OUTBLOCK;
    if zrem < zlength then BLOCKERROR;
    if ztop = sfirst then UPDATE (4);
    goto 1;

```

```
PROCEDURE UPDATE (bytes);  
BEGIN  
  ZREM:= ZREM - bytes;  
  ZTOP:= ZTOP + bytes;  
  ZFIRST:= ZTOP - bytes;  
END;
```

```
PROCEDURE BLOCKERROR;  
BEGIN  
  Z0:= lb8 + lb15;  
  If Zgiveup <>0 then goto GIVEUP;  
  bytes:= ZREM;  
  UPDATE (bytes);  
  GOTO 0;  
END;
```

Notice that if a GIVEUP procedure is called, ZREM must differ from zero at GIVEUP bottom-out return. If ZREM equals zero, the GIVEUP procedure is called again and a loop is established in your program.



```

OZONE:    .+1                ;zone in use
          .TXT .MTO<0><0>.    ;ZNAME
          0                  ;ZMODE
          13                 ;ZKIND
          65535              ;ZMASK
          UGIVE              ;ZGIVEUP

          .LOC    OZONE + ZCONV
          0
          .LOC    OZONE + ZFORM    ;
          2                ;Fixed blocked
          256              ;ZLENGTH defined
          .LOC    OZONE + ZSHAR    ;

          400                ;

          UGIVE:            ;
          JMP      +0,3        ;Return to call point
                              ;in PUTREC

```

Example 23.

6.5      MOVE.

6.5

Procedure MOVE (param.addr)

	Call	Return
AC0	-	destroyed
AC1	-	destroyed
AC2	param.addr	param.addr
AC3	link	destroyed

Parameters in use.

Param.addr: count, no of bytes to be transferred.

Param.addr+1: to address where to put the first byte

Param.addr+2: from address where to take the first byte.

Param.addr+3: work location.

Param.addr: wordaddress.

Example of use:

```

      ZFIRST OF INPUTZONE   zone.ZFIRST
      |-----|
from (IN↑+M): |-----|
      |-----|
to (OUT↑):    |-----|
      |-----|
      ZFIRST OF OUTPUTZONE  zone.ZFIRST

```

The 'to'- and 'from' - strings need not to be disjoint. In case of IN=OUT the procedure may fail to work, if 'to-address' is greater than 'from-address' (data in 'from' may be destroyed before they should be used).

Notice: The parameter area pointed out by the parameter address is destroyed.

6.5.1 Programming Example.

6.5.1

Eks:

```

      .
      .
      LDA      2      PADDR      ;
      MOVE                      ;
      .
      .
      .
      .
PADDR:  PSTAC                      ;
                                           ;
                                           ;
                                           ;Parameter field.
PSTAC:  MCOUNT                    ;bytecount
      MOTO                      ;To address
      MOFRO                      ;From address
      0                          ;work location

```

Example 24.

7. CHARACTER I/O PROCEDURES.

7.

The single character - and the string oriented transfer procedures are contained in the module MUC.

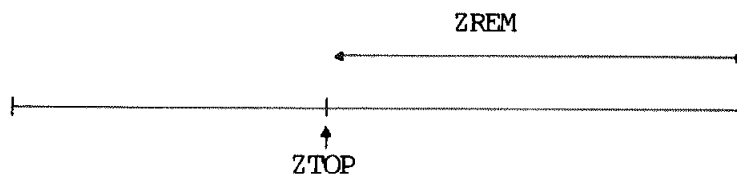
These procedures can be looked upon as a special case of the record I/O procedures. Thus the concepts around the fundamental datastructure, the zone, are equally valid.

The character I/O procedures are augmented with the two utility procedures DECBIN and BINDEC.

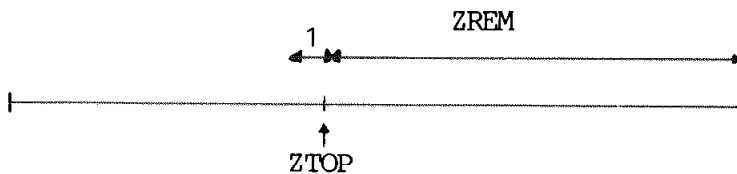
7.1 Single Character Procedures.

7.1

Before call:



After call

7.1.1 INCHAR

7.1.1

	Call	Return
AC0	- - -	destroyed
AC1	- - -	character
AC2	zone	zone
AC3	link	destroyed

Gets the next byte from the zone. INBLOCK (see chapter 5) is called if ZREM upon entry equals 0.

### 7.1.2 OUTCHAR.

7.1.2

	Call	Return
AC0	- - -	unchanged
AC1	character	destroyed
AC2	zone	zone
AC3	link	destroyed

Outputs the character contained in AC1 onto the zone. OUTBLOCK (see chapter 5) is called, if ZREM upon entry equals 0.

### 7.1.3 OUTSPACE.

7.1.3

	Call	Return
AC0	- - -	unchanged
AC1	- - -	destroyed
AC2	zone	zone
AC3	link	destroyed

Calls OUTCHAR with a preloaded character value equal to 32 (ASCII space).



7.1.4     OUTEND.

7.1.4

	Call	Return
AC0	- - -	destroyed
AC1	character	destroyed
AC2	zone	zone
AC3	link	destroyed

Outputs the character onto the zone by means of `OUTCHAR`. In order to avoid any blocking effects on character oriented devices, `OUTBLOCK` will be called if the device is character oriented. Thus the share will be emptied by outputting that part presently being filled with characters.

E.g. the terminating line, when outputting a disc file on the TTY could else be displayed in two steps, if the line was spanning across two disc sectors.

7.1.5     OUTNL.

7.1.5

	Call	Return
AC0	- - -	destroyed
AC1	- - -	destroyed
AC2	zone	zone
AC3	link	destroyed

Calls `OUTEND` with a preloaded character value equal to 10 (ASCII New Line).

7.2 String Oriented Procedures.

7.2

7.2.1 OUTTEXT.

7.2.1

	Call	Return
AC0	address (byteaddr)	destroyed
AC1	- - -	destroyed
AC2	zone	zone
AC3	link	destroyed

Outputs the text by means of OUTCHAR, until a character equal to 0 is encountered. The terminating 0 will not be output.

7.2.2 OUTOCTAL.

7.2.2

	Call	Return
AC0	value	destroyed
AC1	- - -	destroyed
AC2	zone	zone
AC3	link	destroyed

Converts the value to a 6-digit octal character and outputs this string onto the zone by means of OUTCHAR.

E.g. 115 will be output as 000163

7.2.3 INNAME.

7.2.3

	Call	Return
AC0	- - -	destroyed
AC1	address (word addr)	destroyed
AC2	zone	zone
AC3	link	destroyed

Calls INCHAR repetively until the character string being input follows the definition of name (from 1 to 5 characters long and having unused positions equal to 0).

E.g. '<32><32>NAME<13>' will be stored as 'NAME<0><0>'

7.3 Utility Procedures.

7.3

The user must realize that the utility procedures are using the location SAVE, SAVE1 and SAVE2 or CUR+24, CUR+25 and CUR+26, where the displacements are octal.

7.3.1 DECBIN.

7.3.1

	Call	Return
AC0	- - -	destroyed
AC1	address (byteaddr)	value
AC2	cur	cur
AC3	link	destroyed

The address must point at a sequence of ASCII characters and the procedure will deliver its binary equivalent in 'value'. The procedure will terminate, when the first non-digit character (a character outside the range of 48 to 57) is encountered. No check for overflow is made. If address upon entry is pointing at a

non-digit character (e.g. a sign character) the value delivered will be 0.

E.g. '+123' will be returned with value 0

'12<32>34' will be returned with value 12.

### 7.3.2 BINDEC.

7.3.2

	Call	Return
AC0	value	destroyed
AC1	address (byteaddr)	destroyed
AC2	cur	cur
AC3	link	destroyed

The binary value 'value' is converted into a 5 decimal and a <0> byte string.

E.g. value =123 will be converted into '00123<0>'.

7.4 Programming Examples.

7.4

```

      .
      .
OTEXT: .+1
.TXT   .OUTPUT STRING.
      .
      .
      .
      LDA      2      IZONE      ;
      INCHAR                                ; GET CHARACTER
      .
      .
      .
      LDA      1      CHAR       ;
      LDA      2      IZONE      ; OUTPUT CHARACTER
      OUTCHAR                                ;
      .
      LDA      0      OTEXT      ;
      MOVZL                                0,0      ; ADDRESS:=BYTEADDRESS
      LDA      2      IZONE      ; OUTPUT TEXTSTRING
      OUTTEXT                                ;
      .
      .
      .
WORK:  .+1
      0                                ; NAME
      0                                ; NAME
      0                                ; NAME
      .
      .
      .
      LDA      1      WORK       ;
      LDA      2      IZONE      ; GET NAME
      INNAME                                ;

```

## 8. CATALOG SYSTEM.

8.

### 8.1 Introduction.

8.1

RC3600 file system makes it possible to divide a disc drive into smaller independent units, files. These files are identified by names, with a length of 5 characters. The descriptions of the files, entries, are kept in a catalog stored on the disc. The entries give among other things the name, the length, and the starting position on the disc of the file.

RC3600 file system allows up to 255 discs to be connected to the same RC3600 central unit. Each disc is handled independent of all others as it holds one or more catalogs of its own. Each catalog and the disc space covered by the catalog files is called a catalog unit. Up to 255 catalog units can be handled by the file system, and more than one catalog unit can be placed on the same physical disc pack or cartridge. Files on different catalog units may have the same name, and it is therefore necessary to give the number of the catalog unit, when accessing a file.

With the RC3600 file system you can create, remove and change files, and read or write into existing files.

Operations involving update of the catalogs are handled in a special catalog handler process, CAT, and this process administrates the creation and removal of area processes. The area process will when created act as a normal driver process, and file read/write operations are directed to this area process, which will modify and read data only on the file with the same name as the area process. Moreover the area process administrates the current position in the file for up to three different user processes, and takes care of the different kinds of user reservations.

A file may be a new catalog (sub catalog) for a number of files. To access files in a sub catalog, a catalog process must first be created for that sub catalog. Creation and removal of catalog processes are also done by the process 'CAT'. All operations on the sub catalog and the files within it are carried out by sending messages to the catalog process for the sub catalog in question.

Catalog operations are in the basic level of procedures requested by sending messages to the process CAT. A detailed description of the message interface can be found in [5].

In the MUS Basic I/O system a number of procedures have been build to ease the use of the catalog system. These procedures are working on the file descriptor (zone) structure as described in chapter 5, and enable the user to work on well known structures and with simple exception handling by means of the defined giveup procedure.

## 8.2 Catalog System Disc Structure.

8.2

The smallest unit accessible on an RC3600 disc is one sector, each with 512 bytes data. Each file created by the catalog system consists of a number of sectors random distributed over the disc surface, and the relative position on the disc of each sector is collected on one sector, called the indexblock. In order to get fast access to each sector in the file the sectors are collected in physical contiguous blocks called slices, each consisting of a fixed number of sectors.

The indexblock contains then a number of slice discriptions each describing a collection of contiguous slices. The first word of the indexblock gives the number of slice descriptions in use. The maximal number of slice descriptions for a file is 127.

A slice description consists of two words. The first word gives the number of sectors and the next word gives the start address relative to the catalog start.

If a file is extended the catalog system will use the physically next slice on the disc, if possible. Thus it suffices to increase the number of sectors occupied by the last slice description in the indexblock. I.e. if a new disc is created all files will be physical contiguous and described by only one slice description in the indexblock.

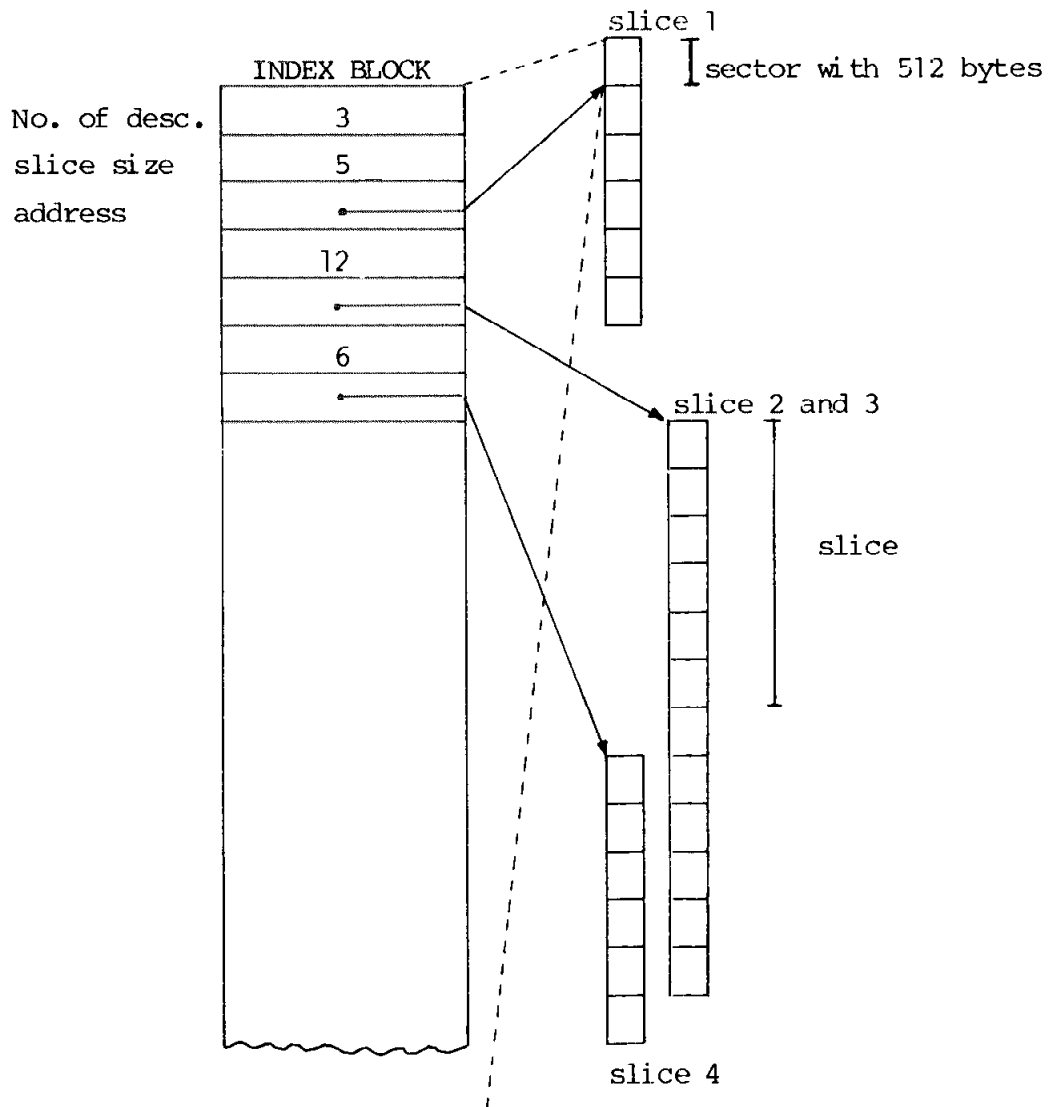


Fig.31 : Index block structure. The index block is the first segment in the first slice.



The reference to the file indexblock is placed in a file entry which also contains the filelength, name and specifications bits (attributes).

All file entries on a catalog unit are collected in a standard data file SYS. This file can be read by the user, and the file only differs in the fact that its indexblock is fixed on sector 6 and it is protected against user writing and removal.

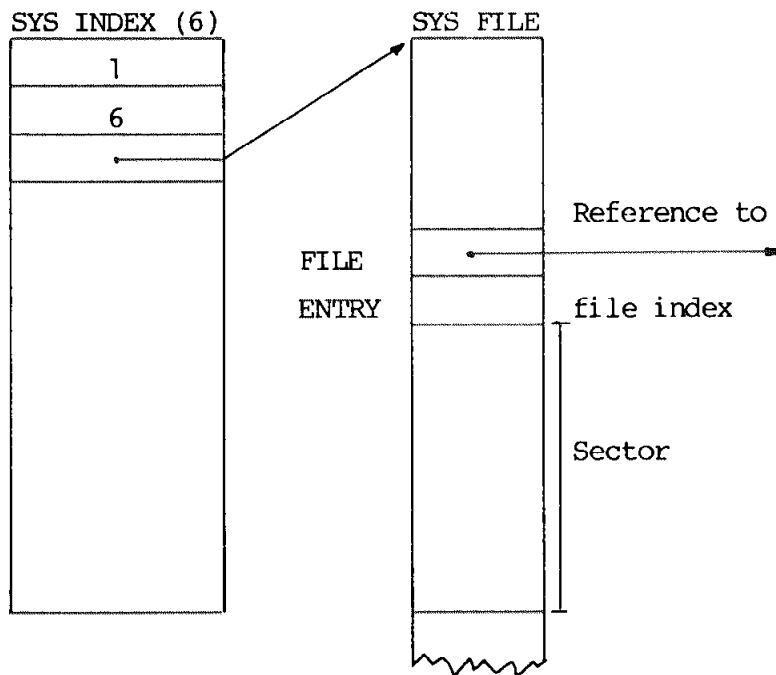


Fig.32 The catalog file SYS. Please note that the entry SYS is represented in the file itself.

Each entry in the SYS file is represented by 16 words with the layout given in fig.33.

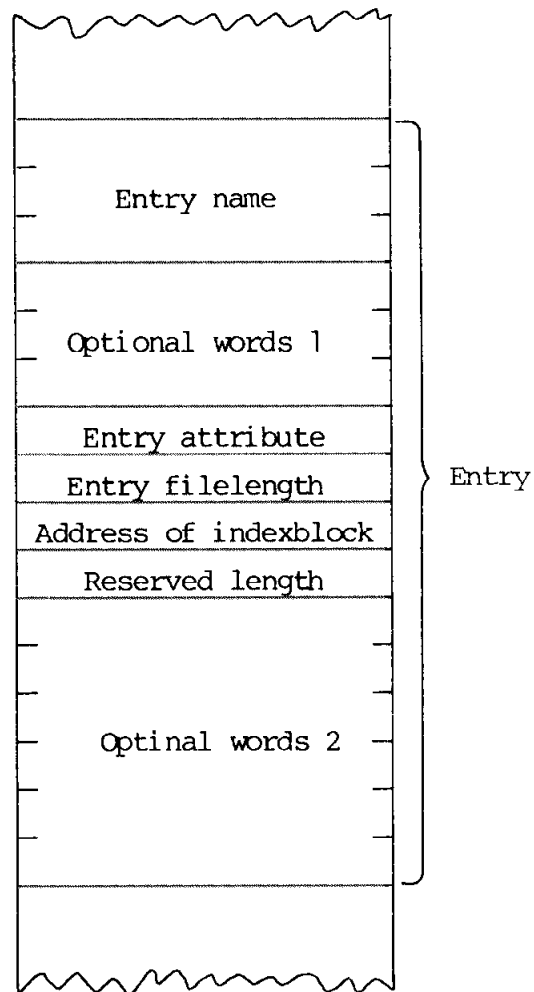


Fig.33 Entry layout in file SYS.

Entries not occupied in the catalog file are given by a zero name, and all entries are placed in the file SYS by means of a hashkey value computed from the entryname.

Entries with the same hashkey are thus found in the sectors  $k$ ,  $k+n$ ,  $k+2n$ , ... in the catalogfile, where  $k$  is the hashkey and  $n$  is the SYS-size, which is an integral multiple of the slice size. A new entry is inserted in sector  $k$  (the hashkey), but if this sector is occupied sector  $k+n$  is used, an so on. The catalog is extended and the added sectors are zeroized if no space is found in existing sectors.

Sysslicesize = n.

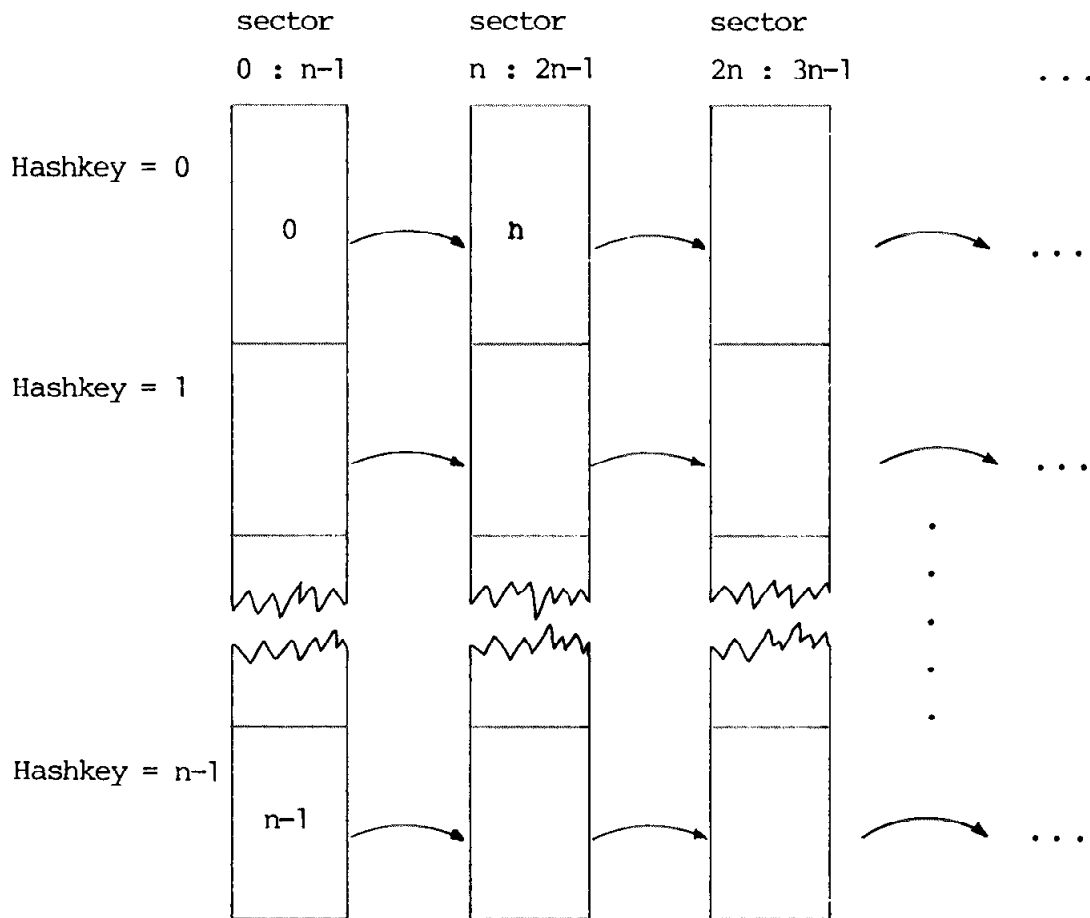


Fig.34 Hashkey organisation in file SYS.

As seen from the entry layout two lengths are connected to each file, the filelength and the reserved length. The reserved length is always an integral number of the number of segments in a slice minus the one occupied by the indexblock, and the relation

$$\text{file length} \leq \text{reserved length}$$

is therefore valid, as the file length is the number of user accessible segments (data segments) on the file, and the reserved length is the number of segments occupied on the disc by the current file.

Each slice on the disc is mapped on a bit in the map sector, which means that if a slice is included in a file and placed in the fileindexblock, the equivalent mapbit is zeroized, else it is set to one.

Further information about the catalog unit structure is kept in a fixed sector on the disc. These informations are catalog size, slicelength and data sector start.

The map sector and disc description sector are placed in a user accessible datafile too, called MAP.

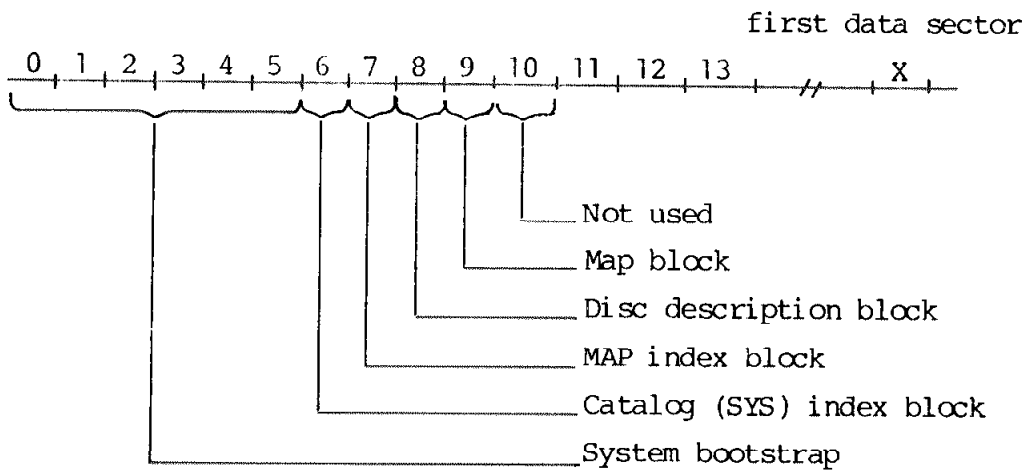


Fig.35 Disc and catalog organisation

When the MAP file is read the first sector contains the following information :

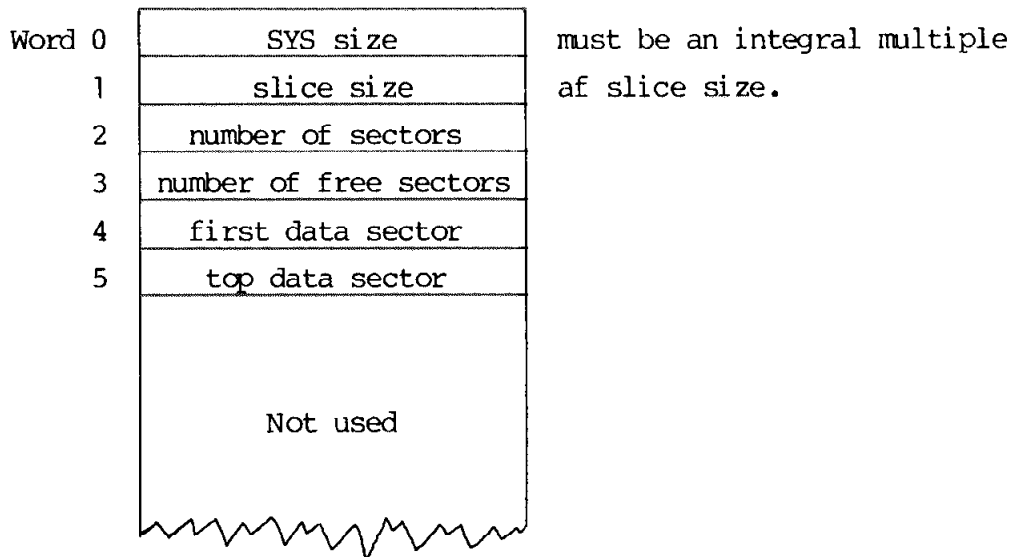


Fig.36 Disc description sector.

The next sector is the bitmap. A free slice is indicated as a one in the appropriate bitposition.

The physical position on the actual disc configuration of each catalog unit can be found in a separate description program, which gives the connection between the catalog unit and the name of the used disc driver process and the start address of the first sector on the disc. The description program is created as an assembler program containing a predefined data structure. The catalogprocess CAT is then the same in all system configurations and the actual configuration is given by the tables in the description program with the fixed name CATW. A part of the disc description sector is read and moved to the description program when the actual catalog unit is initialized.

A detailed description of the formats in the CATW program can be found in [5].

### 8.3 Catalog System Procedures.

8.3

#### 8.3.1 General.

8.3.1

When catalog system procedures are called a fixed parameter is always the file descriptor address. By this reference to the file descriptor the name of the file to be manipulated and the giveupprocedure to be called in case of errors, are defined. In addition the conversion table address in the file descriptor (zone.zconv) is used to specify if the catalog procedure should be performed on a main catalog or on a sub catalog. When accessing files in a sub catalog the conversion table address should point to a string containing the sub catalog name.

#### 8.3.1.1 Main Catalogs.

8.3.1.1

If zone.zconv = 0 the operation is send to the process CAT, which handles operations on main catalogs. The document name in the file descriptor must hold the name of the file to work on (zero characters for unused positions), and the main catalog unit number in the last byte of the name.

Operations which can be performed on main catalogs are:

- 1) Createentry
- 2) Removeentry
- 3) Lookupentry
- 4) Changeentry
- 5) Setentry
- 6) Initcat
- 7) Newcat
- 8) Freecat

8.3.1.2 Sub Catalogs.

8.3.1.2

If zone.zconv  $\diamond$  0 the operation is send to a process with name pointed out by the byteaddress in zone.zconv. This name should be the name of a catalog process consisting of the 5 character sub catalog name followed by a sub catalog number as the 6. character. The sub catalog number is gained from the catalog system when the catalog process is created by using the catalog procedure NEWCAT.

The document name in the file descriptor must hold the name of the file to work on (zero characters for unused positions) in the first 5 characters. The 6. character is automatically supplied with the sub catalog number specified as the 6.th character of the catalog process name.

Operations which can be performed on sub catalogs are the same as for main catalogs except for 6, 7 and 8. Before the operations can be used the catalog process must be created by using the procedure NEWCAT. Catalog processes are removed by the procedure FREECAT.

8.3.1.3 Procedure CREATEENTRY.

8.3.1.3

	Call:	Return:
AC0	attributes	destroyed
AC1	size	destroyed
AC2	zone	zone
AC3	-	destroyed

The file given by the document name in zone is created with the given size and the given attribute mask.

The attribute mask has the following interpretation:

- bit 15 = 1      The file should be extendable, else the file is created with fixed length, i.e. it is not allowed to change the file length.
  
- bit 14 = 1      Not used in createentry. Device descriptor in DQMUS.
  
- bit 13 = 1      Not used in createentry. Entry only.
  
- bit 12 = 1      The file is writeprotected, i.e. it is allowed only to read from the file.
  
- bit 11 = 1      The file is permanent, i.e. removal and change (except attribute change) of the file is not allowed.
  
- bit 10 = 1      Not used in createentry. Link entry.
  
- bit 1 = 1      Sub catalog. The file is a sub catalog.
  
- bit 0 = 1      Catalog file. Can be read by the user, but is protected against write, change and removal. Only system use is allowed.

If other bits are set in attribute mask, the giveup procedure is called.

For status, see table 1.



Create the file PIP on catalog unit 0:

```

FDO:                                ; ZONE:
      .TXT .PIP<0><0>.                ;
      .                                ;
      .                                ; STANDARD LAYOUT
      .                                ;
      .                                ;
      .                                ;
      .                                ;
      .                                ;
      .                                ;
      .                                ;
      .                                ;
      .                                ;

WMASK: 1B12+1B15                      ;
FD1:   FDO                             ; REF ZONE
ENTRY: LDA      2      FD1              ;
      LDA      1      .1                ; ATTRIBUTE:= 1B12+1B15;
      LDA      0      WMASK             ;
      CREATEENTRY                      ; CREATEENTRY('PIP', 1,
      .                                ; ATTRIBUTE);
      .                                ;
      .                                ;

```

Example 25.

#### 8.3.1.4 Procedure REMOVEENTRY.

8.3.1.4

	Call	Return
AC0	-	destroyed
AC1	-	destroyed
AC2	zone	zone
AC3	-	destroyed

The entry given by the document name in zone is, if allowed, deleted, and the slices reserved are released for other files.

Status: see table 1.

Remove the file PIP created in example 1:

```

FD2:          FDO          ; REF ZONE
      .
      .
      .
ENTRY: LDA      2          FD2      ;
      REMOVEENTRY      ;
      .
      .
      .
      .

```

Example 26.

### 8.3.1.5 Procedure LOOKUPENTRY.

8.3.1.5

	Call	Return
AC0	-	destroyed
AC1	storage area.	destroyed
AC2	zone	zone
AC3	-	destroyed

The catalog entry given by the document name in the filedescriptor is transferred to the 32 bytes area given by the word address storagearea. The entry can then be examined for its length,

attributes and optional words.

The entry format is:

```
WORD 0-2   :   Entry name
WORD 3-5   :   Optional words
WORD 6     :   Attributes
WORD 7     :   File length (in segments)
WORD 8     :   Sector address of indexblock relative to
               catalog start
WORD 9     :   Reserved length (sectors)
WORD 10-15 :   Optional words.
```

Status: see table 1.

Find the length of the file PIP created in example 25:

```
FD3:          FDO          ; REF ZONE
LENGTH = 7    ;
AREA:         .+1         ; AREA:
               .BLK      16          ;

ENTRY:        ;
              LDA        2          FD3          ;
              LDA        1          AREA         ;
              LOOKUPENTRY          ; LOOKUPENTRY('PIP',
              ; STORE);
              LDA        3          AREA         ;
              LDA        0          LENGTH,3     ; LENG:= STORE.LENGTH;
              .          ;
              .          ;
```

Example 27.

8.3.1.6 Procedure CHANGEENTRY

8.3.1.6

	Call	Return
AC0	-	destroyed
AC1	storage area	destroyed
AC2	zone	zone
AC3	-	destroyed

The storage area is destroyed after call.

The function is to change the name, filelength, attribute or optional words of the entry given by the document name in the file-descriptor.

The parameter storage area is the word address of a 32 byte memory area, with a layout as the area used in LOOKUPENTRY.

If the name has to be changed, the first 3 words of the area must hold the new name, else the first word must be set to zero, or the name must equal the filename.

If the file attribute has to be changed, word 6 of the area must hold the new attribute mask, obeying the same rules as given in CREATEENTRY. If the attribute has to be unaltered, the word must be equal to -1.

If the filelength has to be changed, word 9 of the area is defined equal to the new length, else the word must be equal to -1, meaning no change in filelength.

It is allowed to change both name, optional words, attribute and length or a mix of these in the same call. Status can be found in table 1.

Change the name of the file PIP to PAP using the same filede-  
scriptor as in example 25.

```

FD4:          FD0          ; REF ZONE
AREA:         .+1         ;
               .TXT .PAP<0><0>. ; NEWNAME
               0         ;
               0         ;
               0         ;
               -1        ; OLD ATTRIBUTE
               0         ;
               0         ;
               -1        ; OLD LENGTH
               .BLK 6    ;
               ;
ENTRY:  LDA      2      FD4  ;
        LDA      1      AREA ;
        CHANGEENTRY ;
        .
        .
        .

```

Example 28.

### 8.3.1.7 Procedure SETENTRY.

8.3.1.7

	Call	Return
AC0	-	destroyed
AC1	storage area	destroyed
AC2	zone	zone
AC3	-	destroyed

This is an extended CREATEENTRY function as all the parameters are taken from the storage area pointed out by the word address storagearea.

With this function the optional words are copied to the file entry, and they can then be used for a number of user information concerning the file.

The storage area has the same layout as shown in the LOOKUPENTRY call, and only word 7 and 8 are set by the catalog system.

If the catalogbit no. 13 (entry only) is set in the word defining the attribute mask, the length of the file is set to zero by the catalog system.

The catalog unit and entry name are taken from the first three words of storage area, i.e. the file descriptor is only used to define the user giveup action and the sub catalog.

Status: see table 1.

Create an entry PIP with the same user information in the optional words.

```

FD5:          FDO          ; REF ZONE

ENTRY:        .+1          ;
              .TXT .PIP<0><0>. ; NAME (USER SET)
              0            ;
              0            ; OPTIONAL
              0            ;
              1b12+1b15    ; ATTRIBUTES (USER SET)
              0            ; LENGTH (SET BY CAT)
              0            ; INDEX BLOCK (SET BY
                          CAT)
              10           ; RES.LENGTH (USER SET)
              .TXT .MYFILE. ; OPTIONAL (USER SET)
              .BLK 3       ;

SET:          LDA          1          ENTRY ;
              LDA          2          FD5  ;
              SETENTRY      ; SETENTRY(ZONE, ENTRY);
              .
              .
              .

```

Example 29.

8.3.1.8 Procedure INITCAT.

8.3.1.8

	Call	Return
AC0	-	destroyed
AC1	unit	destroyed
AC2	zone	zone
AC3	-	destroyed

The catalogunit given by parameter 'unit' is initialized, i.e. made accessible to the process running. During initialization, the number of free segments are counted and the catalog description in the configuration program CATW is updated. This procedure must be called before any access to the unit in question, either by the user program or by the operator by a command to the operating system if this supports disc.

The filedescriptor reference is only used for definition of the giveup procedure.

Status: see table 1.

```

FD6:          FD0          ; REF FILE DESCR

ENTRY:  LDA    2          FD6    ;
        LDA    1          .0     ; UNIT:=0;
        INITCAT          ; INITCAT(UNIT);
        LDA    1          .1     ; UNIT:=1
        INITCAT          ; INITCAT(UNIT);
        .
        .

```

Example 30.



## 8.3.1.9 Procedure NEWCAT.

8.3.1.9

	Call	Return
AC0	key	destroyed
AC1		destroyed
AC2	zone	zone
AC3	-	destroyed

This procedure creates a catalog process for the sub catalog specified by the name starting in the byte address in zone.zconv. The first 5 characters should contain the name of a sub catalog (zero characters for unused positions). The 6th. character is updated by the procedure. A sub catalog number delivered by the catalog system is inserted into this character, and the name of the created catalog process consists of all six characters. The sub catalog number is also delivered in zone.zfile.

Each sub catalog has a sub catalog key (sub-key), which is checked against the parameter 'key'. The result is delivered in zone.zblock, which thus gives an indication of the use of the sub catalog. If zone.zblock = 0, then only reading should be done. If zone.zblock <> 0, then all operations are legal.

Zone.zblock is calculated in the following way:

```

if ('key' = sub-key or sub-key = 0) then
  zone.zblock:= < 0
else if 'key' = 0 then zone.zblock:= 0;
In all other cases the status error illegal will be
returned.
```

The parameter 'key' should thus be used in the following way:

If only reading is wanted, then 'key' = 0.

If other operations will be used on the sub catalog, then  
'key' = sub-key.

A sub-key equal to zero will never result in an 'illegal' status.

NOTE The catalog system does not check the usage of a sub catalog, except for this key checking.

If zone.zconv = 0 or points to a string containing 'CAT<0><0><0>' then the operation is dummy.

#### 8.3.1.10 Procedure FREECAT.

8.3.1.10

	Call	Return
AC0		destroyed
AC1		destroyed
AC2	zone	zone
AC3	-	destroyed

The sub catalog pointed out by zone.zconv is released. The sub catalog name must be the full name including the sub catalog number as the 6th. character. If the catalog process has no more users and no more area processes for files in the sub catalog exists then the catalog process is removed.

If zone.zconv = 0 or it points to a string containing 'CAT<0><0><0>' then the operation is dummy.

Table 1 :

Status received from the catalog system after call of catalog procedures. No call of give-up-procedure is performed if operation OK, i.e. status = 0. All status received from the catalog system is marked with 1b3. Lack of status bit 1b3 and 1b4 indicates an hardware error (offline, parity etc.).

status	CREATEENTRY/SETENTRY	REMOVEENTRY	LOOKUPENTRY	CHANGEENTRY	INIT CAT
1b0 + 1b3	Cat system error	Cat system error	Cat system error	Cat system error	-
1b1 + 1b3	-	-	Entry does not exist	Entry does not exist	-
1b6 + 1b3	Parameter error* Unit undefined	File permanent Unit undefined File in use	Unit undefined	File permanent Parameter error* Unit undefined File in use	Unit undefined
1b7 + 1b3	Disc full	-	-	Disc full	-
1b11 + 1b3	Entry exist	-	-	New name exist	-
1b12 + 1b3	Map full or index block full	-	-	Map full or index block full	-

\* Parameter error : Wrong attribute, wrong size (<>), wrong name (name = 0)

## 8.4 Catalog File INPUT/OUTPUT.

8.4

When accessing data on a catalog file normal zone procedures can be used. The only difference is that the kindbit 11 must be set, which is checked by the basic I/O procedures when performing OPEN, CLOSE and block shifts.

### 8.4.1 Procedure OPEN.

8.4.1

A create area process message is send to the catalog process if zone.zconv = 0, otherwise to the sub catalog process given by the name address in zone.zconv//2. If the mode is input, the opening process is inserted as user. If the mode is output, the process is inserted as exclusive user.

If no area process exists, a free area process is taken from a common pool and initialized with the file characteristics read from the file entry.

The file can be read/written in two modes random or sequential. When sequential mode is used, the file is read/written from the last defined position, i.e. the position can be initialized or reset by a SETPOSITION command. If random mode is selected, the block number to read/write is taken from the MESS3 part of the resulting message. MESS3 is fetched from the zone record word ZBLOCK, which then must be set by the user before the block is input/output.

; AC2 equals zone address

```

LDA      0      .5      ;
OPEN                                ; OPEN(ZONE,
                                ; RANDOM INPUT);

LDA      0      .10     ;
STA      0      ZBLOCK,2 ; BLOCKNO:= 10;
INBLOCK                                ; INBLOCK(ZONE);

```

; When PUTREC is used, the procedure only makes space in the  
; zone, i.e. the block number must be specified before the next  
; PUTREC procedure causing a block transfer.  
; AC2 equals zone address (UB format).

```

LDA      0      .7      ;
OPEN                                ; OPEN(ZONE,
                                ; WRITE RANDOM)

LDA      0      .512    ;
PUTREC                                ; PUTREC(ZONE, 512);
; MOVE DATA TO BUFFER                ;
.                                     ; MOVE (DATA, ZONE);
.
.
LDA      0      .10     ;
STA      0      ZBLOCK,2 ; BLOCKOUT:= 10;
LDA      0      .512    ;
PUTREC                                ; PUTREC(ZONE, 512);

```

Example 31.

MODE/RESR	Random	Sequential
Read	5/USER	1/USER
Write	7/EXC.USER	3/EXC.USER

Fig. 37 Modes for read/write operation and reservation status.

In fig. 37 the possible modes can be found. Other modes are allowed too, but only the last three bits of the modeword is checked by the catalog system (bit 13, 14, 15), and the mode bits 8, 9 ... 12 are transferred unmodified to the disc driver in question, thus enabling use of special disc driver features as read after write, word address etc. Please consult the disc driver description for further information.

During read/write only one sector is transferred, and the user sharelength must then be defined as 512 bytes, else a blocklength error is indicated, and no data is transferred.

For each OPEN call (create area process) the user OPEN/CLOSE count is incremented by one. This count indicates how many zones the user has opened to the same file and is used by the catalog system in the decision when to remove the area process, as this happens only when all 3 user OPEN/CLOSE counts has reached zero. (Further details, see procedure CLOSE).

8.4.2 Procedure SETPOSITION.

8.4.2

The user position is set to the block parameter, which indicates number of sectors from file start, i.e. the first sector is sector 0. The file parameter is ignored. Negative block numbers are illegal and positions outside the file are rejected by a status. See table 2 for status.

8.4.3 Procedure CLOSE.

8.4.3

After normal close action, a remove area process command is sent to the catalog system or to the sub catalog process as indicated by zone.zconv, if the release parameter is nonzero. When the remove command is received the user OPEN/CLOSE count is decremented, and if zero, and no other user reservations exist, the area process is removed and included in the common pool of free area processes.

Please note that when release parameter is zero no remove area process command is send, i.e. the sequence

```
CLOSE(zone, 0);
OPEN(zone, mode);
```

increments the OPEN/CLOSE count and removal of the process is only possible, if CLOSE with nonzero release parameter is called twice.

8.4.4 Catalog Input/Output.

8.4.4

During input/output bytecount must equal 512 bytes, else the message is rejected with blocklength status. It is then a general rule that if kind bit 11 is set (which it must be when OPEN/CLOSE is used) the whole share is output/input independent of the actual state of zone pointers and values, i.e. the zrem record ele-

ment does not give the number of bytes output, except when zrem equals 512 and defines no bytes to be output. After input to a zone with kind bit 11 set, zrem is always defined to 512. This fact can cause trouble when the formats F, FB, V and VB are used in connection with disc files with the Record I/O procedures if the record size does not fit in a 512 bytes block.

Special care must also be taken, if partial updating of single file sectors is wanted, i.e. mixed read/write are performed.

```

; PROCEDURE PART_UPDATE
;
;          CALL          RETURN
; AC0     SECTOR        DESTROYED
; AC1     VALUE         DESTROYED
; AC2     ZONE          ZONE
; AC3     LINK          DESTROYED
;
; WORD 10 IN THE GIVEN SECTOR IS SET EQUAL THE VALUE GIVEN IN AC1.
; THE SECTOR IS OUTPUT AFTER MODIFICATION. BEFORE CALL THE ZONE
; MUST BE OPENED IN MODE 5 (RANDOM READ);
; AFTER RETURN THE ZONE IS OPENED IN MODE 5 AND NO BYTES DEFINED
; INPUT.

```



```

PAR00:  STA      3      PAR01      ; PART_UPDATE:
        STA      1      PAR02      ;
        STA      0      ZBLOCK,2   ; ZONE.ZBLOCK:= SECTOR
        INBLOCK                                     ;
        LDA      3      ZTOP,2     ; INBLOCK(ZONE); !ONLY
                                                ; ZTOP DEFINED!
        MOVZR    3,3                                     ;
        LDA      1      PAR02      ; WORD(ZONE.ZTOP//2):=
                                                ; VALUE
        STA      1      +10,3     ;
        LDA      1      .7        ;
        STA      1      ZMODE,2   ; ZONE.ZMODE:= 7; !WRITE!
        DSZ                                     ZBLOCK,2 ; ZONE.ZBLOCK:=
                                                ; ZONE.ZBLOCK-1;
        JMP                                     .+1      ;
        LDA      1      .0        ; ZONE.ZREM:= 0;
        STA      1      ZREM,2    ; OUTBLOCK(ZONE);
        OUTBLOCK
        LDA      0      .5        ;
        STA      0      ZMODE,2   ;
        LDA      0      .0        ;
        STA      0      ZREM,2    ;
        JMP@                                     PAR01      ;
PAR01:  0                                               ; LINK
PAR02:  0                                               ; SAVED VALUE

```

### Example 32.

The example can be used for more special update tasks if the assigning of word 10 is replaced with appropriate code.

It should be noted that the zone in question is defined with a single share, as multibuffering in random access makes no sense.

Table 2 :

Status received from the catalog system after call of

procedures OPEN, CLOSE, SETPOSITION and I/O procedures.

No call of give-up-procedure is performed if operation OK, i.e. status = 0.

All status received is marked with 1b4.

status	OPEN	CLOSE	SETPOSITION	INPUT	OUTPUT
1b0 + 1b4	Cat system error	Cat system error	Cat system error	Cat system error	Cat system error
1b1 + 1b4	Entry does not exist	-	-	-	-
1b6	Process reserved	-	-	Process reserved	Process reserved
1b6 + 1b4	Illegal*	Illegal*	Illegal*	Illegal*	Illegal*
1b7 + 1b4	No free area process	-	Position < 0	-	File not extendable
1b8 + 1b4	-	-	-	Block length error	Block length error
1b11 + 1b4	-	Not user	Not user	Not user	Not user
1b12 + 1b4	User count exceeded	-	Position outside file	-	-

\* Illegal name (= 0), file reserved for exclusive write

A. REFERENCES.

- [1] DOMAC, Domus Macro Assembler,  
User's Guide.  
Keywords: RC3600, Macro Assembler, User's Guide.  
Abstract: This paper described the RC3600 Macro Assembler Language and operation of the DOMAC macro Assembler.
- [2] RC3600 Pagings System,  
System Operators Guide.  
Keywords: MUS, Paging System, Virtual Memory, Address Mapping.  
Abstract: This manual describes, how to use the RC3600 paging system from assembly programs under the MUS system.
- [3] Extended RC3600 Coroutine Monitor.  
Programmer's Manual.  
Keywords: RC3600, Coroutine Monitor, MUS.  
Abstract: The coroutine monitor is a set of reentrant utility procedures for RC3600 MUS, with facilities for mutual synchronization and exchange of data between cooperating parallel activities. This manual is the MUS programmer's reference for the extended monitor.
- [4] XCOMX, Processor Expansion.  
User's Guide.  
Keywords: RC3600, MUS, Processor Expansion.  
Abstract: This manual describes the use of the processor expansion system, XCOMX, under the MUS monitor for the RC3600 line of computers.

- [5] RC3600 Catalog System.  
System Programmer's guide.  
Keywords: Catalog system, file system, area process,  
subcatalog.  
Abstract: This manual describes, how to use the RC3600  
Catalog System from assembler programs. Also  
the organization of the disc(s) is described.  
The user must be familiar with the MUS sys-  
tem.
- [6] DOMUS Linkage Editor.  
Keywords: DOMUS, Macro Assembler, Linkage Editor.  
Abstract: This manual describes the linkage editor for  
the disc operating system DOMUS for RC3600  
line of computers.
- [7] DOMUS, USER'S GUIDE, Part 1, Version 3  
Keywords: DOMUS, MUS, Operating System, Loader, Disc.  
Abstract: This manual describes the disc operating  
system DOMUS for the RC3600 line of compu-  
ters.
- [8] DOMUS, System Programmers Guide, Version 3  
Keywords: MUS, Operating System, Loader, Disc, Version  
3.  
Abstract: This manual describes the interface between  
assembly programs and DOMUS.
- [9] MUS Operating System, User's Guide  
Keywords: MUS, Operating System, Master Device Media  
Abstract: This manual describes the operating system S  
contained in the MUS system for the RC3600  
line of computers.

B. TERMINOLOGY.

- address An address may be a word address, which is a 15 bit unsigned integer, corresponding to a physical address in primary storage. Or it may be a byte address, which is a word address left shifted one bit and with a one added into bit 15 if the byte addressed within the word is the rightmost.
- bit A computer word consists of 16 bits, numbered from left to right:  
B0, B1, B2, .....B15.
- byte A computer word is regarded as two 8 bit bytes. The left byte B0 to B7 has an even address and the right byte B8 to B15 an odd address.
- character A character is a byte. The common alphabet within the system is the ASCII alphabet.
- text A text is a sequence of characters. Starting at a byte address and left justified. A text is terminated by a Null character with byte value zero.
- descriptor A collection of information, which describes an object, is called a descriptor. Descriptors are found as part of items and as part of zones.
- item An item is a primary storage area, which is headed by a descriptor, the first part of which usually has a standard layout. This ensures that an item always may be in some chain and possibly also in a queue. The first words of an item contain the fields:

next: next item in a queue  
 prev: previous item in a queue  
 chain: next item in a chain  
 size: the size of the storage  
       area of the item  
 name: (3 words) a text identi-  
       fying the item

field A field is a displacement, which identifies a piece of information within a descriptor. Some important fields are predefined in the system assembler, and/or in the musil compiler.

chain (linked linear list). A chain consists of a chain head and a number of chain elements. The head and each element points to the next item in the chain, the last element equals zero.

queue (doubly linked cyclical linear list). A queue consists of one or more queue elements. One of the elements is the queue head. A queue element consists of two consecutive words pointing to the next element in the queue and the previous element in the queue respectively.

When the queue is empty, the head points to itself. When an element is not in a queue, it normally points to itself.

length The term length is used to express the number of bytes contained in some storage area.

<u>size</u>	The term size is used to express the number of words contained in some storage area.
<u>program</u>	A collection of instruction and data which may be executed or accessed by one or more processes.
<u>process</u>	A sequential execution of programs under control of the monitor. All information about a process is collected in a process descriptor.
<u>monitor</u>	The nucleus of the system which implements multiprogramming, i.e. the parallel execution of several processes on a single processor.
<u>device</u>	One of a collection of units which can receive data from the processor or transmit data to the processor, often in parallel with the execution of computer instructions.
<u>driver</u>	A process executing a driver program in order to control input/output to a device.
<u>disc</u>	Any random access storage unit connected to the computer.
<u>drive</u>	A disc unit station in the system. All drives are numbered from zero and to a maximum and are administrated by the cat process.
<u>file</u>	A logical collection of data residing on a disc having a name (discfile). Sometimes we shall denote a roll of paper tape or a collection of data between two tape marks on a magtape reel as a file too.

zone

A collection of information and associated storage areas necessary to perform operation on files and devices.



C. DEVICE CODES.

C.

Decimal code	Octal code	Mnemonic	Maskbit	Device
01	01			Extended Memory
02	02			
03	03			
04	04			
05	05	ASL		Automatic System Load
06	06			
07	07			
08	10	TPI	14	Teletype Input
09	11	TPO	15	Teletype Output
10	12	PTR	11	Paper Tape Reader
11	13	PTP	13	Paper Tape Punch
12	14	RTC	13	Real Time Clock
13	15	PLT	12	Incremental Plotter
		SPC2	9	Third Standard Parallel Controller
14	16	CDR	10	Card Reader
15	17	LPT	12	Line Printer
16	20	DSC	4	Disc Storage Channel
17	21	SPC	9	Standard Parallel control- ler
18	22	SPC1	9	Second Standard Parallel controller Second Dial-up Controller
19	23	PTR1	11	Second Paper tape Reader
20	24	AMX3	2	Fourth 8 Channel Asynchro- nous Multiplexor
		TMX10	0	Second 64 Channel
21	25	TMX11	1	Asynchronous Multiplexor
22	26	TMX0	0	64 Channel Asynchronous
23	27	TMX1	1	Multiplexor

Decimal code	Octal code	Mnemonic	Maskbit	Device
24	30	MT	5	Magnetic tape
25	31	PTP1	13	Second Paper Tape Punch
26	32	TTI2	14	Third Teletype Input OCP-Function, Button Out
27	33	TTO2	15	Third Teletype Input OCP-function, Button IN
28	34	TTI3	14	Fourth Teletype Input OCP-Numeric Keyboard In
29	35	TTO3	15	Fourth Teletype Output
		DISP	7	OCP-Display
30	36			OCP-Autoload
31	37	LPS	12	Serial Printer
32	40	REC	8	BSC Controller
33	41	XMT	8	
34	42	REC1	8	Second BSC Controller
35	43	XMT1	8	
36	44	MT1	5	Second Magnetic Tape
37	45	CLP	12	Charaband Printer
38	46	FPAR	3	Inter Processor Channel Receiver
39	47	FPAX	3	Inter Processor Channel Transmitter
40	50	TTI1	14	Second Teletype Input
41	51	TTO1	15	Second Teletype Output
42	52	AMX	2	8 channel Asynchronous Multiplexor
43	53	AMX1	2	Second 8 channel Asynchro- neous Multiplexor
44	54	HLC	8	HDLC Controller
		FPAR2	3	Third Inter Processor Channel Receiver
45	55	HLC1	8	Second HDLC Controller
		FPAX2	3	Third Inter Processor Channel Transmitter

Decimal code	Octal code	Mnemonic	Maskbit	Device
46	56	CDR1	10	Second Card Reader
47	57	LPT1	12	Second Line Printer
		LPS2	12	Third Serial Printer
48	60	SMX		Synchronous Multiplexor
49	61	FDD	7	Flexible Disc Drive
50	62	CRP	10	Card Reader Punch
51	63	CLP1	12	Second Charaband Printer
52	64	FDD1	7	Second Flexible Disc Drive
53	65	LPS3	12	Fourth Serial Printer
54	66	DTC	9	Digital Cartridge Control- ler
		LPS4	12	Fifth Serial Printer
55	67	LPS1	12	Second Serial Printer
56	70	DST		Digital Sense
57	71	DOT		Digital Output
58	72	CNT		Digital Counter
				Dial-up Controller
59	73	DKP	7	Moving Head Disc Channel
60	74	FPAR1	3	Second Inter Processor Channel Receiver
61	75	FPAX1	3	Second Inter Processor Channel Transmitter
62	76	AMX2	2	Third 8 channel Asyn- chronous Multiplexor
63	77	CPU		Central Processor

D. FIRST AND SECOND PAPER TAPE PUNCH DRIVER.

D.

!0002 PP002

```

01      000001 .TXTM 1
02      000012 .RDX 10
          .TITL PP002
04      .NREL
05
06      ; ***** PAPER TAPE PUNCH DRIVER *****
07
08      DC4: ; PROGRAM:
09 000001'140401 1R0+1R1+1R7+1 ; SPECIFICATION
10 000001'000007' DC10 ; START
11 000002'000000 0 ; CHAIN
12 000003'000073 DC0-DC4 ; SIZE
13 000004'050124 .TXT .PTP<0><0>. ; NAME
14      050000
15      000000
16
17 000007'126400 DC10: SUB 1,1 ; BREAK:
18 00010'101123 MOVZL 0,0 SNC ; IF -,POWERINT. THEN
19 00011'045030 STA 1 RESERVER,2; RESERVER.CUR:=0;
20 00012'025034 DC11: LDA 1 DCDEVICE,2; START:
21 00013'006170 SETINTERRUPT ; SET INTERRUPT(DEVICE.CUR);
22 00014'006164 NEXTOPERATION ; NEXT OPERATION(MODE,COUNT,BUF,
23 00015'006171 SETRESERVATION ; +0: CONTROL,
24 00016'000437 JMP DC15 ; +1: IPR);
25 00017'041033 STA 0 DCMODE,2 ; MODE.CUR:=MODE;
26      DC12: ; NEXT CHAR:
27 00020'025026 LDA 1 ADDRESS,2 ; ADDR:= ADDRESS.CUR;
28 00021'006174 GETBYTE ; GETBYTE(ADDR,CHAR);
29 00022'006173 CONBYTE ; CONBYTE(CHAR);
30 00023'035033 LDA 3 DCMODE,2 ;
31 00024'175005 MOV 3,3 SNR ; IF MODE<>0 THEN
32 00025'000410 JMP DC120 ; BEGIN
33 00026'105320 MOVZS 0,1 ;
34 00027'127004 ADD 1,1 SZP ; PARITY:=PARITY(CHAR);
35 00030'000777 JMP .-1 ;
36 00031'175263 MOVCR 3,3 SNC ; IF MODE=EVEN THEN
37 00032'177240 ADDR 3,3 ; PARITY:=-,PARITY;
38 00033'175300 MOVS 3,3 ; CHAR:=CHAR ADD PARITY SHIFT 7;
39 00034'163000 ADD 3,0 ; END;
40 00035'005037 DC120:JSR DCDOAS,2 ; DOAS.CUR(CHAR);
41 00036'025034 LDA 1 DCDEVICE,2;
42 00037'030113 LDA 2 .32 ;
43 00040'006003 WAITINTERRUPT ; WAITINTERRUPT(DEVICE.CUR,32);
44 00041'000411 JMP DC14 ; +0: TIMER;
45 00042'011026 ISZ ADDRESS,2 ; INCR(ADDRESS.CUR);
46 00043'015027 DSZ COUNT,2 ; IF DECR(COUNT.CUR)<>0 THEN
47 00044'000754 JMP DC12 ; GOTO NEXT CHAR;
48 00045'005035 JSR DCDTA,2 ; DIA.CUR(WORD);
49 00046'020114 LDA 0 SEM ;
50 00047'125202 MOVCR 1,1 SZC ; IF WORD(15)=1 THEN
51 00050'000403 JMP DC141 ; GOTO RETURN ANSWER(EM);
52      DC13: ; DONE:
53 00051'102401 SUB 0,0 SKP ; STATUS:= 0, OR
54      DC14: ; TIMER:
55 00052'020117 LDA 0 STIMER ; STATUS:= TIMER;
56 00053'006165 DC141:RETURNANSWER ; RETURN ANSWER(STATUS);
57 00054'000736 JMP DC11 ; GOTO START;
58

```

```

0003 PP002
01          DC15:          ; CONTROL:
02          ; SET RESERVATTON(MODE);
03 00055'006172      SETCONVERSION ; SET CONVERSION(MODE);
04 00056'101005      MOV      0,0 SNR ; IF MODE=0 THEN
05 00057'0000772     JMP      DC13 ; GOTO DONE;
06 00060'034111     LDA      3 .128 ;
07 00061'055027     STA      3 COUNT,2 ; COUNT_CUR:=128;
08          DC16:          ; REPEAT
09 00062'1102400     SUB      0,0 ;
10 00063'005037     JSP      DCDOAS,2 ; DOAS_CUR(0);
11 00064'025034     LDA      1 DCDEVICE,2;
12 00065'030113     LDA      2 .32 ;
13 00066'006003     WAITINTERRUPT ; WAITINTERRUPT(DEVICE_CUR,32);
14 00067'000763     JMP      DC14 ; +0: TIMER;
15 00070'015027     DSZ      COUNT,2 ; UNTIL COUNT=0;
16 00071'000771     JMP      DC16 ;
17 00072'000757     JMP      DC13 ; GOTO DONE;
18

```

```

10000 0402
01          DC0:                ; PROCESS DESCRIPTOR:
02 00000000 0                  ; NEXT
03 00000000 0                  ; PREV
04 00000000 0                  ; CHAIN
05 00000041 00000000          DC0-DC0    ; SIZE
06 00000124 .TXT .PTR<0><0>.    ; NAME
07          050000
08          000000
09 00000102 00000000          DC0+EVENT  ; FIRST EVENT
10 00000102 00000000          DC0+EVENT  ; LAST EVENT
11 00000000 0                  ; BUFFER
12 00000000 00000000          DC0       ; PROGRAM
13 00000000 0                  ; STATE
14 00000000 0                  ; TIMER COUNT
15 00000100 1000000          PR0       ; PRIORITY
16 00000007 00000000          DC10      ; BREAK ADDRESS
17 00000073 00000000          DC0       ; AC0
18 00000000 0                  ; AC1
19 00000073 00000000          DC0       ; AC2
20 00000000 0                  ; AC3
21 00000160 00000000          DC10*2    ; PSW
22 00000000 0                  ; SAVE
23          ; OPTIONAL:
24 00000000 0                  ; BUF
25 00000000 0                  ; ADDRESS
26 00000000 0                  ; COUNT
27 00000000 0                  ; RESERVE = 0
28 00000000 0                  ; CONVERSION TABLE = 0
29 00000166 10000000          CLEAR     ; CLEAR INTERRUPT
30          00000033 00000000          DC0MODE=.-DC0    ; MODE
31 00000000 0                  ;
32          00000034 00000000          DC0DEVIC=.-DC0    ;
33 00000014 00000000          PTR       ; DEVICE = PTR
34          00000035 00000000          DC0DIA=.-DC0     ;
35 00000644 00000000          DIA 1 PTR  ; DIA(PTR,AC1)
36 00000140 00000000          JMP      +0,3  ; RETURN
37          00000037 00000000          DC0DOAS=.-DC0    ;
38 00000611 00000000          DOAS 0 PTR  ; DOAS(PTR,AC0)
39 00000140 00000000          JMP      +0,3  ; RETURN;
40          DC0:                ; TOP OF PROCESS DESCRIPTOR:
41
42          ; ***** END OF PAPER TAPE PUNCH DRIVER *****
43
44          .END DC0

```

0000 SOURCE LINES IN ERROR

14002 PP100

```

01      000001 .TXIM 1
02      000012 .RDX 10
          .TIIL PP100
04      .NREL
05      ; ***** PAPER TAPE PUNCH DRIVER *****
06
07      DC4:                                ; PROGRAM:
08      000000'140001      1B0+1B1+1      ; SPECIFICATION
09      000001'000007'    DC10           ; START
10      000002'000000      0             ; CHAIN
11      000003'000024      DC0-DC4       ; SIZE
12      000004'050124 .TXT .PTP<0>.    ; NAME
13      050001
14      000000
15
16      000007'024071 DC10: LDA      1      PROGRAM ; START:
17      000010'030410      LDA      2      .PTP      ;
18      000011'006010      SEARCHITEM      ; SEARCHITLM(.PTP,PROGRAM,RESULT);
19      000012'155005      MOV      2,3    SNR      ; IF RESULT=0 THEN
20      000013'000774      JMP      DC10    ; GOTO START;
21      000014'035001      LDA      3      PSTART,2 ; STARTADD:=START,RESULT;
22      000015'030040      LDA      2      CUR      ;
23      000016'055016      STA      3      BREAD,2 ; BREAKADDR.CUR:=STARTADD;
24      000017'001400      JMP      0,3    ; GOTO STARTADD;
25
26      000020'000021' .PTP: .+1
27      000021'050124      .TXT .PTP<0><0>. ; NAME OF MOTHER;
28      050000
29      000000

```

```

0003 PPI00
01      000031 PTP2=25      ;
02      DC0:                ; PROCESS DESCRIPTOR:
03 00024'000000      0      ; NEXT
04 00025'000000      0      ; PREV
05 00026'000000      0      ; CHAIN
06 00027'000041      DC2-DC0 ; SIZE
07 00030'050124 .TXT .PTP1<0>. ; NAME
08      050001
09      000000
10 00033'000033'      DC0+EVENT ; FIRST EVENT
11 00034'000033'      DC0+EVENT ; LAST EVENT
12 00035'000000      0      ; BUFFER
13 00036'000000'      DC4      ; PROGRAM
14 00037'000000      0      ; STATE
15 00040'000000      0      ; TIMER COUNT
16 00041'100000      1B0     ; PRIORITY
17 00042'000007'      DC10    ; BREAK ADDRESS
18 00043'000024'      DC0     ; AC0
19 00044'000000      0      ; AC1
20 00045'000024'      DC0     ; AC2
21 00046'000000      0      ; AC3
22 00047'000016"      DC10*2  ; PSW
23 00050'000000      0      ; SAVE
24      ; OPTIONAL:
25 00051'000000      0      ; BUF
26 00052'000000      0      ; ADDRESS
27 00053'000000      0      ; COUNT
28 00054'000000      0      ; RESERVER = 0
29 00055'000000      0      ; CONVERSION TABLE = 0
30 00056'100166      CLEAR   ; CLEAR INTERRUPT
31      000033 DCMODE=.-DC0 ; MODE
32 00057'000000      0      ;
33      000034 DCDEVICE=.-DC0 ;
34 00060'000031      PTP2    ; DEVICE = PTP
35      000035 DCDIA=.-DC0   ;
36 00061'064431      DIA 1 PTP2 ; DIA(PTP,AC1)
37 00062'001400      JMP +0,3 ; RETURN
38      000037 DCDOAS=.-DC0 ;
39 00063'061131      DOAS 0 PTP2 ; DOAS(PTP,AC0)
40 00064'001400      JMP +0,3 ; RETURN;
41      DC2:                ; TOP OF PROCESS DESCRIPTOR:
42
43      ; ***** END OF PAPER TAPE PUNCH DRIVER *****
44
45      .END DC0

```

0000 SOURCE LINES IN ERROR



**RETURN LETTER**

Title: MUS SYSTEM, Programming Guide, Rev.1.00RCSI.No.: 43-GL9546

A/S Regnecentralen af 1979/RC Computer A/S maintains a continual effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback, your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability:

---

---

---

---

Do you find errors in this manual? If so, specify by page.

---

---

---

---

How can this manual be improved?

---

---

---

---

Other comments?

---

---

---

---

---

Name: \_\_\_\_\_ Title: \_\_\_\_\_

Company: \_\_\_\_\_

Address: \_\_\_\_\_


Date: \_\_\_\_\_

Thank you

..... Fold here .....

..... Do not tear - Fold here and staple .....

Affix  
postage  
here

 **REGNECENTRALEN**  
af 1979  
Information Department  
Lautrupbjerg 1  
DK-2750 Ballerup  
Denmark

**RC COMPUTER**  
**A/S REGNECENTRALEN af 1979**

HEADQUARTER: LAURUPBJERG 1 - DK 2750 BALLERUP - DENMARK  
Phone: + 45 2 65 80 00 - Cables: rcbalrc - Telex: 35 214 rcbaldk