Alfaskop System 41

Operating System Reference Manual

1 Introduction to OS with Task Handling

2 OS Hardware Environment

3 OS Software Components

4 Diskette Format

5 Initialization and Logon

6 Multitasking and Interrupts

7 Internal Communications Protocol

8 User Interface Functions

9 Display Unit Functions

10 Keyboard Functions

11 FD Unit Functions and File Handling

12 Printer Unit Functions

13 Timer Functions

14 Error Handling

15 Appendices

16 Index

17

18

19

20

Ericsson Information Systems AB
Data Terminals
S-175 86 Järfälla, Sweden

E90003145E
1984-05-04
Subject to alteration without prior notice

ERICSSON

© Ericsson Information Systems AB 1984

Preface

This manual describes the operating system of Alfaskop System 41, Version 3.5.

The manual comprises a general description of the objectives and the function of the operating system, and information about how to use the operating system to develop system software.

Before studying this manual, a general familiarity with the SPL programming language should be acquired.

Alfaskop System 41 is based on the microprocessor Motorola 6800. A general comprehension of the function of this processor is most valuable for the understanding of the operating system.

CONTENTS

1. INTRODUCTION TO OS WITH TASK HANDLING

2. OS HARDWARE ENVIRONMENT

3. OS SOFTWARE COMPONENTS

4. DISKETTE FORMAT

5. INITIALIZATION AND LOGON

6. MULTITASKING AND INTERRUPTS

7. INTERNAL COMMUNICATION PROTOCOL

8. USER INTERFACE FUNCTIONS

9. DISPLAY UNIT FUNCTIONS

10. KEYBOARD FUNCTIONS

11. FD UNIT FUNCTIONS AND FILE HANDLING

12. PRINTER UNIT FUNCTIONS

13. TIMER FUNCTIONS

14. ERROR HANDLING

APPENDICES

1. Character Generators

2. List of OS Externals

INDEX

Contents

1.1	GENERAL	3
1.2	REFERENCE MANUAL OVERVIEW	4
1.3	OBJECTIVES OF THE OPERATING SYSTEM	6
1.4	PARALLELL PROCESSING CONCEPT	7
1.4.1	Tasks	7
1.4.2	Task Communication	8
1.4.3	Task States	10
1.4.4	Interrupts and Priorities	12
1.4.5	Mutual Exclusion	12
1.5	EFFICIENCY CONSIDERATIONS	13
1.5.1	Task Structures	13
1.5.2	Parameter Passing	15
1.5.3	Overlay Technique	15-16

1.1 GENERAL

This manual is primarily intended for system and application programmers.

References are from this manual made to the following documents:

SPL Reference Manual
Alfaskop System 41 Technical Description

The following documents may also be of assistance:

Alfaskop System 41 Introduction
" Terminal Console Functions and Customizing Instructions
" Reference Manual IBM Emulation
" Uniscope/UTS Emulation
" Installation and Maintenance Manual

1.2 REFERENCE MANUAL OVERVIEW

1. Introduction

   This section contains a general discussion on the objectives of the operating system and the parallel processing concepts.

2. OS Hardware Environment

   This section contains a brief description of the hardware components in an Alfaskop System 41 cluster.

3. OS Software Overview

   This section contains a brief description of the various OS software components, their interconnections and their interface to the user.

4. Diskette Software Format

   This section contains a description of the storage format on Alfaskop diskettes. Various volume types and data set types are discussed.

5. Initialization and Logon

   This section contains a description of the logon and program load procedures. Associated system data sets are briefly presented.

6. Multitasking Functions and Interrupts

   This section contains a description of how the multitasking concept can be implemented in system software. It also contains a presentation of the interrupt structure of Alfaskop System 41.

7. Internal Communication Protocol

   This section contains a description of the two-wire protocol used internally within the cluster. The internal communication is carried out by the Communication Handler which does not have any interface directly to the user.

8. User Interface Functions

This section contains a description of the User Interface Module which serves as an interface between the user and the Communication Handler.

9. Display Unit Functions

This section contains a description of the display area, the cursor handling and the message line. Some hardware dependent functions are also discussed.

10. Keyboard Functions

This section contains a description of all input functions, i.e. keyboard functions, MID functions and selector pen functions. The data structures of the keyboard tables are also described.

11. FD Unit Functions and File Handling

This section contains a discussion on the FD configurations and the diskette I/O functions. All file handling commands are described with several examples.

12. Printer Functions

This section contains a description of the OS module PRIOS, which is the interface between the user and the printer unit hardware. Print function requests and print editing are discussed.

13. Timer Functions

This section contains a description of how real time measurements can be included in a system or application module.

14. Error Handling

This section contains general description on the error handling concepts of the operating system. The error messages from OS are listed and explained.

1.3 OBJECTIVES OF THE OPERATING SYSTEM

The main objective of the operating system is to provide for efficient use of the resources of the terminal system.

The system resources are hardware resources such as CPU's, main storage, peripheral drivers etc, as well as software resources such as programs, procedures and data.

The resources are utilized by processes. The processes are in the Alfaskop terminology denoted tasks.

The following functions are performed by the operating system:

- o internal communication
- o start/restart handling
- o time supervision
- o task supervision
- o input/output supervision
- o interrupt control
- o basic error control

These operating system concepts are further discussed below.

Another objective of the operating system is to provide an interface between the hardware configuration and the system programmer.

The operating system presents a virtual machine to the programmer. This virtual machine has the same capabilities as the underlaying hardware, but it does not require the detailed understanding of the hardware's complexity.

For example, the I/O (input/output) capabilities of the hardware may require very sophisticated assembly programming. The operating system relieves the programmer of this complexity, and presents a set of input/output procedures that can be invoked by means of procedure calls with appropriate parameters.

1.4 PARALLELL PROCESSING CONCEPT

Many different activities are carried out in a terminal system: Operator interaction via the keyboard, data transmission between system units, output on peripheral units etc.

Many of these activities are performed asynchronously, i.e. the sequence is not predefined.

At a given time several activities can be initiated and not completed. This is the meaning of the parallel processing concept, although not all of the activities actually are processing concurrently. Some of them might be waiting for other activities to be completed. This is the case when, for example, one process must wait for a result computed by another process.

Each activity utilizes one or several system resources. However, most of the resourses such as CPU time, main storage cells and peripheral device drivers can only be used by one activity at a time. The asynchronous activities must be synchronized when they attempt to use the same system resource.

Obviously parallell processing requires both synchronization of asynchronous processes and inter-process communication. In Alfaskop System 41 this is achieved by introducing the "task" and the "event".

1.4.1 Tasks

The task is the logical concept of a process or an activity. A task is an entirely abstract entity, which is attached and detached dynamically.

A task can be regarded as a sequence of actions, performed by executing a sequence of instructions. The instructions are defined in program procedures. Several tasks can have their activities defined by the same procedure. A task is often implemented as an infinite loop. When a specified event occurs, the task is executed in one loop, thereafter its execution is suspended until a similar event occurs again.

A task is always associated with a superior entity, which attached the task and thereby initiated the execution of a certain procedure.

The distinction between a set of instructions (the procedure) and its execution (the task) is essential to the understanding of the operating system functions.

In Alfaskop System 41, a supervisor task is automatically initiated when power is turned on to the system. This task thereafter attaches the appropriate subtasks. The supervisor task is not terminated until power is turned off.

1.4.2 Task Communication

The "event" is the main SPL concept for communication between tasks.

The termination of a task is an implicity declared event. All other events must be declared as SPL variables of the EVENT type.

The EVENT variable can be regarded as a flag which can have either of two logical values: set and unset.

The flag is set by the statement POST name_of_event.

The flag is unset by the statement ASSIGN name_of_event.

When one task is depending on a result obtained by another task, the depending task must comprise a WAIT name_of_event or a WAIT name_of_task statement.

If data is to be interchanged between tasks, they must use a common data area, which can be accessed by both tasks.

Mr Wilson's newspaper

An example of synchronization of asynchronous processes.

Assume two asynchronous processes (tasks):
The first task is the mailman's delivery of newspapers in the morning. The other task is the fetching of Mr Wilsons newspaper from the mailbox.

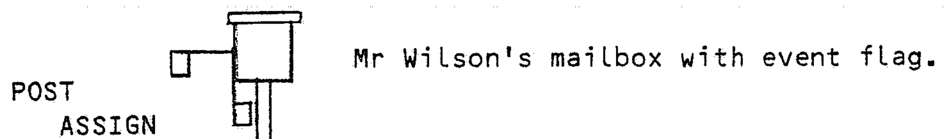
To synchronize the two tasks, we use the event "Wilsons_newspaper_arrived".

The delivery task was attached (started) before 5 o'clock in the morning.

Mr Wilson's task is attached as soon as his alarmclock sounds.
After that the following statements are executed in the task:
"Walk up to the kitchen window";
"Look at the mailbox";
WAIT Wilson's_newspaper_arrived;
The last statement causes this task to be suspended if the event has not already occured.

When the mailman delivers Mr Wilson's newspaper, the delivery task executes the following statement:
POST Wilson's_newspaper_arrived;
This statement causes the event flag to be raised at the mailbox.

The posting of the event causes Mr Wilson's task to enter the active state again. He fetches his newspaper, and probably also executes the following statement:
ASSIGN Wilson's_newspaper_arrived;
This implies that the flag is reset again.
If he doesn't assign the event, he will not be able to know when the newspaper arrives the morning after.



Mr Wilson's mailbox with event flag.

Note: The POST statement does not affect a posted event. Analogously, the ASSIGN statement does not affect an event which has not been posted. The EVENT is similar, but not equivalent, to the "semaphor" introduced by Dijkstra.

1.4.3 Task States

Each task in the system is in one of the following four states.

- Active state
  The task is in control of a processor unit and is thus currently being executed.

- Ready state
  The task is ready to enter the active state, but since it has a lower priority than the active task, the active task must first be terminated or suspended. (Priorities are discussed below.)

- Wait state
  The task is waiting for a specified event to occur, which will cause the task to enter the ready state (or, if it has the highest priority, the active state).

- Inactive state
  The task is terminated or not yet attached, but can enter the ready state (or the active state) by being attached by another task.

If only one processor is available, only one task at a time can be in the active state.

The tasks in the ready state can be regarded as forming a queue, in which the task with the lowest priority is last in queue.

Tasks in the wait state are not queued.

An illustration of the task state is found on next page.

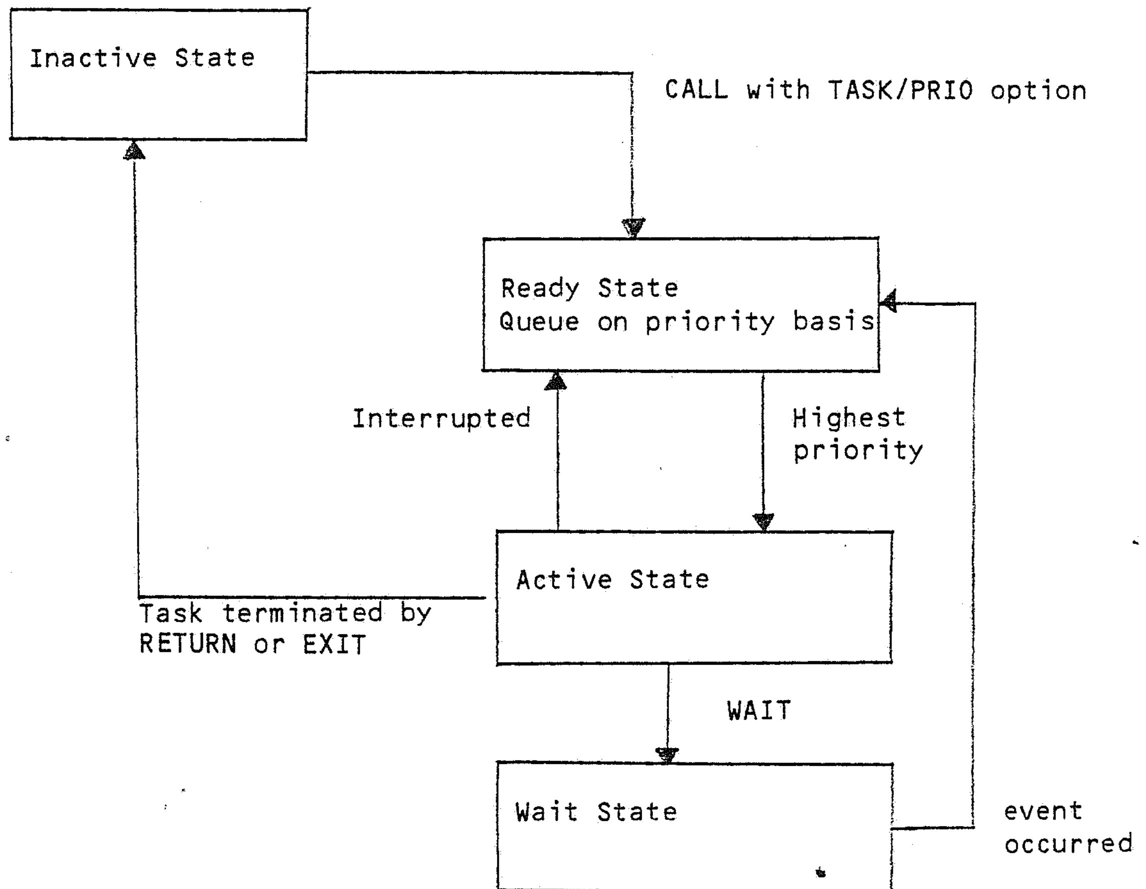


Figure 1.1 Task States

Note: A subtask can be cancelled by the attaching task. When a task is cancelled, it is immediately turned into the inactive state from the ready state or from the waiting state.

1.4.4 Interrupts and Priorities

In order to utilize the system resources efficiently, each task is assigned a priority.

Each time an event occurs, the processing task is temporarily interrupted. The interrupt can be initiated from different sources, e.g. program statements, peripheral circuits or the Reset pushbotton.

If any task was waiting for the occurred event, there are now at least two tasks ready to be executed. The task with the highest priority will enter the active state, while the other task enters the Ready state queue.

1.4.5 Mutual Exclusion

In certain cases, two asynchronous tasks cannot be permitted to interrupt each other. This is the case if the tasks use the same memory area and an interrupt would affect the calculation for both procedures.

The problem of mutual exclusion can be solved in two ways:

- An event can be used to synchronize the tasks.
- The tasks can be assigned the same priority. As a task only can be interrupted by a task having higher priority, they will not interrupt each other.

1.5 EFFICIENCY CONSIDERATIONS

The multitasking feature is normally used for system programming, but may as well be used for advanced application programming.

The task concept is very useful to implement parallel processing and pseudo parallel processing. However, there are also some disadvantages, due to the required system overhead.

- A 64-byte task control block is required for each attached task.
- A queue is maintained for all tasks in the Ready state. If many tasks are attached, the queue handling requirements will increase.
- Parameter passing between tasks increases execution time.

1.5.1 Task Structures

The same function can be implemented in several ways, with or without tasks.

If tasks are to be used, the following questions must be considered:

- Will the advantages gained by parallel processing justify the created system overhead?
- Could the number of tasks be reduced?
- Could the efficiency be increased by changing the task priorities?

The time required to enter a task into the Ready-queue is proportional to the number of tasks with higher priority in the queue. This implies that tasks which are expected to enter the Ready state frequently should be assigned a high priority.

The following two figures show how the same execution can be performed in two structures. The main task is assumed to have higher priority than the subtasks.

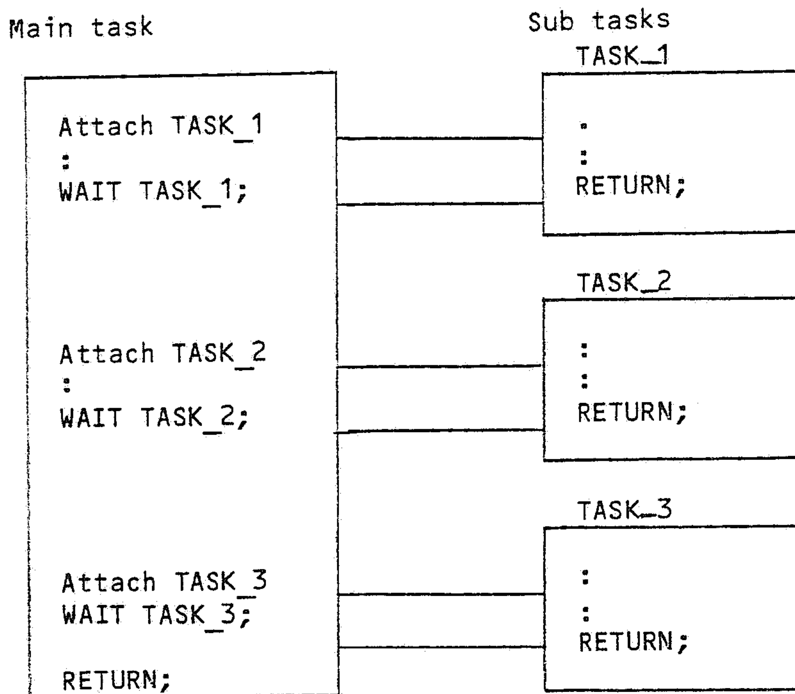


Figure 1.2 Task structure with three subtasks

The subtasks execute one procedure each, and the procedure execution is indicated by the subtask termination.

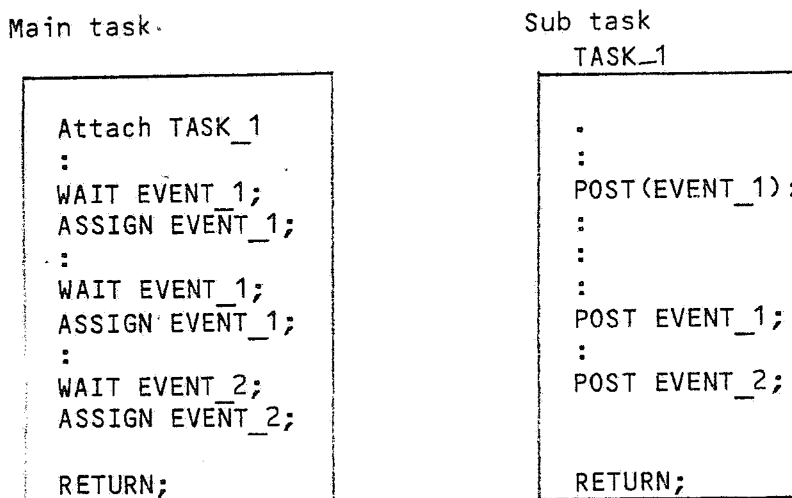


Figure 1.3 Task structure with only one subtask

The subtask may execute several procedures and post several events. Two different events are used in this case to indicate completion of the procedure executions. The subtask has been assigned a lower priority than the main task.

1.5.2 Parameter Passing

The procedure parameters in SPL are based variables. The processing of based variables increases time and program size.

The processing of SPL parameters entails overhead in both the calling procedure (where arguments are gathered into a package) and in the called procedure (where parameters occur as based variables).

In situations where parameters serve as input data to a called procedure and are not to be changed by the procedure, the overhead can be partially reduced by using arguments having the VALUE attribute (see also SPL Reference Manual).

However, the most efficient method of passing parameters is to use common variables (PUBLIC/GLOBAL, EXTERNAL) whenever possible. Moreover, this method is the only possible way of passing parameters between two tasks as well as between interrupt procedures and tasks.

1.5.3 Overlay Technique

The size of main storage is limited, and there are many program modules required to carry out a work session in a terminal system.

However, not all of the programs are used concurrently. Thus, when one program module is no longer needed, another module can be loaded into the same main storage area, thereby overlaying the part that is not needed.

The overlay technique is used by the operating system itself, and it can also be used by system and application modules. The programmer has to define the linking structure and control the loading of modules.

Overlays can be used on various levels. Similar levels are linked to the same root module as illustrated below.

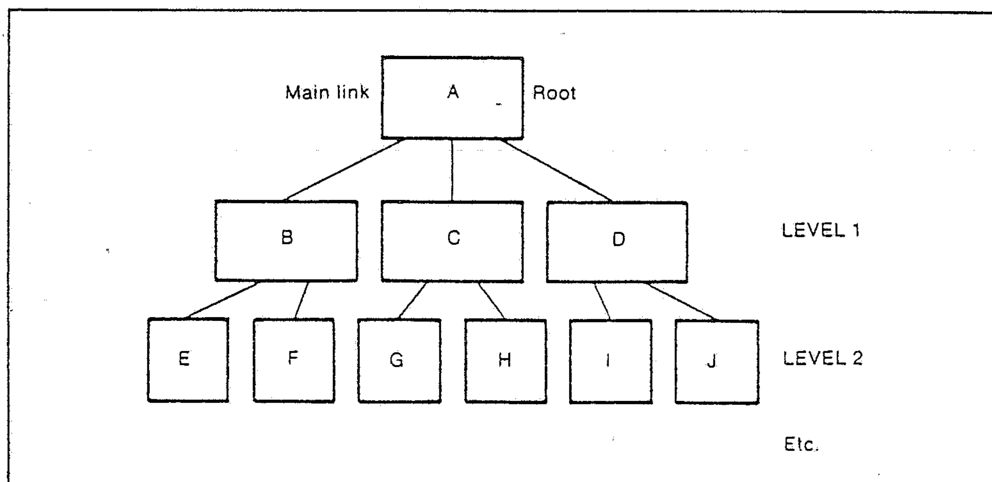


Figure 1.4 Example of overlay structure

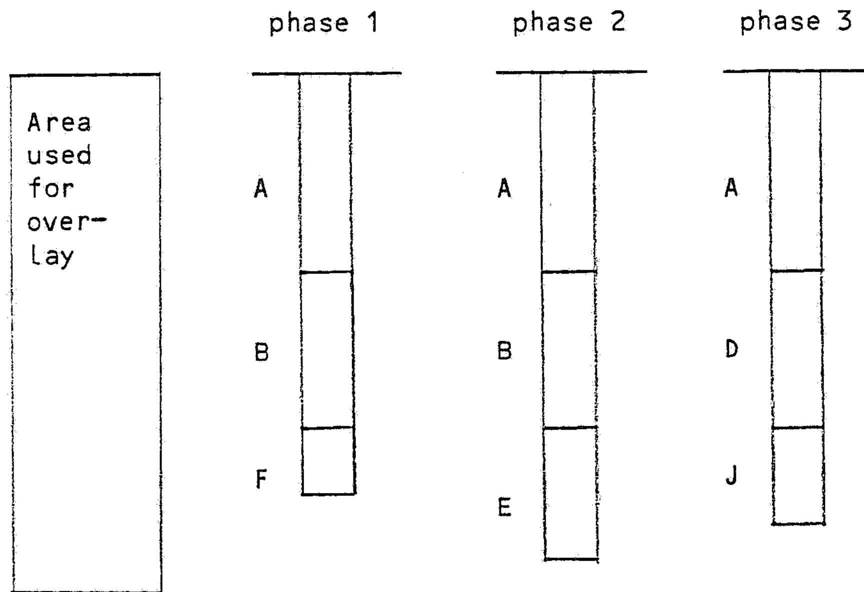


Figure 1.5 Main storage utilization in different phases of execution

Contents

2.1 GENERAL 3
2.1.1 Minimum configuration 3
2.1.2 Display Unit 5
2.1.3 Communication Processor 6
2.1.4 Flexible Disk Unit 8

2.2 MEMORY OPTIONS 8
2.2.1 MRW Option 8
2.2.2 MRO Option 9

2.3 KEYBOARD OPTIONS 9
2.3.1 MID Option 9
2.3.2 Selector Pen Option 9

2.4 PRINTER OPTION 10

2.5 PCU OPTION 10

2.6 SCC OPTION 10

2.7 DPU OPTION 11-11

2.1 GENERAL

This section contains a general description of the Alfaskop System 41 hardware, presented on the block diagram level. For information about how the hardware units function internally, see the Technical Description. The required software components for the different configurations are presented in section on OS Software Components.
Multi host configurations are not treated in this manual.

2.1.1 Minimum configuration

The minimum cluster configuration consists of one Communication Processor (CP), one Flexible Disk unit (FD) and one Display Unit (DU).

The minimum configuration of the single display unit version of System 41 consists of a display unit to which an FD is connected for program loading. The single display unit operating system is not treated in this manual.

Each CP, FD and DU contains a microprocessor and a memory. When power to the units is turned on, the operating system is loaded into each unit from the system diskette. The system diskette may also contain emulation software and some of the terminal console functions.

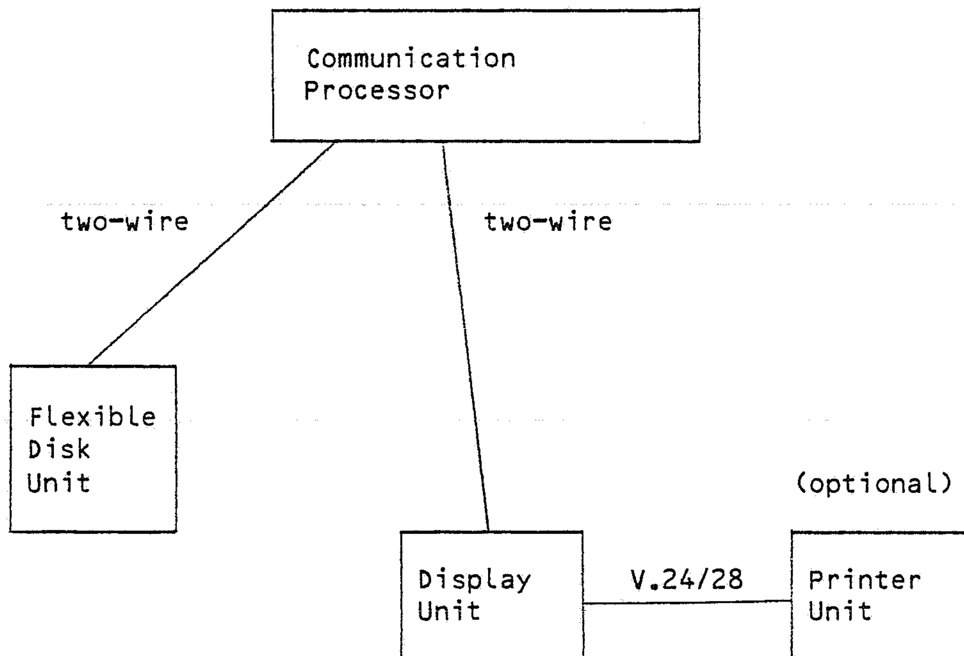


Fig. 2.1. Minimum cluster configuration.

A number of options can be selected to supplement the minimum configuration. The cluster can be expanded to contain several display units with keyboards, several printer units, FD units etc. Some of the options require the connection of adaptation boards and, in certain cases, the addition of optional program modules in the operating system.

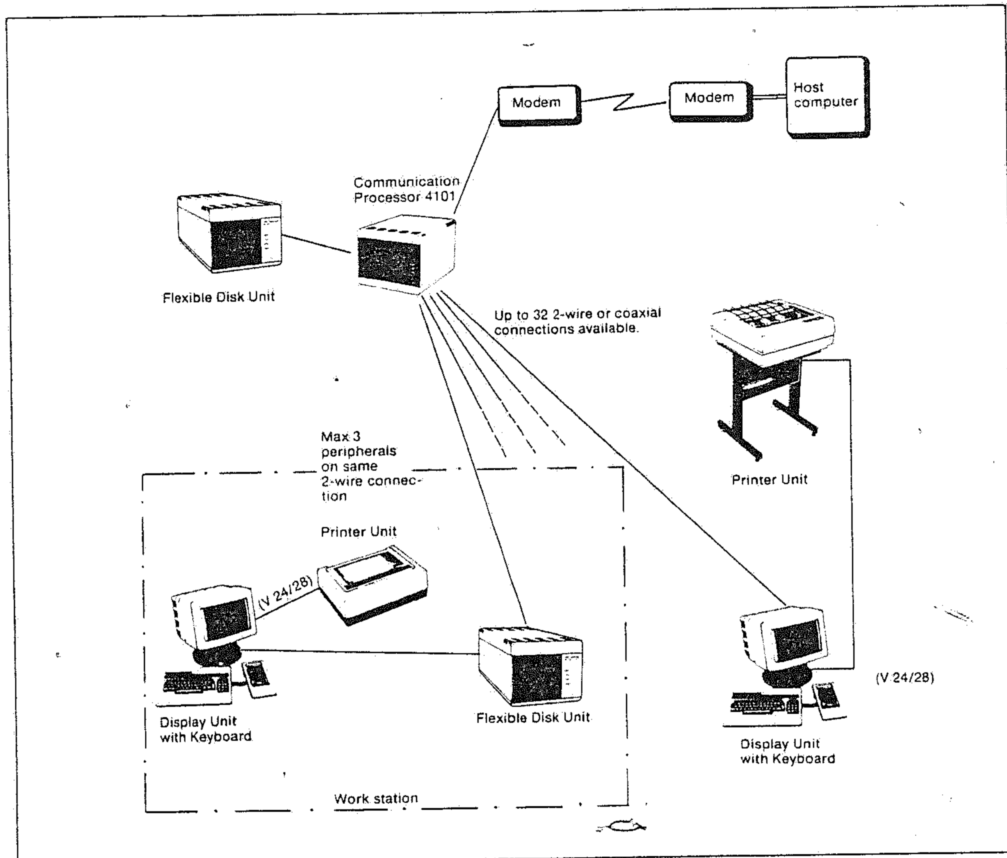


Fig. 2.2 Example of cluster configuration

2.1.2 Display Unit

Several different display units can be included in an Alfaskop System 41 cluster:

DU 4110-XXX
DU 4111
DU 4112 (4 colour)
DU 4113 (7 colour)
DU 3111

The following presentation is not specific to any particular DU model. The model characteristics are presented in section on Display Unit Functions.

The main purposes of the display units are:

- o Presentation of visual information on the Cathod Ray tube Unit (CRU).
- o Communication with other system units as communication processors, flexible disk units, printers, keyboards etc.
- o Editing and processing of data.

The main components of the display unit are:

- o Cathod Ray tube Unit (CRU).
- o Basic circuit board (DTC) containing the microprocessor, the basic memories and the basic communication interfaces.
- o Basic mechanics
- o Power supply
- o Optional circuit boards

The notation and capacity of the hardware components depends on the selected DU model.

Functionally, the DTC board can be divided into the following parts:

- o MPU - Microprocessor Unit containing the microprocessor with associated electronics for handling the bus system, interrupt system, timing etc.
- o RWM - Read Write Memory.

  The capacity of the basic RWM provided in the display unit depends on the selected DU model.

Display Unit	Basic RWM
DU 4110	32 kbyte
DU 4111	64 kbyte
DU 4112	64 kbyte
DU 4113	64 kbyte
DU 3111	64 kbyte

See also section on Memory Maps.

- ROM - Read-Only Memory. The ROM contains an IPL (Initial Program Loader) program which permits the DU operating system to be immediately loaded from flexible disk when power is turned on.

- Keyboard interface (adapter) for serial communication with a keyboard unit.

- DIA - Display Adapter for presentation on the screen of data obtained from the display memory.

2.1.3 Communication Processor

Three different communication processors can be used in an S41 cluster:
- CPR 4101 for remote clusters with up to 32 terminals
- CPR 4103 for remote clusters with up to 16 terminals (CP 4103 includes a flexible disk drive)
- CPL 4102 for local high speed channel connection to host computer

The main purpose of the communication processor is to:

- Control internal communication in the cluster

- Carry out communication between the cluster and the host computer

- Poll the terminals for status and transmission requests

- Keep track of connected units, inserted volumes etc.

The main components of a communication processor are:

- Communication Processor Board (CPB), containing the basic microprocessor, the memories and the link controller.

- Terminal Unit Adapters (TUA), basic and optional

- Communication adapters for modem connection

- Basic mechanics
- Power supply
- Optional circuit boards

Functionally, the communication processor board can be divided into the following parts:

- MPU - Microprocessor unit with interrupt control logic, timing logic, address decoding logic, memory access multiplexing (including direct memory access logic), selection logic and two-wire data link controllers.
- RWM - Read/Write Memory
- ROM - Read-Only Memory

For further details, refer to the Technical Description.

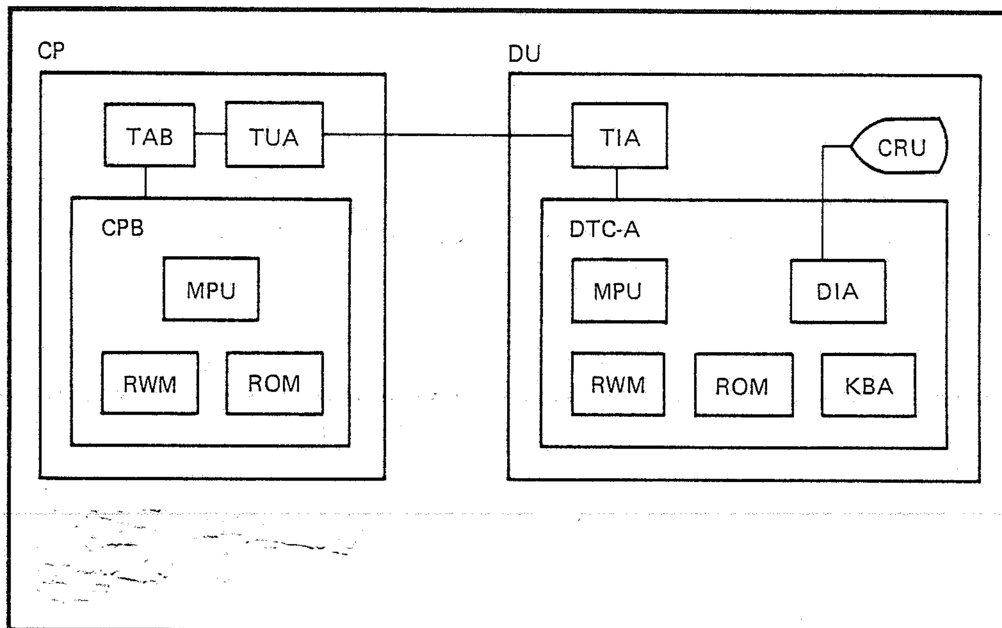


Fig. 2.3 DU - CP interconnection

2.1.4 Flexible Disk Unit

Two different flexible disk units can be used in the S41 cluster:
FD 4120 for 8" diskettes
FD 4122 for 5"1/4 diskettes

Furthermore, the Communication Processor CPR 4103 contains one flexible disk drive for 8" diskettes.

The main purpose of the flexible disk unit is to:
- o Provide program loading from flexible diskettes
- o Provide secondary data and program storage

The main parts of the flexible disk unit are:

- o Flexible Disk Processor (FDP) containing the microprocessor and the memories

- o Flexible Disk Adapter (FDA)

- o Flexible disk drive

- o Power Supply

The flexible disk processor contains:

- o MPU - Microprocessor Unit with interrupt logic
- o RWM - Read Write Memory
- o ROM - Read Only Memory with IPL program

Several FD units can be incorporated in the cluster, although only one can be the system-FD, i.e. contain the system diskette. If supplementary FD units are included in the cluster, they are used as data-FD units. They provide loading and storage of application programs and data.

The hardware of a system-FD is identical to the hardware of a data-FD. The FD unit which contains the system diskette when power is turned on becomes the system-FD of the cluster.

Note that FD 4122 cannot be used as system-FD.

2.2 MEMORY OPTIONS

2.2.1 MRW Option

If the read-write memory on the basic circuit board does not suffice, an additional capacity can be obtained by equipping the unit with an MRW board. (See also Section on Memory Maps). This option is used for DU 4110 and CP and can be equipped/strapped for 16, 20, 24 or 28 K byte read/write memory.

2.2.2 MRO Option

For systems without an FD, software may be stored in PROMS on the MRO board. This option contains a maximum of 24 K bytes read only memory and 256 bytes of battery backup RWM. This option is used for DU single and CP and is not supported by the operating system. (See also section on Memory Maps).

2.3 KEYBOARD OPTIONS

Keyboard units are connected to the display units to provide for operator entry and terminal interaction. The adaptation unit for keyboard communication is included in the basic configuration of the DU.

Alfaskop System 41 comprises the following keyboard units:
Keyboard Unit 4140 with
Keyboard Expansion Unit 4141 (optional)
Keyboard Unit 4143 with
Keyboard Expansion Unit 4146 (optional)

The keyboard unit contains a key matrix, a microprocessor and communication interfaces for connection to the display unit and to the optional Magnetic Identification Device (MID).

The purpose of the keyboard logic is to:

- o Scan the key matrix and to store the key numbers of the depressed keys in a keyboard buffer

- o Maintain communication with the display terminal controller in the DU.

- o Read ID-data if a MID is connected and an ID-card is inserted.

2.3.1 MID Option

A magnetic identification device can be connected to the keyboard. The MID is used to check the authorization of personnel using the system. The authorization check is normally performed in the host computer, but it can also be implemented in Alfaskop system software.

2.3.2 Selector Pen Option

The Selector Pen Device 4130 can be connected only to Display Unit 4110-003. The selector pen is used to select predefined fields on the screen, thus providing simple input to the display unit without using a keyboard.

To support the selector pen, the display unit must be equipped with a Selector Pen Adapter (SPA).

2.4 PRINTER OPTION

One Printer Unit (PU) can be connected to each display unit (except DU 4113) in the cluster via a V24/28 interface. The display unit must be equipped with an Asynchronous Communication Adapter (ACA).

Alternatively, the printer unit can be connected to the communication processor via a Peripheral Control Unit (PCU), equipped with an asynchronous communication adapter.

One DU with a PU connected, one FD and one PCU with a printer can be connected to the same port on the communication processor. All PUs and PCUs must have different logical addresses.

2.5 PCU OPTION

Peripheral Control Unit (PCU) is used to connect a peripheral device, like a printer or a modem, to the cluster. The PCU is connected to the communication processor via a two-wire cable.

Functionally, the PCU can be divided into two main parts:

- o GPB - General Processor Board
- o ACA/SCA - Asynchronous Communication Adapter for printer connection, or Synchronous Communication Adapter for modem connection.

Several emulations and program products require a PCU.

2.6 SCC OPTION

The Synchronous Communication Controller is used to control serial synchronous data communication. The main microprocessor is thus relieved of message formatting. SCC is intended for SNA/SDLC and other advanced protocols.

SCC communicates on the line via a V24/28 or X21/24/27 interface.

A communication processor can incorporate one or two SCC units. SCC can also be included in a display unit operating in a single display unit configuration.

SCC is arranged on two circuit boards designated SCC-1 and SCC-2.

The SCC-1 board contains a microprocessor, an IPL-PROM and communication control circuits. The SCC-2 board contains a memory area, partly shared by the main processor in the CP or the DU.

2.7 DPU OPTION

Display Processor Unit 4173 is used only in emulations of IBM 3178. The configuration is called WS 3111 and consists of a monitor, a DPU 4173 and a keyboard.

Contents

3.1	GENERAL	3
3.1.1	System Overview	3
3.2	BASIC OS MODULES	5
3.2.1	OS Request Handler	5
3.2.2	OS Interrupt Handler	5
3.2.3	Task Manager	5
3.2.4	Timer Handler	5
3.2.5	SPL Support	5
3.3	LOAD MODULES	6
3.3.1	IPL	6
3.3.2	NIP	6
3.3.3	Map Loader	6
3.4	LOGON MODULES	6
3.4.1	Logon Handler	6
3.4.2	Logon Supervisor	6
3.5	INTERNAL COMMUNICATION MODULES	7
3.5.1	Communication Handler	7
3.5.2	User Interface Module	7
3.6	DISPLAY FUNCTION MODULES	7
3.6.1	Display Handler	7
3.6.2	Message Line Handler	8
3.7	INPUT FUNCTION MODULES	8
3.7.1	KB/MID/SP Handler	8
3.8	FD FUNCTION MODULES	8
3.8.1	FDIOS	8
3.9	PRINTER FUNCTION MODULES	9
3.9.1	Printer Handler	9
3.9.2	PRIOS	9
3.9.3	Print Editor	9
3.10	UTILITY SUPERVISOR	9
3.11	TERMINAL CONSOLE FUNCTIONS	10
3.12	MEMORY REQUIREMENTS	10
3.12.1	Page Area	10
3.13	MEMORY MAPS	11-12

3.1 GENERAL

This chapter presents the software components of the operating system in Alfaskop System 41. The general function of each component is briefly described.

How to use the operating system components is further discussed in the following chapters.

3.1.1 System Overview

The simplified figure below shows the operating system modules in a cluster configuration.

Some of the OS modules are optional and can be omitted to reduce OS memory requirements.

The main part of the operating system is loaded into the hardware devices from the system diskette when power is turned on to the unit in question. Some OS components are permanently stored in ROMs.

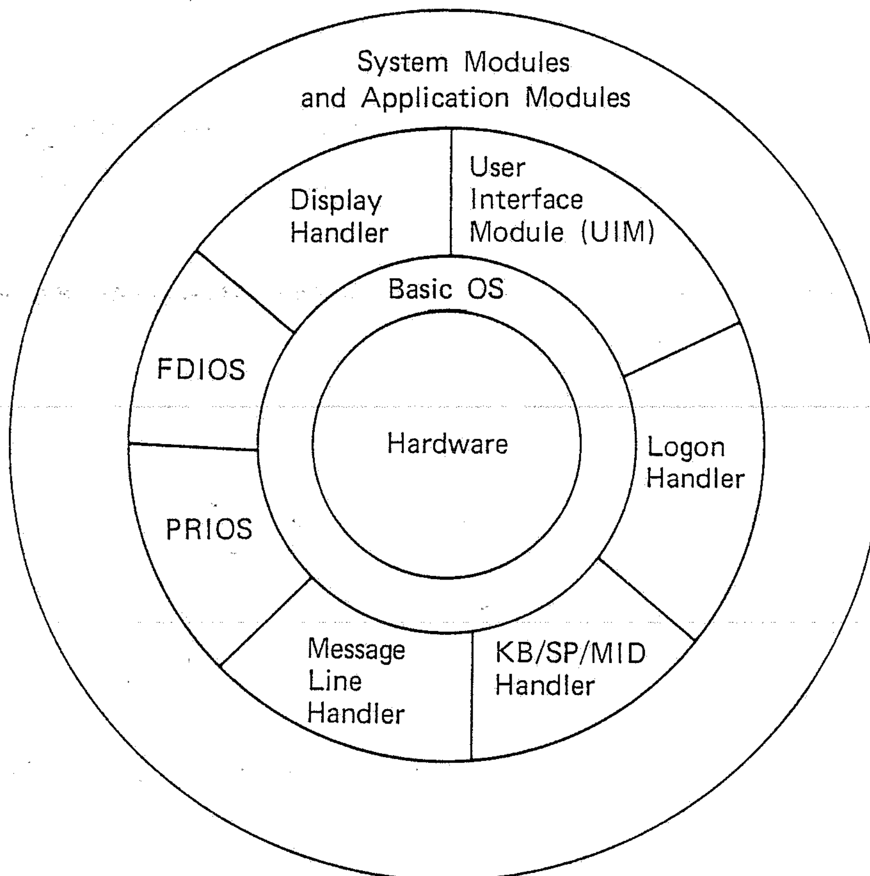


Figure 3.1 System overview in DU

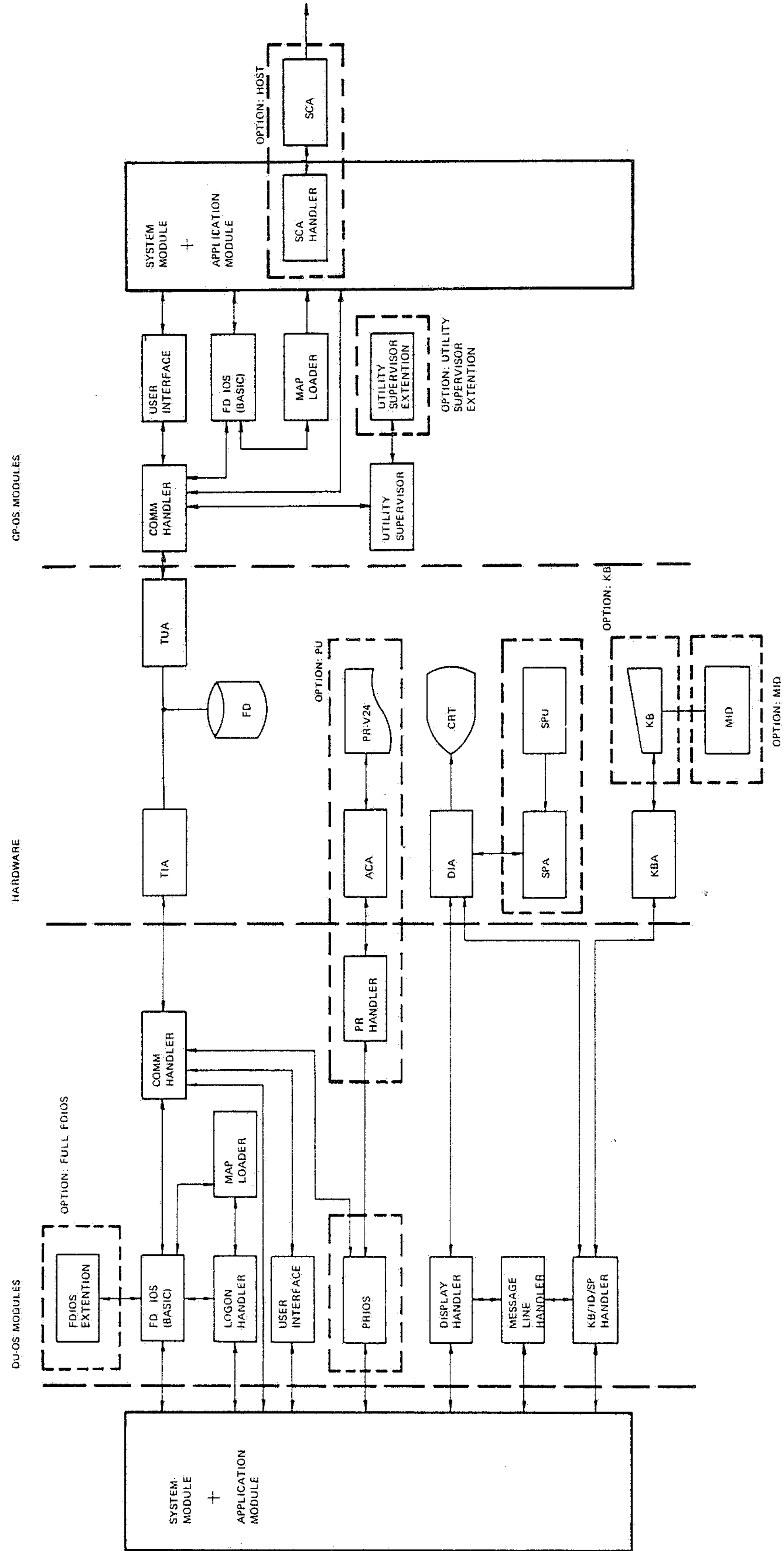


Figure 3.2 A diskette contains a number of data sets

3.2 BASIC OS MODULES

The following modules are not associated with any particular unit or function, but are essential parts of the operating system.

3.2.1 OS Request Handler

This module comprises an interrupt procedure for software interrupts (SWI). The OS Request Handler issues calls to procedures in Task Manager and FDIOS as per the parameters in the OS Request call.

The SPL statements CALL, WAIT, POST etc. are translated by the compiler to an OS request with suitable parameters. The appropriate procedure in Task Manager is invoked by the OS Request Handler to execute the function.

3.2.2 OS Interrupt Handler

This module handles the queues for processing at the lowest IRQ interrupt level, INTERRUPT.

3.2.3 Task Manager

Task Manager procedures carry out two main functions: They handle the SPL statements for multitasking, and they maintain the task queues.

Task Manager maintains queues of tasks in the ready state and allocates processor time to the task with the highest priority on the next interrupt.

3.2.4 Timer Handler

The Timer Handler module is a group of procedures carrying out software timing in the system. The user can also request time counters and have an event variable set when the specified amount of time has expired.

3.2.5 SPL Support

The SPL Support modules contain general procedures which are used to implement a number of SPL (System Programming Language) functions which comprise:

- Operations on character strings and arrays.
- Multiplication and division.
- Translation of unit names to unit numbers for internal presentation.
- Locking and unlocking of interrupts.

3.3 LOAD MODULES

3.3.1 IPL

The Initial Program Load module is stored in PROM. IPL handles the loading of the operating system after power is turned on or after the RESET pushbotton has been depressed.

3.3.2 NIP

The Nucleus Initialization Procedure starts up the operating system after IPL is completed. NIP consists of a resident part and an overlay part.

3.3.3 Map Loader

The Map Loader is used for automatic program loading, e.g. loading of a predefined set of emulation parameters for the DU emulation. Map loading is performed from flexible disk according to specifications in a list called Load Map.

3.4 LOGON MODULES

3.4.1 Logon Handler

When power is turned on and the operating system is loaded into the display unit, the Logon Handler awaits further commands, either from the operator or from an autologon definition.

The logon name is passed to the Logon Handler to define which emulation or application is desired for program load.

3.4.2 Logon Supervisor

The logon procedure is supervised and controlled by an operating system module called Logon Supervisor.

For more detailed information about the logon and program load procedures, refer to section on Initialization and Logon.

3.5 INTERNAL COMMUNICATION MODULES

3.5.1 Communication Handler

Internal communication via the two-wire connection between the different system units (CP, DU, FD, PCU) is performed by the Communication Handler module.

The following main functions are carried out:

- Polling from CP to the other units connected to the two-wire circuit in order to ascertain their current status and whether or not they need data transmission.

- Sending and receiving of messages

- Administration of traffic on the two-wire circuit.

The Communication Handler has no direct interfaces with the system modules. The interface is constituted by the User Interface Module.

3.5.2 User Interface Module

The User Interface Module provides the connection between the system module and the Communication Handler.

The User Interface has two main purposes:

- To handle the internal communication protocol.

- To provide an interface to the system and application modules.

3.6 DISPLAY FUNCTION MODULES

3.6.1 Display Handler

The Display Handler contains procedures which can be called by the system modules for manipulating the hardware in DIA (display interface adapter). The Display Handler functions comprise cursor positioning and (re)definition of the display configuration (number of lines, line lengths etc.).

Presentation on the screen is handled by the hardware in DIA. Reading and updating of the screen content are carried out by a conventional memory reference procedure. The Display Handler supplies global variables which define the current screen layout.

3.6.2 Message Line Handler

The Message Line Handler:

- Administers presentation of messages from the system modules on the message line.
- Handles, via the KB/ID/SP Handler, input from KB to the message line.
- Carries out certain limited editing of data entered into the message line.

3.7 INPUT FUNCTION MODULES

3.7.1 KB/MID/SP Handler

The KB/MID/SP Handler takes care of operator input functions to the screen. It consists of three main parts:

- Procedures used for communication with KBU for entering key data and MID data as well as controlling the indicators on the keyboard.
- Procedures for buffering the input data.
- Procedures which can be called by the user to control input and indicators.

3.8 FD FUNCTION MODULES

3.8.1 FDIOS

The FDIOS module in DU contains one part of the logic used by the file handling system in System 41. However, most of this logic is in the FD software system.

FDIOS consists of two main parts:

- Routines that handle communication (via the Communication Handler) with FD via a two-wire connection.
- Routines that execute the SPL statements for file handling. The compiler translates a file handling statement into an OS Request with suitable parameters. A procedure in FDIOS which initiates execution of the desired function is invoked by the OS Request Handler.

The basic version of FDIOS contains the communication parts and software support for certain file handling statements, namely ASSIGN, OPEN, CLOSE, READ, and REWRITE for unbuffered I/O.

By adding the FDIOS Extension module, complete file handling is obtained.

3.9 PRINTER FUNCTION MODULES

3.9.1 Printer Handler

The Printer Handler in DU carries out the hardware dependent printer functions. Communication between DU (PCU) and the printer via the V24 interface is controlled by the Printer Handler.

The Printer Handler receives fully edited data from PRIOS and the system modules.

3.9.2 PRIOS

PRIOS provides the interface between the system module and the Printer Handler. PRIOS is a task in DU which can be attached by the system modules to perform different printer output functions. The desired function is specified in a common parameter list.

3.9.3 Print Editor

The Print Editors are not part of the operating system, but must be linked with system modules which are intended to use PRIOS.

3.10 UTILITY SUPERVISOR

The Utility Supervisor contains routines to dump and update the memory of the unit, in which the supervisor is located. Requests to perform these operations are issued from supervisors in other units, where the requesting supervisor is part of the system module.

The Utility Supervisor is used by the Console Mode software, when customizing and maintaining the terminal system.

The Utility Supervisor in CP maintains the list of mounted diskettes in the cluster.

3.11 TERMINAL CONSOLE FUNCTIONS

The Terminal Console Functions are used to define the system configuration and to carry out diskette handling, fault tracing etc.

Terminal Console Functions are not part of the operating system. The Customizing procedure and the Terminal Console Functions are described in separate documents.

3.12 MEMORY REQUIREMENTS

The amount of RWM used by the operating system varies somewhat, depending upon the system configuration and selected options.

The ROM contains the permanent Initial Program Load (IPL) module, which causes the program loading from the system diskette to be immediately started when power to the units is turned on, or when a reset button is pressed.

Some of the RWM areas into which the operating system is loaded are used for overlays. This implies that appropriate modules of the system software, or the application software, is loaded into the RWM when being requested. When an overlay module is loaded, the previous content of the affected memory section is destroyed, and the content must be reloaded if needed again.

If a printer is to be connected to a DU 4110, a MRW memory extension board of at least 16 kbyte is required.

Overlay technique is used for initial program loading of the NIP module and the Logon Handler. This workspace of 8.4 K bytes is overlayed by the reminder of OS. An overlay procedure is also used by FDIOS.

3.12.1 Page Area

The page area is the first 256 bytes in main storage. Addresses to the page area are specified by one byte only. This feature of the page area is used to reduce program code and CPU time.

The greater part of the page area in a DU is permanently occupied by the operating system. The remainder can be used by the emulation or application modules. Refer to the SPL Reference Manual.

3.13 MEMORY MAPS

The following figures presents the usage of the 64 K byte memory space in the system units.

Note that the selected options affect the memory requirements.

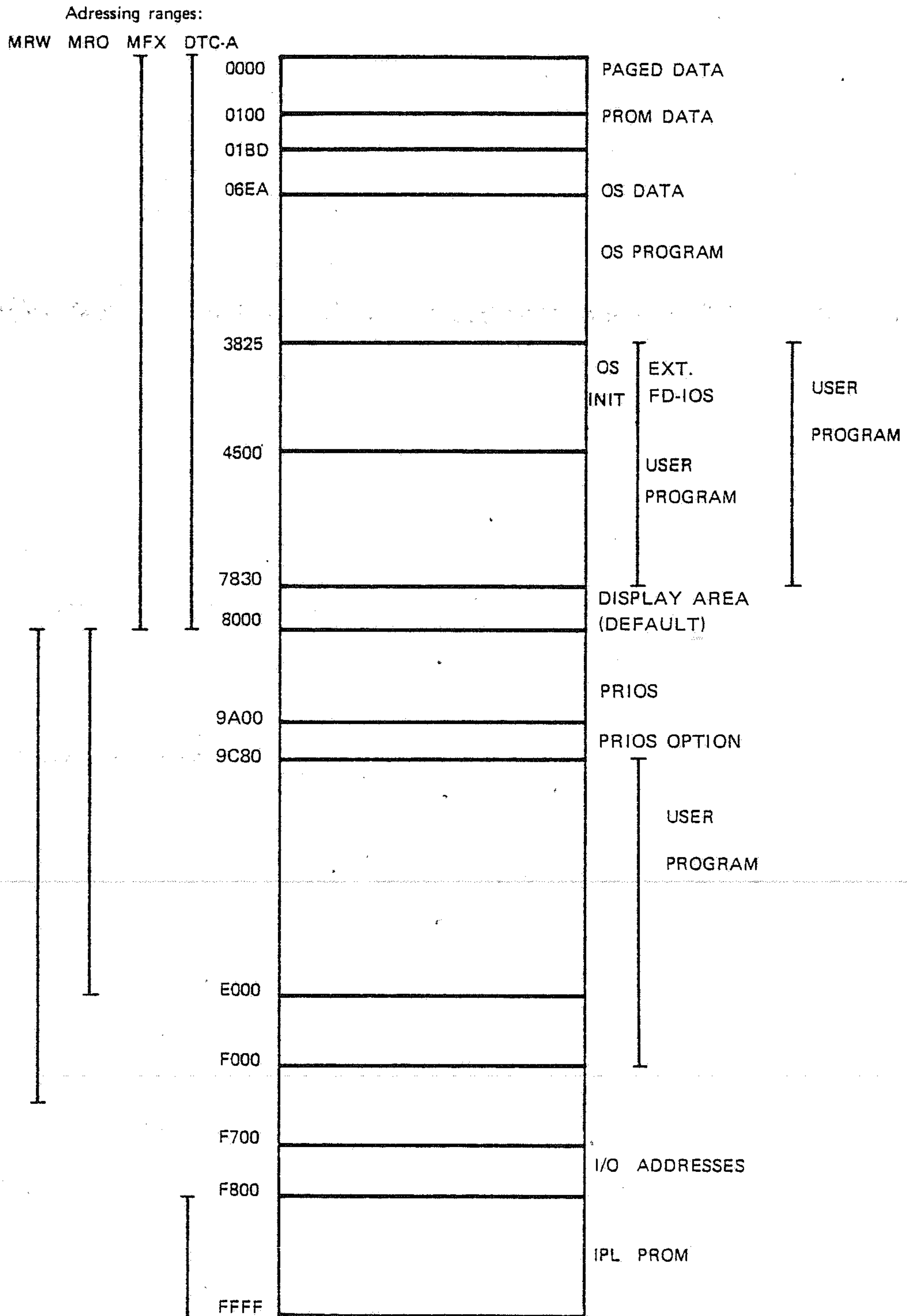


Figure 3.3 Memory map of DU 4110/4111/4112

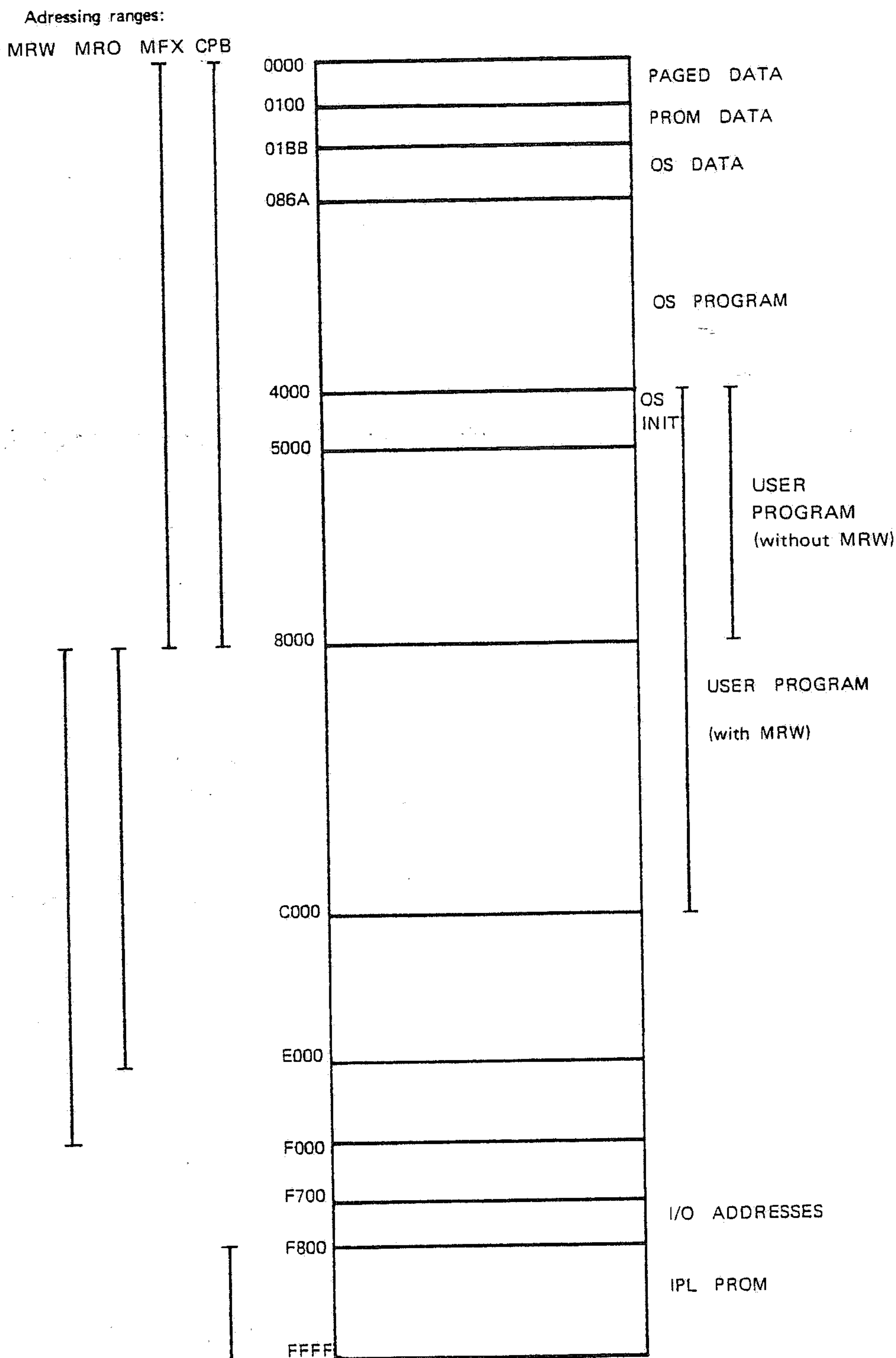


Figure 3.4 Memory map of CPR 4101/4103

Contents

4.1	GENERAL	3
4.2	PHYSICAL CAPACITY	3
4.2.1	8" Diskettes	3
4.2.2	5"1/4 Diskettes	4
4.3	DATA ORGANISATION	5
4.3.1	Organisation Overview	5
4.3.2	Library Organisation	6
4.4	VOLUME TYPES	8
4.4.1	Empty Diskette	8
4.4.2	Data Diskette	10
4.4.3	System Diskette	11
4.4.4	PC Diskettes	12
4.5	VOLUME LABEL (VOLLAB)	13
4.6	VOLUME TABLE OF CONTENTS (VTOC)	15
4.6.1	General FDE Format	16
4.6.2	Special FDE Format	17
4.6.3	SYSFDE Types	19
4.7	LIBRARY TABLE OF CONTENTS (LTOC)	20
4.8	DATA SETS TYPES	21-21

4.1 GENERAL

This chapter describes the storage format used for diskettes in Alfaskop System 41.

This chapter also presents the tables and labels of the volume which are normally used only by the operating system. Since System 41 enables the user to read and write these items from the system modules, they are presented in detail. However, system information in labels and tables must be handled with care.

File handling is presented in section on FD Functions and File Handling.

4.2 PHYSICAL CAPACITY

Two types of flexible disks can be used in Alfaskop System 41 - the single sided 8" diskette complying with IBM 3740 standard format, and the double sided 5"1/4 diskette used in FD 4122.

4.2.1 8" Diskettes

Tracks	The diskette is divided into 77 concentric tracks numbered from 0 to 76. Track 0 is at the outer edge of the diskette.
Sectors	A track is divided into 26 sectors containing 128 bytes each.
Capacity	An 8" diskette can thus store 256 256 bytes (77 x 26 x 128 = 256 256 bytes).
	The physical storage format complies with the IBM standard (compatible with IBM 3740).

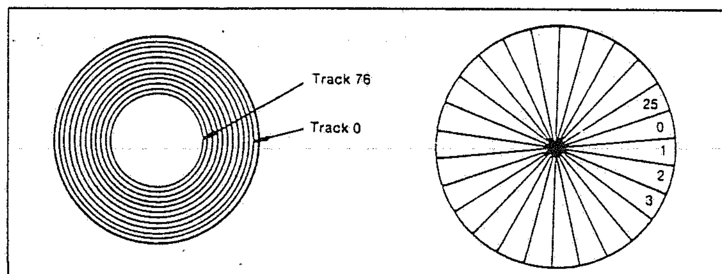


Figure 4.1 Tracks and sectors on the 8" diskette.

4.2.2 5"1/4 Diskettes

Tracks — The diskette is on each side divided into 80 concentric tracks numbered from 0 to 79. Track 0 is at the outer edge of the diskette.

Sectors — A track is divided into 9 sectors containing 512 bytes each.

Capacity — A 5" 1/4 diskette can thus store 2 x 80 x 9 x 512 = = 720 kbytes.

The storage format complies with the IBM PC standard.

4.3 DATA ORGANISATION

A collection of external data items is called a data set.

Data in a peripheral storage module (e.g. a diskette) is called a volume. A volume can contain one or more data sets.

The items within a data set are normally arranged in distinct physical groupings called blocks. A block is the smallest entity which can be brought into main storage during an input operation or transferred from main storage during an output operation.

The record is the smallest logical entity which can be referred to by a procedure making input/output requests. A block can comprise one or more complete records; only fixed-length records are supported by the operating system.

To allow a module to deal primarily with the logical aspects of data rather than with its physical organization a logical entity called a file is used. A file consists of one single data set. The programmer may regard a file as a contiguous area containing records. File handling is further discussed in section on FD Functions and File Handling.

4.3.1 Organisation Overview

Data stored on the diskette is arranged in a number of data sets. The data sets are described by control information in labels and tables on the diskette.

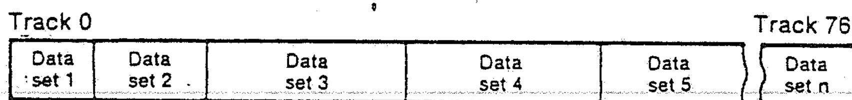


Figure 4.2 A diskette contains a number of data sets

Each diskette contains a volume label (VOLLAB), which presents information about the diskette, its identification, version number etc.

Each diskette also contains a volume table of contents (VTOC). The VTOC consists of a number of entries designated File Directory Entries (FDEs). Each FDE contains information about one data set on the diskette.

Several data sets can be collected in a library. Data sets collected in a library are called members.

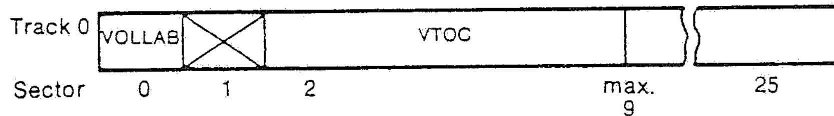


Figure 4.3 Track 0 contains VOLLAB and VTOC

4.3.2 Library Organization

The library concept supports data sets arranged in a 2-level hierarchy. This makes it possible to handle a number of data sets as one unit.

A library consists of a number of data sets immediately preceded by a directory. The directory is called Library Table of Contents (LTOC) and is built up analogously to the VTOC.

The members are described by elements called Member Director Entries (MDE). These are built up in the same way as the FDEs. Certain user-specified information can be stored in and retrieved from an MDE. This information is not processed by the I/O system.

A library member is a data set which differs from ordinary user data sets in that its directory is kept in an LTOC instead of a VTOC. A library cannot be a member in another library.

Modules such as source code modules, object code modules and load modules are mostly stored in libraries.

The VOLLAB, VTOC, FDE, LTOC and MDE are all described in greater detail in the following sections.

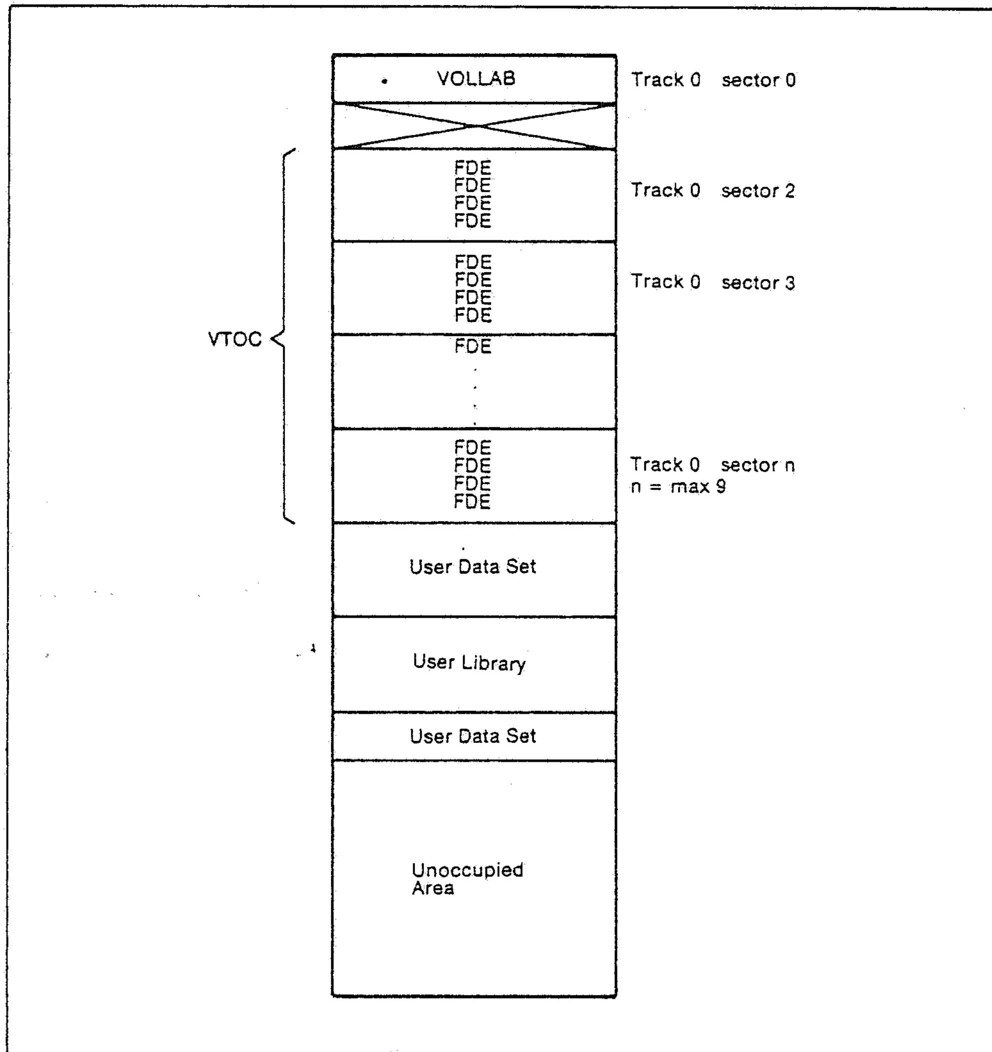


Figure 4.4 Diskette volume organization

4.4 VOLUME TYPES

System 41 recognizes four main types of diskettes: system diskettes, data diskettes, personal computer diskettes and empty diskettes.

4.4.1 Empty Diskette

5"1/4 diskettes are unformatted when delivered. 8" diskettes that are to be used in System 41 are formatted and initialized prior to delivery from EIS.

The diskette is provided with a volume label (VOLLAB) and a mini-VTOC. By using the Allocate Volume function in Console Mode, the user can create a diskette according to his requirements.

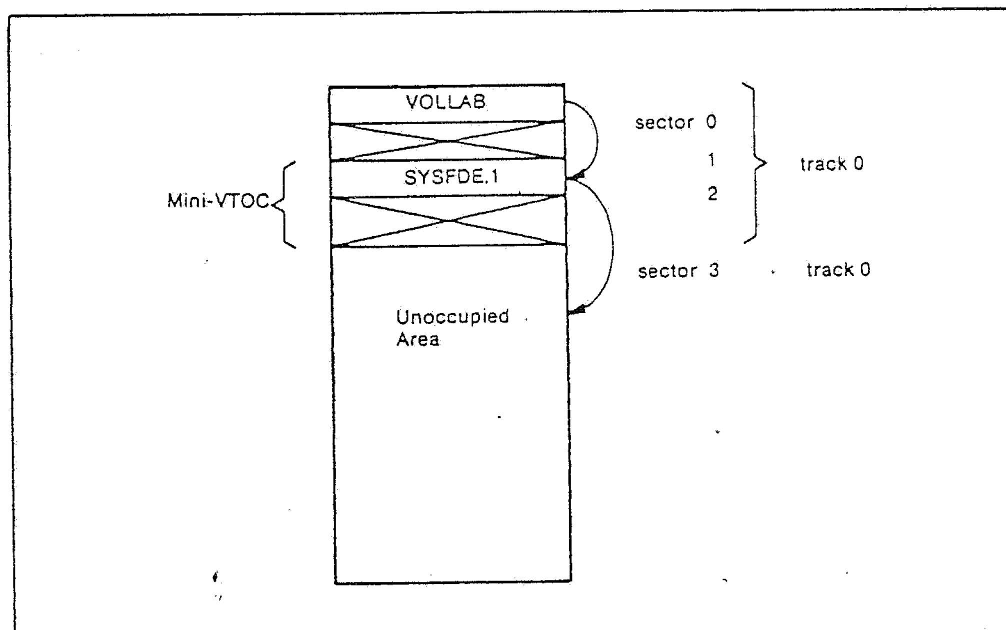


Figure 4.5 Initialized empty diskette

An empty 8" diskette as delivered from Ericsson is initialized as follows:

- o The volume label (VOLLAB) contains a pointer to a mini-VTOC (Volume Table of Contents).
- o The mini-VTOC contains only one data set directory entry (FDE), namely SYSFDE, which is a pointer to the unoccupied area at the end of the volume.

A system/data diskette can be converted to an empty diskette using the delete function in Console Mode. The diskette type is then marked empty, but VOLLAB and VTOC remain.

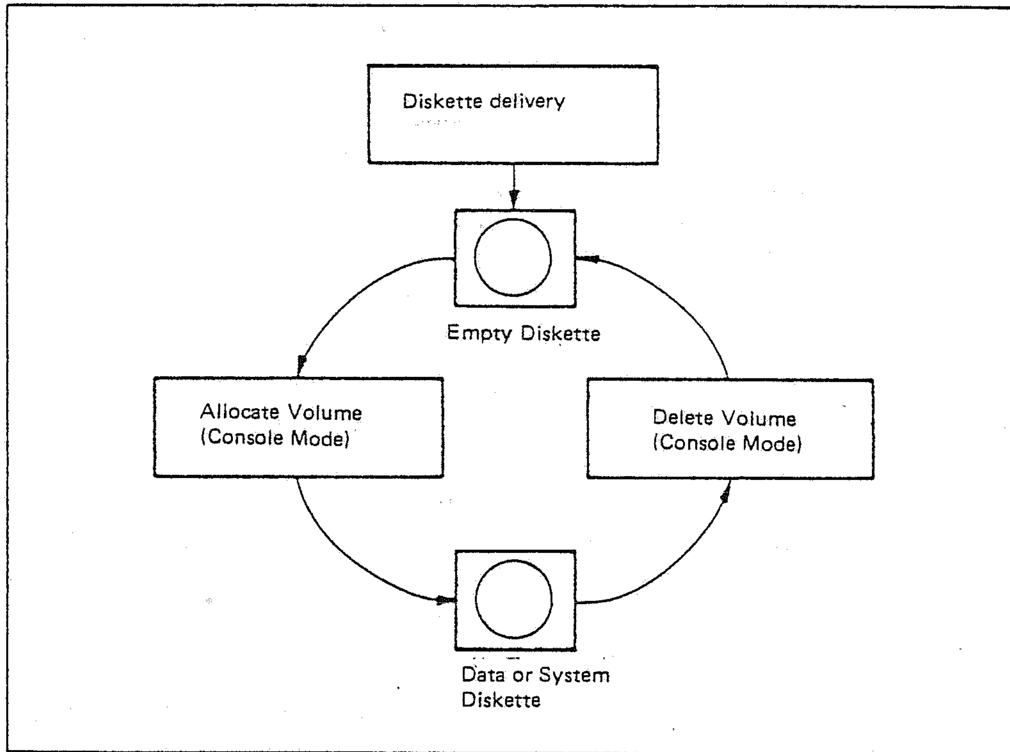


Figure 4.6 Allocating and Deleting diskettes

4.4.2 Data Diskette

A data diskette is created from an empty diskette by using the Allocate function in Console Mode. A data diskette contains Alfaskop program products or user defined data.

Before any data sets have been created, the diskette is marked as empty. The diskette VTOC contains a pointer to the unoccupied area at the end of the volume, and a number of unused data set directory entries. Successively, as data sets and libraries are created on the diskette, the unused FDEs are replaced by FDEs pointing to the created data sets.

The figure below shows how a data diskette can be arranged when a number of different data sets have been defined.

- VOLLAB contains a pointer to VTOC.
- VTOC contains data set directory entries (FDEs) for the user data sets and the user libraries on the diskette. VTOC also contains a pointer to the unoccupied area at the end of the volume.
- LTOC contains member directory entries (MDEs) for data sets included in the user library in question.

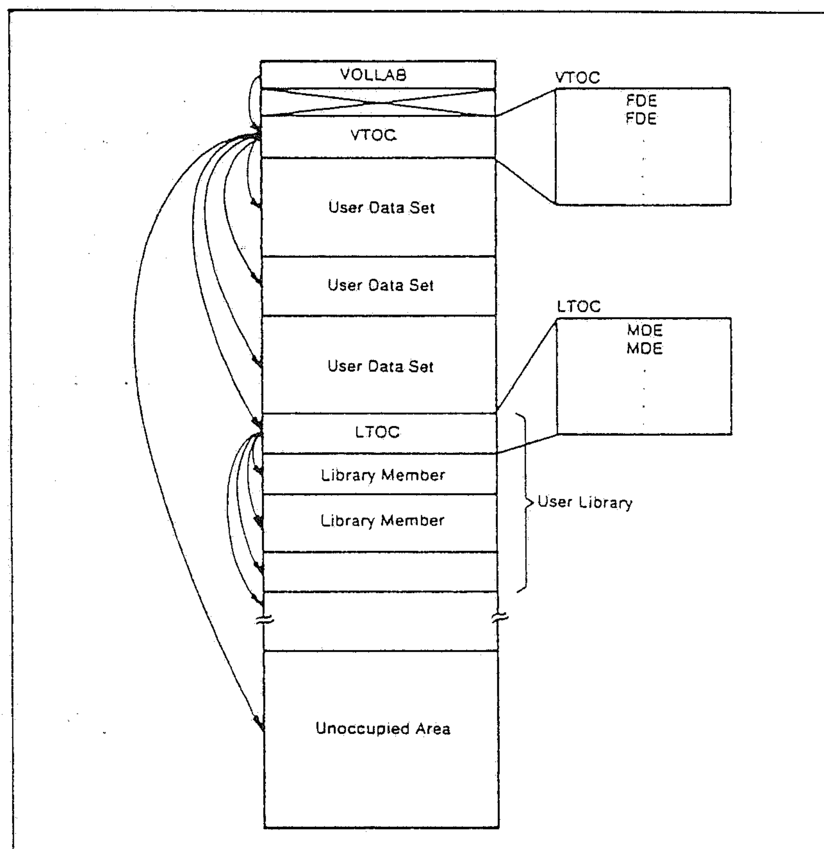


Figure 4.7 Example of a data diskette organization

4.4.3 System Diskette

A system diskette is created by using the Allocate function in Console Mode.

The system diskette contains the operating system and usually an IBM or UNIVAC emulation. Other system programs can also be stored on the system diskette.

Before any data sets have been created, the diskette is marked as empty. The diskette VTOC contains a pointer to the unoccupied area at the end of the volume, and a number of unused data set directory entries (FDEs) . Successively, as the diskette is provided with system data sets, user data sets and user libraries, the VTOC is updated.

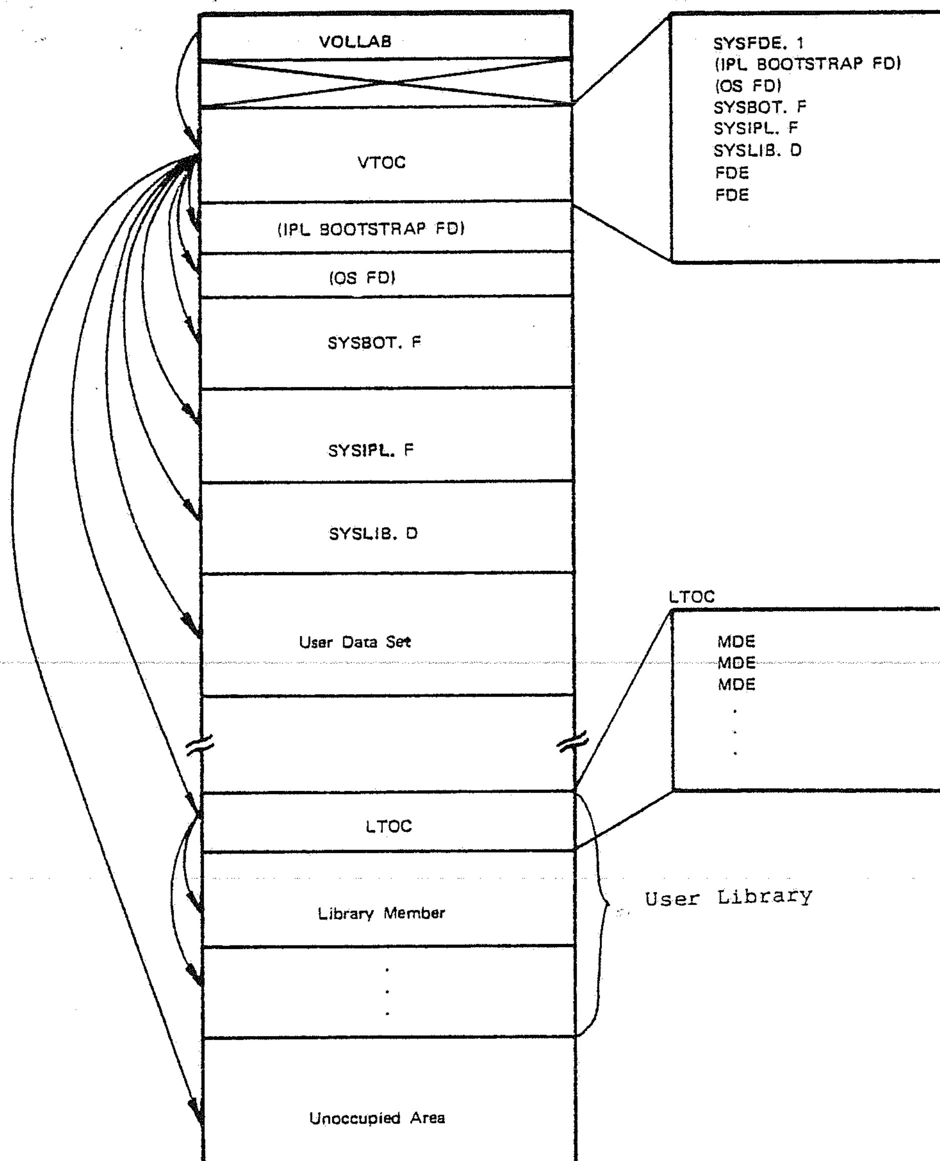


Figure 4.8 Example of System diskette organization

4.4.4 PC Diskettes

The PC diskettes contain bootstrap data for the personal computer system.

There are two types of PC diskettes. They are denoted type A and type P.

A PC diskette of type A is loaded into the FD at reset or power on. A PC diskette of type P is loaded only on request from the display unit on the V.24 connection.

The S41 operating system cannot handle other information on the PC diskette than the volume label (VOLLAB). No VTOC is provided on a PC diskette.

The volume label of a PC diskette is extended by information to the PC loader, which is part of the S41 operating system. See section on Volume Label below.

4.5 VOLUME LABEL (VOLLAB)

All diskettes in System 41 contain a volume label (VOLLAB). VOLLAB is stored in track 0, sector 0, and contains the following information concerning the entire diskette volume.

Byte (dec)	Length (bytes)	Name	Comments
0	1	DTYPE	Diskette type: D Data diskette E Empty diskette S System diskette A Personal Computer diskette type A P Personal Computer diskette type P
1	8	DVOLNR	The volume number is a unique number within a system, i.e. this number must only be used for one diskette.
9	41	DUINFO	User information, specified below
9	8	DNAME	Volume name
17	2	DVERS	Volume version
19	10	DREVDT	Revision date
29	20	DUSER	User name
49	1	DFLAG	≡ 0 (0000 0000)
50	2	DSTAT	Volume/system status
51			LLSSSSSS, Bits 7-6 (LL) specify sector length expressed as a number of bytes. Bits 5-0 (SSSSSS) specify the number (n) of sectors per track. LL = 00 128, n = 26 (8" standard) LL = 01 256, n = 15 LL = 10 512, n = 8 LL = 11 1024, n = 4 Standard value of DSTAT is 1A (hex) for 8" diskettes, and 89 (hex) for 5"1/4 diskettes.
52	2	DVPTR	Address of VTOC
52			Track address of VTOC
53			Sector address of VTOC
54	2	DVSIZE	Size of VTOC expressed in number of file directory entries (FDEs)
56	8	DFDBOT	FD bootstrap file name
64	24	DNAT	National version
88	10	DRPQ	RPQ made at a Ericsson subsidiary

The following information is provided only on PC diskettes

Byte (dec)	Length (bytes)	Name	Comments
98	1	DHEAD	Side address of PC bootstrap
99	1	DCYL	Cylinder (track) address of PC bootstrap
100	1	DSECT	Sector address of PC bootstrap
101	2	DNUMB	Number of sectors used for PC bootstrap
103	2	DLSEGM	Segment address of load point
105	2	DLOFFS	Offset for load point
107	2	DJSEGM	Segment address of PC-boot program start
109	2	DJOFFS	Offset for program start

```
          **********  C O N S O L E   M O D E   **********
        D I S P L A Y  / C H A N G E   V O L U M E   L A B E L

GENERAL VOLUME INFORMATION
TYPE           S                        NUMBER      016101>00<
VERSION        00                       NAME        >SYSTMAED<
DATE           831007                   NATION      >SE/FI       <
USER           M305-00                  RPQ         >      <->   <
SECTORSIZE     128
VTOCSIZE       032

SYSTEM VOLUME INFORMATION
PRODNAME       ALFASKOP S41 IBM3274/78BSC,CLUSTER


OSVERSION      OS M305-01: CP M305-01  DU M305-01  PU -------
               FD M305-01  SCC -------



EXECUTE     UNDO      RETURN
ENTER       PF1       PF12
```

Figure 4.9 The terminal console function Display/Change Volume Label applied to a system diskette.

4.6 VOLUME TABLE OF CONTENTS (VTOC)

Each data set on a diskette is described in a data set directory entry (FDE). The FDEs, taken together, constitute another data set called VTOC (Volume Table of Contents).

Each FDE contains information about:

- o Data set type, DSTYPE
- o Data set name, DSNAME
- o Access authorization level
- o Physical start address of data set
- o Data set length
- o Block and record sizes
- o Last block physical address, degree of utilization
- o Internal information

A VTOC containing a maximum of 32 FDEs can be stored in primary storage, thus providing fast diskette access. One of the FDEs is always used to point to the unoccupied area on the diskette.

There are two types of FDEs in VTOC:

- o General FDE, pointing to user data sets
- o Special FDE, pointing to user libraries and to the unoccupied area at the end of the volume

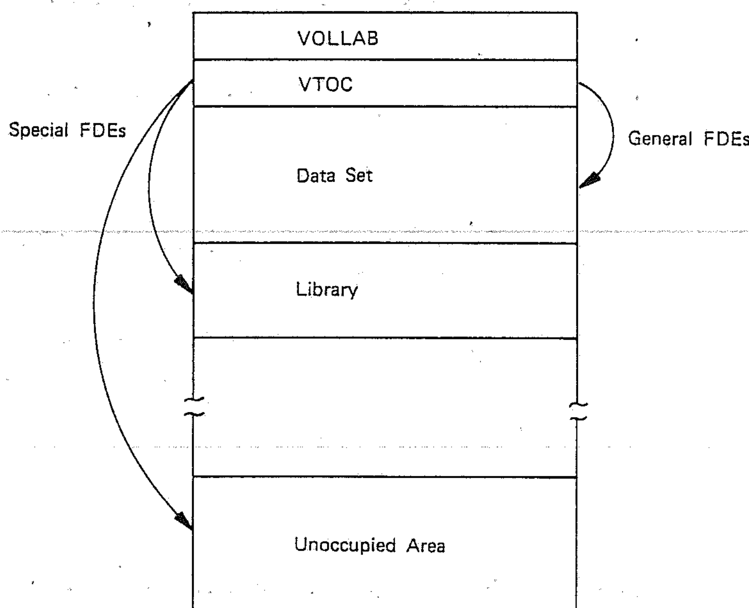


Figure 4.10 Data set directory entried (FDEs) in VTOC

4.6.1 General FDE Format

General FDEs are used in VTOC to point to user data sets, and in LTOC to point to member data sets. The following information is found in the general FDEs:

Byte (dec)	Length (bytes)	Name	Comments
0	1	DSTYPE	Data set type A = Absolute code (unpacked data) F = Fixed-length record data (unpacked data) R = Relocatable file T = Table data set (packed data) V = Variable-length record data set (packed data) 0 = Undefined data set 7 = Released data set
1	8	DSNAME	Data set name
9	2	DSSTAT	Status
9		DSAUTH	Access authorization level
10		DSPROT	Protection code (to be defined)
11	1	DSFBSZ	Block size, expressed as an integral number of sectors.
12	2	DS1STA	Data set size, expressed in sectors First allocation value
14	1	DSDOVH	Device overhead (bits 7-0, LLSSSSSS). Bits 7-6 (LL) specify sector length in bytes. Bits 5-0 (SSSSSS) specify number (n) of sectors per track. LL = 00 128, n = 26 (8"standard) LL = 01 256, n = 15 LL = 10 512, n = 8 LL = 11 1024, n = 4 Standard value of DSDOVH is 1A (hex) for 8" diskettes, and 89 (hex) for 5"1/4 diskettes.
15	1	DSFUUS	Not assigned
16		DSPADR	Address of first sector in the data set
16		DSPCYL	Track number
17	1	DSPREC	Head (side) and record (sector) number HRRRRRRR. Bit 7 (H) head. Bits 6-0 (RRRRRRR) record number.
18	2	DSSIZE	Data set size in blocks (either DSFBSZ or DSBSIZ).
20	2	DSBSIZ	Block size in bytes (logical blocks).
22	2	DSRSIZ	Record size in bytes (logical records).
24	2	DSFNUB	Number of blocks used. (This is the "logical end of file")
26	2	DSLUSE	Number of bytes used in last block
28	2	DSENPT	User-defined (for SYSBOT.A, this is a pointer to an entry point in a loaded program).
30	2	DSLDPT	User-defined. Used by system for LOAD/IPL command execution (for SYSBOT.A, this is a pointer to the start address used for loading)

4.6.2 Special FDE Format

Special data set directory entries are used in VTOC to point to the unoccupied area at the end of the volume, and to user libraries. The following information is found in the special FDEs.

Byte (dec)	Length (bytes)	Name	Comments
0	1	MSTYPE	Data set type D = Directory 1 = Unoccupied area (FDE contains volume information)
1	8	MSNAME	Data set name
9	2	MSSTAT	Status
9		MSAUTH	Access authorizational level
10		MSPROT	Protection code (to be defined)
11	1	MSFBSZ	SNNNNNNN (bits 7-0). If bit 7 (S) is 1, it indicates that this is a split track directory. Bits 6-0 (NNNNNNN) specify the number of data set directory entries per sector.
12	2	MS1STA	Data set size in sectors. First allocation value.
14	1	MSDOVH	Device overhead (bits 7-0), LLSSSSSS. Bits 7-6 (LL) specify sector length in bytes. Bits 5-0 (SSSSSS) specify number (n) of sectors per track. LL = 00 128, n = 26 (8" standard) LL = 01 256, n = 15 LL = 10 512, n = 8 LL = 11 1024, n = 4 Standard value of MSDOVH is 1A (hex) for 8" diskettes, and 89 (hex) for 5"1/4 diskettes.
15	1	MSFUUS	Number of sectors in VTOC/LTOC
16	2	MSPADR	Address of first sector in the data set
16	1	MSPCYL	Track number
17	1	MSPREC	Head (side) and record (sector) number HRRRRRRR. Bit 7 (H) head. Bits 6-0 (RRRRRRR) record number.
18	2	MSSIZE	Number of unoccupied sectors
20	2	MSBSIZ	(Default) block size
22	2	MSRSIZ	VTOC/LTGC size (in number of FDEs/MDEs)
22		MSACT	Number of active FDEs/MDEs
23		MSORG	Total number of FDEs/MDEs
24	2	MSFNUB	
24		MSTP0	Number of type 0 (unused) FDEs/MDEs
25		MSTP7	Number of type 7 (released) FDEs/MDEs
26	2	MSLUSE	CHR-address of first unoccupied sector
28	2	MSENPT	
28		MSDIRB	Number of directory blocks
29		MSASEC	Number of sectors in last directory block
30	2	MSLDPT	CHR-address of last sector in this extent

********** C O N S O L E M O D E **********
D I S P L A Y V T O C

UNUSED AREA 0049 SECTORS STARTS AT CYL 4A SECTOR 05 UNUSED FDE 0C

TYPE	NAME	AUTH	FBSZ	1STA	CYL	SEC	SIZE	BSIZE	RSIZE	FNUB	LUSE	ENPT	LDPT
F	PRODNAME	04	01	0001	00	0A	0001	0080	0080	0001	0080	1ACE	17F9
A	FDBOOT	05	01	0003	00	0B	0003	0080	0080	0003	005E	6A04	6A04
F	SYSBOT	05	01	0010	00	0E	0010	0080	0080	0010	0080	8765	4300
F	LOGON	04	07	0007	01	04	0001	0380	0380	0001	0380	0000	0000
F	SYSIPL	05	01	0022	01	0B	0022	0080	0080	0022	0080	0000	0000
F	IPLDUMP	03	08	0038	02	13	0007	0400	0400	0007	0400	0000	0000
D	APLLIB	00		0012	04	17							
D	CHLIB	05		0061	05	0F							
D	DCLIB	00		0007	09	08							
D	KBLIB	00		0016	09	0F							
D	KBLIBA	00		0016	0A	0B							
D	LILIB	00		0011	0B	07							
D	MLLIB	00		0011	0B	18							
D	PRLIB	00		000C	0C	0F							
D	EMLIB1	00		00F8	0D	01							

NEXT PAGE PREV PAGE RETURN
PF1 PF2 PF12

PAGE 1

Figure 4.11 The terminal console function Display VTOC applied to a system volume.

4.6.3 SYSFDE Types

SYSFDE.0

System data set directory entry of type 0 points to an unused data set in the volume.

SYSFDE.1

System data set directory entry of type 1 points to the unoccupied space at the end of the volume, and contains information on the entire volume.

SYSFDE.7

System data set directory entry of type 7 points to a released data set in the volume.

Compare with DSTYPE byte in general FDE and MSTYPE in special FDE, as described above.

4.7 LIBRARY TABLE OF CONTENTS (LTOC)

The members of a library are described by descriptive elements called Member Directory Entries (MDE). A collection of MDEs is called an LTOC (Library Table Of Contents).

Each MDE contains the same type of information as an FDE, namely:

- o Member type, MSTYPE
- o Member name, MSNAME
- o Access authorization level
- o Physical address
- o Member length
- o Block and record sizes
- o Last block physical address, degree of utilization
- o Certain other internal I/O system information

The table elements in LTOC resemble the table elements in VTOC.

********** C O N S O L E M O D E **********
D I S P L A Y L T O C LIBRARY NAME>SYSLIB <
UNUSED AREA 0000 SECTORS STARTS AT CYL 2D SECTOR 0A UNUSED MDE 00

TYPE	NAME	AUTH	FBSZ	1STA	CYL	SEC	SIZE	BSIZE	RSIZE	FNUB	LUSE	ENPT	LDPT
A	BIG00020	05	02	000C	16	12	0006	0100	0100	0006	00E3	3508	3508
A	BCC50200	03	02	0002	17	04	0001	0100	0100	0001	00B0	0B50	0B50
A	BCC50300	03	02	0002	17	06	0001	0100	0100	0001	00F0	0C00	0C00
F	OSVERS	03	02	0002	17	08	0001	0100	0048	0001	0048	0000	0000
A	BXFA031	00	02	00AC	17	0A	0056	0100	0100	0056	00A1	5E77	13B3
A	BXCA031	05	02	0076	1E	00	003B	0100	0100	003B	0009	0B00	0B00
A	LOGONPRG	05	02	004C	22	0E	0026	0100	0100	0026	0036	4766	4766
A	BXDA031	05	02	0084	25	0C	0042	0100	0100	0042	0100	0600	0600
A	PROPTION	05	01	0005	2A	0E	0005	0080	0080	0005	0072	9A0E	9A0E
A	PRDEF	03	02	000C	2A	13	0006	0100	0100	0006	0007	9A07	9A07
A	PUPROGRM	05	02	002E	2B	05	0017	0100	0100	0017	00F9	830E	830E
A	PRSTRAPS	05	01	000B	2C	19	000B	0080	0080	000B	005F	9F0E	9F0E

END OF LTOC
EXECUTE NEXT PAGE PREV PAGE RETURN
ENTER PF1 PF2 PF12 PAGE 1

Figure 4.12 The terminal console function Display LTOC used on the library SYSLIB on a system diskette.

4.8 DATA SET TYPES

The following types of data sets are defined in Alfaskop System 41:

D Library file (D stands for directory).
Specifies a library in the VTOC entry. File type D must not be used in an LTOC element. If the volume is accessed as file type D, VTOC will be accessed as a sequential file.

F Fixed-length record file (contains unpacked data).

A Absolute file (unpacked form) generated by the linkage editor when linking object modules from files of type R. Files of type A require four bytes of user information in the VTOC entry to store the entry point and load point addresses. Absolute files are in absolute code.

V Variable-length record file containing packed data, i.e. series of identical characters are replaced by two bytes. First byte specifies the number of characters in the series. Second byte specifies the character that constitutes the series. Source code is kept in this type of file.

R Relocatable file (contains relocatable packed data and is used by the SPL compiler). This type of file is in object code.

T Table file (packed form) generated by the linkage editor to provide relocation information.

0 Unused data set
1 Unoccupied area
7 Released data set

A short notation is often used in this manual to indicate the data set type. For example, SYSLIB.D indicates that the data set named SYSLIB is of type D, i.e. SYSLIB is a library.

Note that the file handling system of Alfaskop System 41 only supports handling of files of types D, F, A and V. A logical file can be associated with a simple data set, a library or an entire volume. See section on FD Unit Functions and File Handling.

Contents

5.1	INITIALIZATION	3
5.1.1	IPL (Initial Program Loading)	3
5.1.1	NIP (Nucleus Initialization Procedure)	7
5.2	LOGON	9
5.2.1	Manual Logon	10
5.2.2	LOGON File	12
5.2.3	MENUFILE.F	13
5.2.4	Menu Logon	13
5.2.5	Message Line Logon	13
5.2.6	Autologon	13
5.3	PROGRAM LOAD	15
5.3.1	Single Module Load	15
5.3.2	Load Map Load	15
5.4	LOGOFF	16
5.5	SYSTEM DATA SETS	17
5.5.1	FD Load Modules	17
5.5.2	SYSBOT.F	17
5.5.3	SYSIPL.F	18
5.5.4	SYSLIB.D	18
5.6	AUTHORIZATION LEVELS	19-19

5.1 INITIALIZATION

When power is turned on to a system unit the system software must be loaded from the system diskette.

The initialization procedure can be divided into three phases:

- IPL (Initial Program Load) which is used to load the operating system from the system diskette.
- NIP (Nucleus Initialization Procedure) which initializes the operating system.
- Loading of the system modules and application modules. This is performed in association with the logon procedure.

The three phases are all described in the following.

5.2 IPL (Initial Program Loading)

An IPL modules is stored in non-volatile memory (a 2 K byte PROM) so that program loading can be carried out after a power failure in a System 41 unit.

The IPL procedure is performed when power is turned on or when the RESET pushbutton is depressed. The two events initiate slightly differing IPL procedures.

When power is turned on, a test procedure is executed prior to program loading. This test procedure carriers out the following:

- RWM test conducted by writing and reding the first 32 k RWM cells.
- PROM test conducted by carrying out a CRC-16 (Cyclic Redundancy Check) calculation of the PROM content and comparing the result with a sum stored in the PROM.

When the RESET pushbutton is depressed, no test procedures are carried out. Instead, a dump function is performed.

In addition to the test and dump procedures, the IPL PROM contains:

- A routine used for initializing the peripheral circuits
- A limited Communication Handler sufficient to handle initial two-wire communication
- A simple Timer Handler that provides time supervision of the initial communication
- SPL subroutines

The IPL procedure is explained by means of flowcharts showing the most important steps (See figures below). During the loading procedure of a DU unit, messages are presented on the bottom line of the screen. These messages also appear in the flowchart.

The IPL command isued to FD contains, as a parameter, the physical unit number. The unit number is used as an index to the sequential SYSBOT.F file. The index identifies the absolute module that is to be loaded from the library SYSLIB.

Information is also obtained from SYSLIB about the load point (the address at which the module is to be stored) and the entry point (the address at which the execution of the module is to start after loading).

Note. In display units which load the character generator from diskette, the various IPL phases are indicated by filled rectangles on the message line. See document on Maintenance.

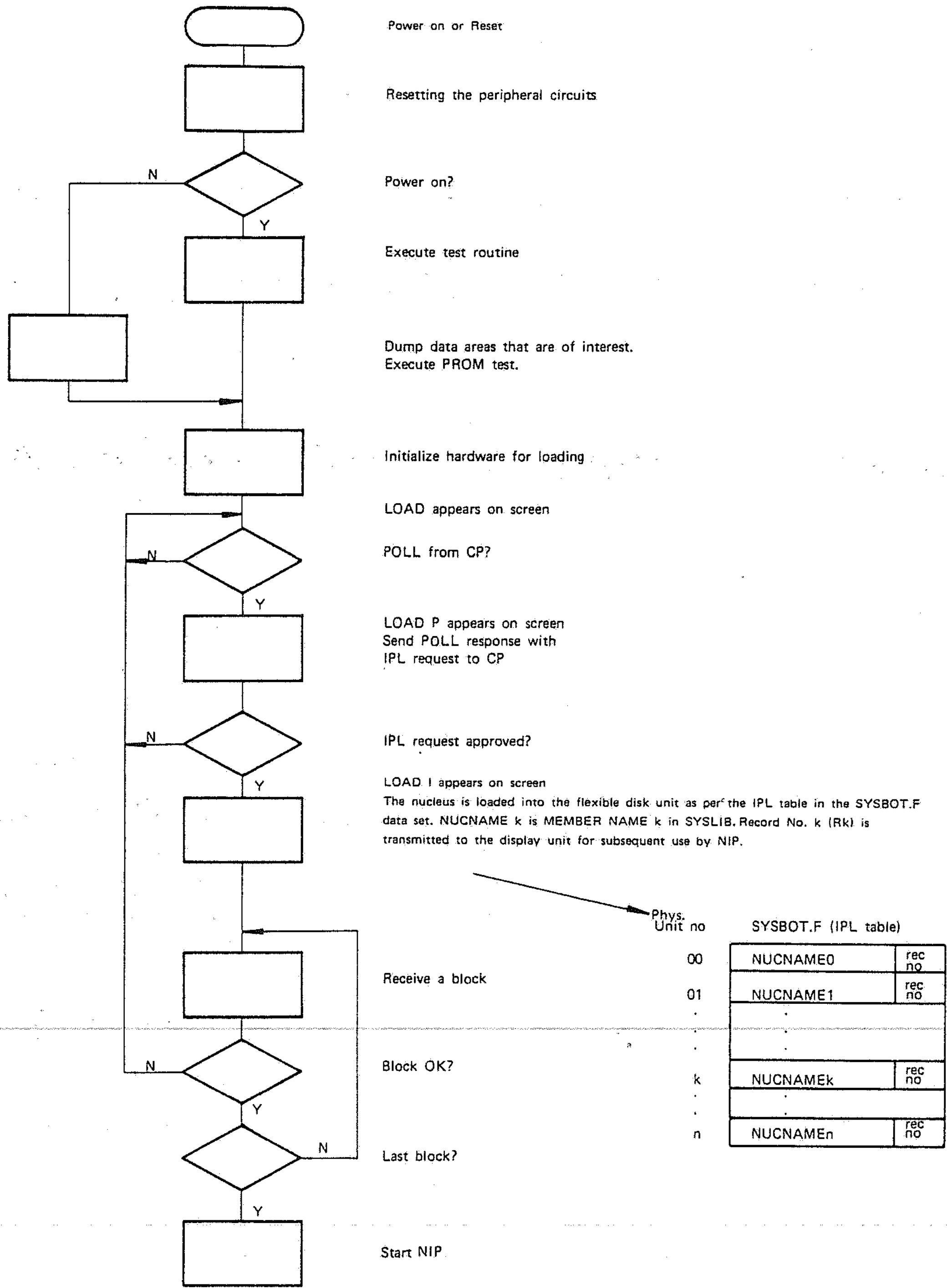


Fig. 5.1 IPL flowchart for DU

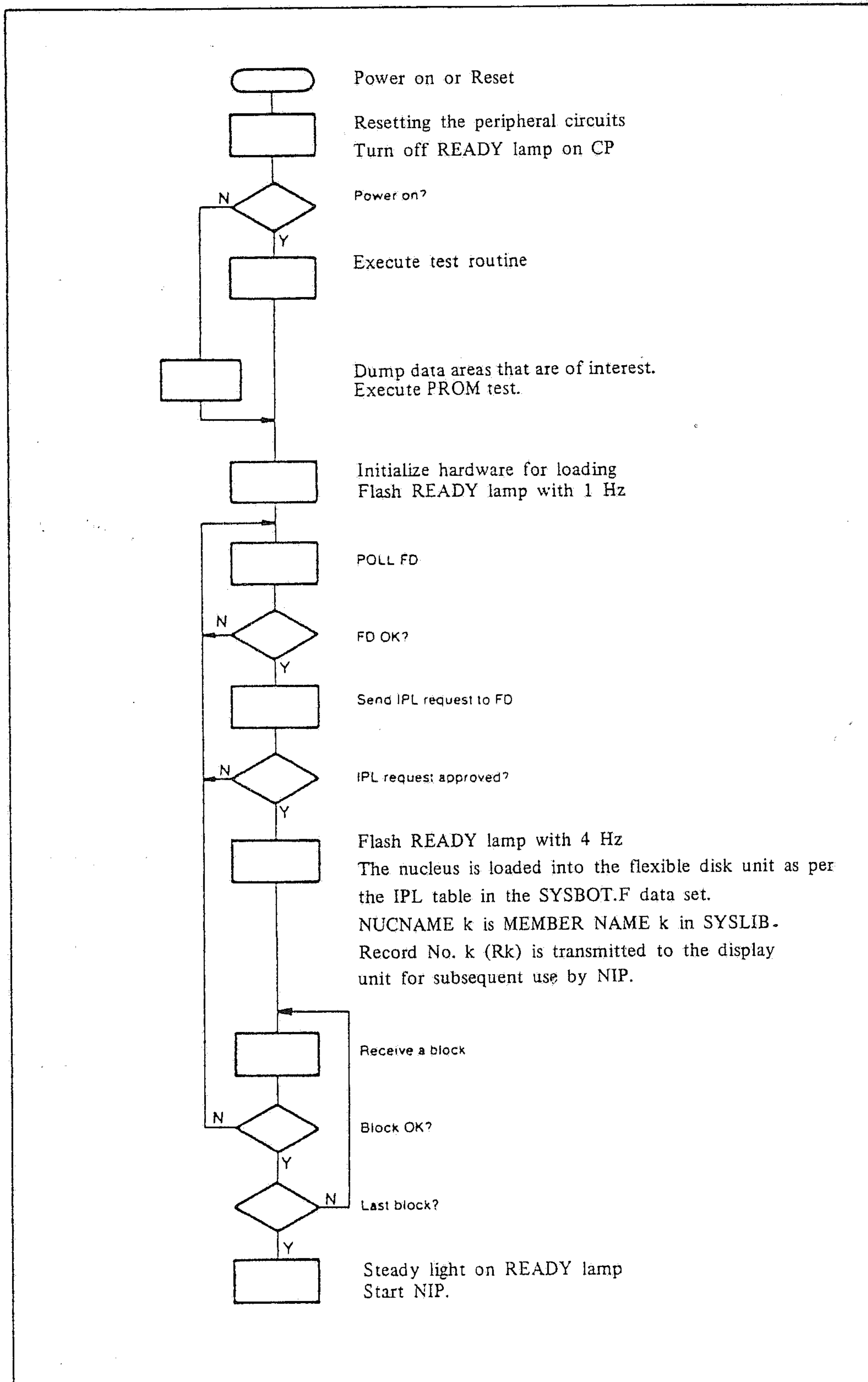


Fig. 5.2 IPL flowchart for CP

5.1.2 NIP Nucleus Initialization Procedure

To start up the operating system after initial program loading (IPL), there is a procedure called NIP which initializes and activates the different parts of the operating system.

NIP is divided into two parts, one of which is resident and one of which is an overlay segment. The overlay segment contains modules which initialize the remainder of OS and write (on flexible disk) the dumps saved in connection with RESET.

The NIP procedure is presented by the simplified flowcharts below:

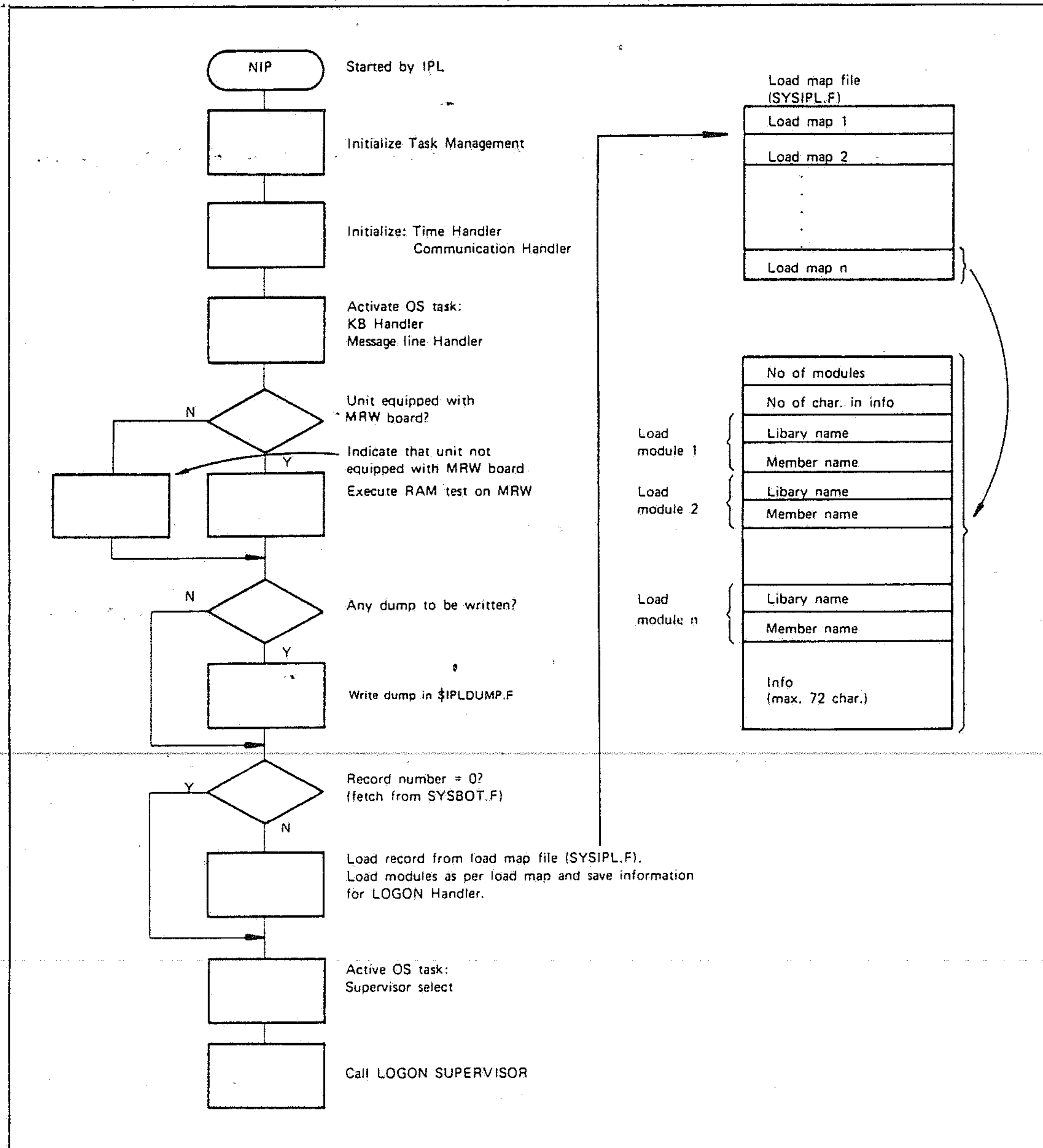


Fig. 5.3 NIP flowchart for DU

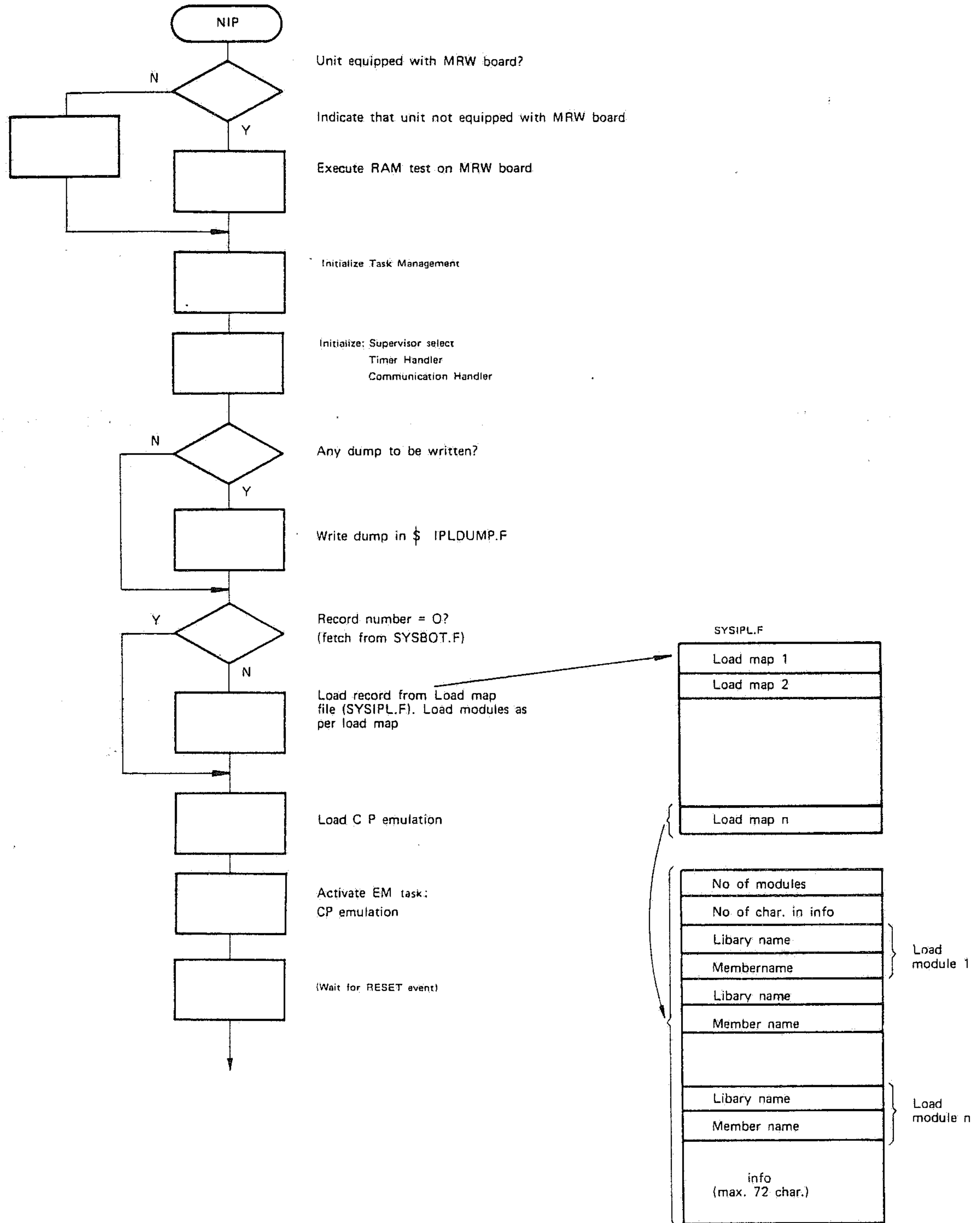


Fig. 5.4 NIP flowchart for CP

5.2 LOGON

Logon is the procedure in which the desired emulation and/or application program is loaded from diskette and started up.

Logon is carried out by the Logon Supervisor and the Logon Handler.

The logon procedure can be performed in various ways:

- o Manually by the operator using the manual logon form.
- o By the operator selecting the desired logon item from a logon menu.
- o By the operator entering the desired module code on message line when an emulation is loaded.
- o Automatically by means of Autologon.

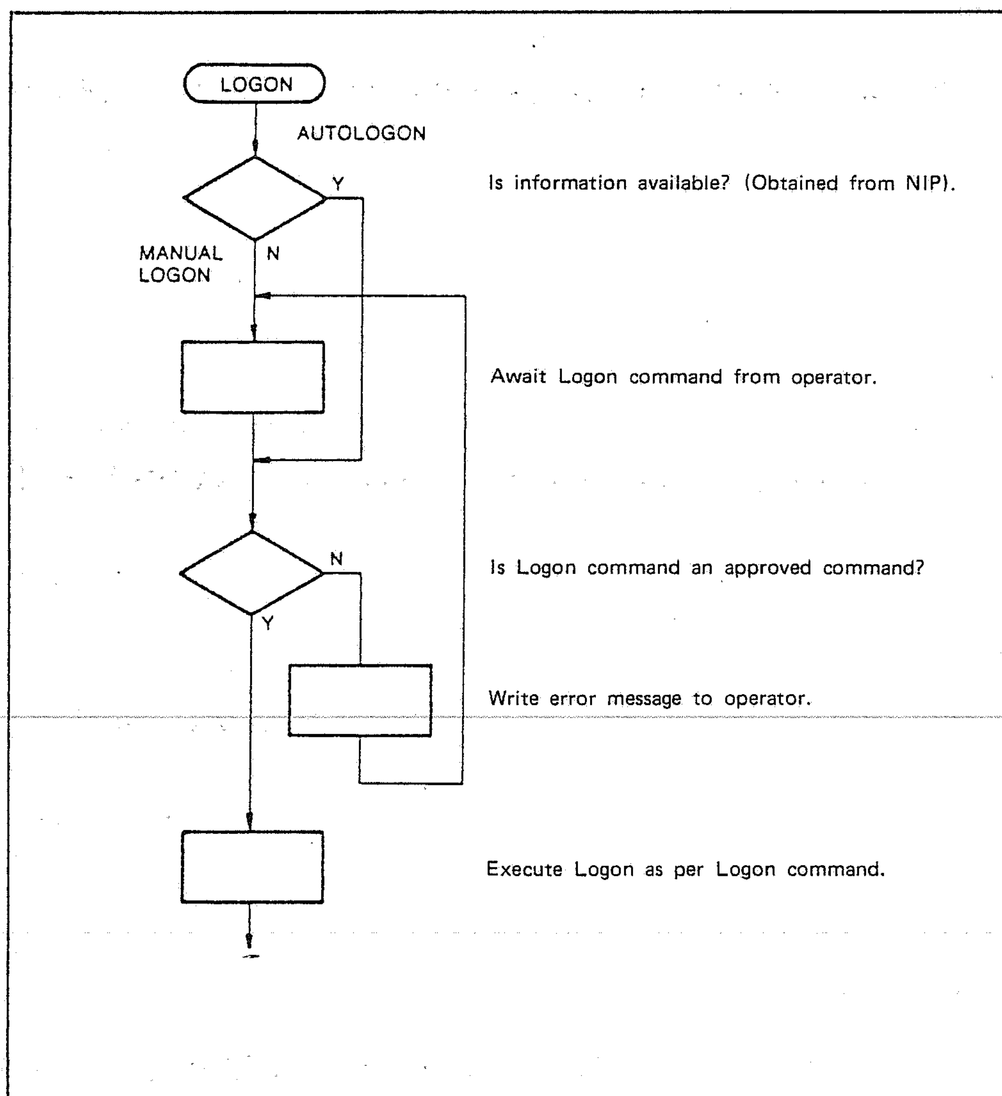


Fig. 5.5 Simplified flowchart for Logon

5.2.1 Manual Logon

The following parameters appear on the manual logon form:

VOLUME NAME
LOGON/FILE NAME
LOAD MAP NO
PASSWORD 1
PASSWORD 2
CONTROL INFO

VOLUME NAME is the name of diskette on which the desired logon item (data set) is stored.

LOGON/FILE NAME is associated with the name of the desired data set via the LOGON file. Several logon/file names can be associated with the same data set. For example, a data set can be identified by its own name in upper and lower case letters, and by one abbreviation.

LOAD MAP NO is the number of the record in which the load map is located. If no load map is specified, the unit will be loaded with a default load map from the LOGON file. See section on Program Load below.

PASSWORDS defines authorization. System 41 provides six different authorization classes denoted 0-5. See section on Authorization below.

CONTROL INFO is information that is turned over to the system module after LOGON. Information entered into this field is available to the application in a globally declared area called BUFFEREM. This area can be declared by the user as follows:

```
    DECLARE 1    BUFFEREM EXTERNAL,
                 2 * CHAR(6),
                 2 MESSEM CHAR(75);
```

Control information can also be found in the load map. If such is the case, information in the load map is overridden by the control information entered by the operator.

Figure 5.6 presents a simplified flowchart showing how manual logon is carried out.

When the LOGON/FILE NAME identifier has been translated to the system module name, the system module is loaded into DU. In the simplified flowchart, this module is named SYSMODK. If a load map load is to be carried out, there will be a fixed-length record file having the same name as the system module but with .F appended. In the flowchart, it is thus designated SYSMODK.F.

The load map format is the same as shown in figures for IPL above.

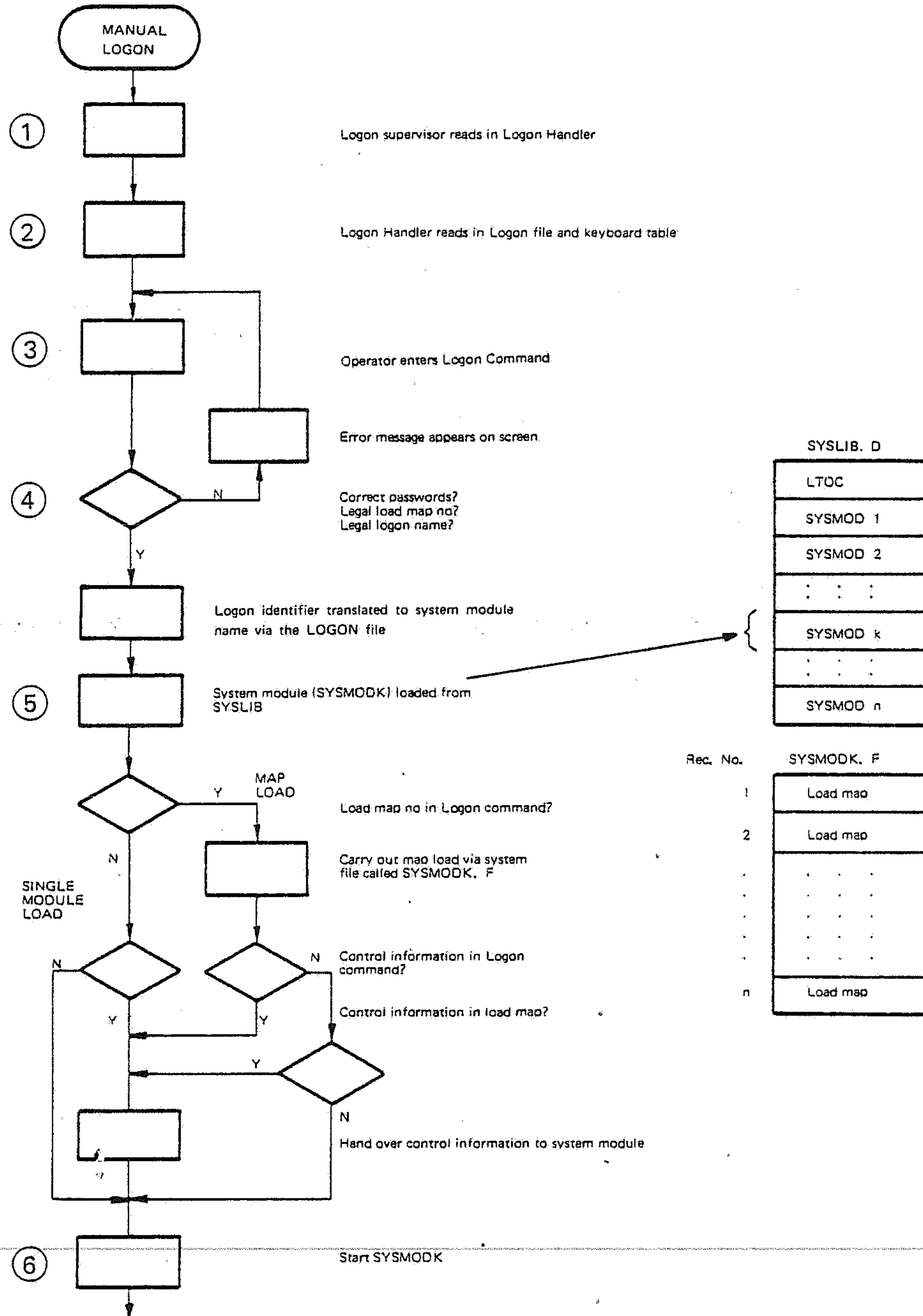


Fig. 5.5 Manual Logon and program Load

Manual Logon, functional steps:

1. Logon Supervisor loads Logon Handler
2. Logon Handler reads in LOGON file and Keyboard Table
3. Operator fills in the manual logon form and depresses ENTER
4. Logon Handler approves Logon command
5. Logon Supervisor loads the selected module (overlay of Logon Handler sometimes occurs)
6. Loaded module is executed.

5.2.2 LOGON File

A LOGON file of type F is provided on the system diskette (and also on some data diskettes containing programs). The following information is stored in the LOGON file:

- A table for conversion from the LOGON/FILE NAME entry to the appropriate data set. If the entered logon identification is a name of a data set, no conversion is performed. This is also the case if a member name is entered, and the name of the library is the same as the name of the member.

- A table for association of LOGON/FILE NAME to communication processor. (Used only in Dual Host configurations.)

- The default keyboard table for all DUs.

- The default load map number for all DUs in the cluster. This load map is used if no load map number is used at logon.

5.2.3 MENUFILE.F

To obtain a simplified logon procedure, a MENUFILE is provided on all system diskettes of version 3.5 and later.

MENUFILE.F can be defined as desired by the user. The file contains 18 records, containing 54 bytes each. The first record contains the menu header and the input prompter. The remaining records are to be defined by the user. (This is performed in Console Mode).

Each record specifies one selectable logon item. Only modules that don't required any password can be specified on the menu.

Each module on the menu is associated with a code consisting of up to three characters. The identificaton code specifies the following items for the module:

VOLUME NAME
LOGON/FILE NAME
LOAD MAP NO

One of the specified codes must refer to the manual logon form, thereby making it possible to load modules which require password authorization.

5.2.4 Menu Logon

When MENUFILE.F is provided, the menu is presented instead of the manual logon form when power is turned on to the DU.

Each selectable logon item is associated with a identification code. The desired code is entered in the input field of the menu, and the predefined combination of logon parameters are handed over to the Logon Handler.

5.2.5 Message Line Logon

When MENUFILE.F is provided, the desired code can also be entered directly on the *OS* message line when an emulation is loaded.

5.2.6 Autologon

Autologon is specified during customizing of the system. The logon name and the load map No. are entered in the autologon form in Console Mode.

The load map contains the name of the absolute module that is to be loaded when power is turned on or the unit is reset. The load map also contains control information analogous to the CONTROL INFO entered by the operator during manual logon. The control information field contains 58 bytes.

The Logon Handler evaluates the control information handed over by NIP in the same way as if a logon command had been received in connection with manual logon. The procedure is then the same as carried out for manual logon. See section 5.2.1.

Autologon, functional steps:

1. Logon supervisor loads Logon Handler and Extende FDIOS

2. Logon Handler reads in LOGON file and Keyboard Table

3. Logon Handler evaluates information obtained from load map received by NIP

4. Logon Handler approves Logon command

5. Logon Supervisor loads specified module (overlay of Logon Handler sometimes occurs)

6. Module that was loaded is executed.

5.3 PROGRAM LOAD

As mentioned in secion on Logon, program loading can be carried out in two ways: as a single module load or as a load map load. The load map load is used only by NIP and during logon.

5.3.1 Single Module Load

The single module load can be carried out from a data set or from a member of a library. Only absolute modules can be loaded from other files. Program loading can comprise one or more blocks.

The entry point and the load point are stored in the data set description element (FDE) on flexible disk, and they are transmitted together with each block (load points are incremented).

The entry point and the load point are defined in connection with the creation of the module, but can also be modified via a change command set to FD.

5.3.2 Load Map Load

The load map provides loading of a predetermined combination of modules into the display unit and/or the PCU. First, a basic module is loaded by means of a single module load. Then additional modules, specified in the load map, are loaded.

The load map function is activated by means of an index which points to a record in the fixed-length record file that contains the load map. This index is obtained from the SYSBOT.F file during IPL. The record number is then handed over to NIP so that a load map load can be carried out.

No load map load is carried out if the record number is 0.

For Logon, the record number is obtained from the LOAD MAP NO. parameter in the Logon command. The basic module is obtained by translating the LOGON/FILE NAME in the command to a system module name via the LOGON file.

The fixed-length record file has the same name as the basic module member, althought .F has been appended. If the record numnber is 0, a default record number, defined at system generation time, is used. See sectionon Logon.

The IPL figures present the load map format.

5.4 LOGOFF

To some extent, Logoff is handled by the system module which was attached as a task by the Logon Supervisor at Logon time. When this task is terminated, logoff occurs.

If a program product is to be logged off, the operator follows the instructions given in the program.

If an emulation is to be logged off, the operator only has to enter LOGOFF on the *OS* message line. (In fact any code that does not correspond to a logon item can be entered in order to logoff the emulation if a menu file is provided.)

The logoff command causes the logoff event to be posted, indicating to the system module that a logoff command has been executed. The module must then immediately conclude its work and pass control back to the Logon Supervisor.

It is the responsibility of the OS user to see that the system module subtasks are terminated properly and that they do not wait for any events. The global byte BID_$0245 indicates whether the printer editor is processing. See section 12.9.4.

When logoff has been carried out, the Logon Supervisor loads the Logon Handler into DU. The Logon Handler then presents the logon menu and waits for an operator entry.

5.5 SYSTEM DATA SETS

5.5.1 FD Load Modules

The FD load modules are loaded when initial program loading takes place for the FD unit. At IPL time the IPL Bootstrap FD module is loaded by the IPL PROM. Its file name is defined in the DFDBOT entry in VOLLAB. Thereafter the IPL Bootstrap FD module loads the FD operating system. Its file name is defined in the software of the IPL Bootstrap FD module. In FD 4122, the FD operating system is immediately loaded.

The system diskette contains a number of system data sets. The data sets presented below are essential to the loading and initializing of the operating system.

5.5.2 SYSBOT.F

SYSBOT.F is a table used to translate physical unit addresses to data set names. The desired data set is loaded into the unit during the initial program load phase (IPL).

The table comprises a list of absolute member names arranged in physical address sequence. The absolute members are collected in the library SYSLIB.

A record number is stored together with each member's name in the SYSBOT file. The record number indicates in which record of the SYSIPL file supplementary information can be found.

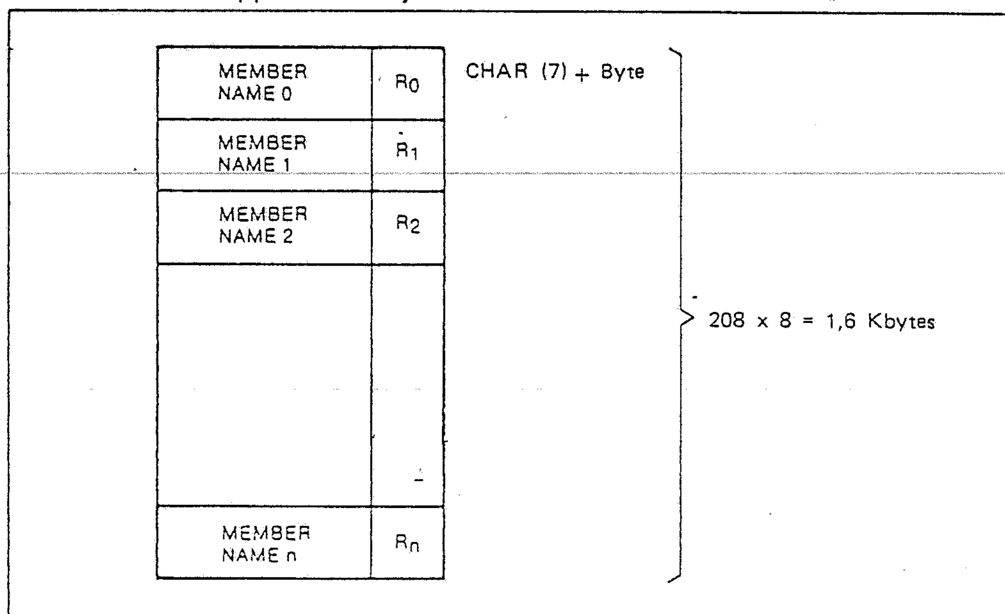


Figure 5.7 SYSBOT table with member names and record numbers.

5.5.3 SYSIPL.F

SYSIPL is a data set which contains supplementary information used for initial program loading of the system units.

5.5.4 SYSLIB.D

SYSLIB is a library which contains, among other things, all absolute members that are used for initial program loading (IPL) of the system units.

5.6 AUTHORIZATION LEVELS

Alfaskop System 41 provides for six different authorization classes, denoted 0-5. The authorization classes are introduced to prevent unauthorized changing or deleting of program modules.

Each system module is assigned one of the six authorization classes.

Two passwords are associated with the authorization classes. The passwords are to be entered by the operator on the manual logon form.

Modules with authorization 0 do not require any password.

Modules with authorization 1-3 require at least password No 1.

Modules with authorization 4-5 require both password No 1 and password No 2.

Contents

6.1	GENERAL	3
6.2	EVENT CONTROL	3
6.2.1	Event Declaration	3
6.2.2	Event Control Block	4
6.3	MULTITASKING FUNCTIONS	5
6.3.1	Task Declaration	5
6.3.2	Task Control Block (TCB)	5
6.3.3	Attach Task	8
6.3.4	WAIT Event	9
6.3.5	POST Event	9
6.3.6	ASSIGN Event	9
6.3.7	WAIT Task	10
6.3.8	Task Termination	10
6.3.9	CANCEL Event	10
6.3.10	CANCEL Task	10
6.3.11	EXIT	11
6.4	INTERRUPT HANDLING	11
6.4.1	Reset Interrupts	12
6.4.2	Software Interrupts	12
6.4.3	Interrupts from Peripheral Circuits	12
6.4.4	Interrupt Levels	13
6.4.5	Interrupt Procedures	13
6.4.6	Interrupt Vectors	15
6.4.7	Examples of Interrupt Handling	17-21

6.1 GENERAL

This section is a functional description of the OS module Task Manager. The Task Manager processes the muiltitasking statements in an SPL program.

A general description of the multitasking concept is presented in Section 1.

The SPL commands are described in detail in the SPL Reference Manual.

The hardware circuits used for Interrupt handling are described in the Technical Description.

6.2 EVENT CONTROL

The "event" is the SPL concept for temporary synchronization of asynchronous processes.

The event can be regarded as a global flag, whose current status can be checked by any task in the system.

The event status is contained in a control block in main storage. The following control blocks can be used for events:

- Task Control Block (TCB)
  An event occurs when the task terminates or is terminated by a superior task. The task control block is presented in detail in section on Multitasking Functions below.

- Volume Control Block (VCB)
  An event occurs when a volume command is executed by FDIOS. See section on FD Functions.

- File Control Block (FCB)
  An event occurs when a file command is executed by FDIOS. See section on FD Functions.

- Event Control Block (ECB)
  The event control block is used for events which are explicitly declared as SPL variables of type EVENT. An event occurs when the event is posted. The event control block is further discussed below.

6.2.1 Event Declaration

The event variable must be explicitly declared in the SPL code.

Example

```
DECLARE   CODE_IN EVENT;                          /* event        */
DECLARE   BGD_$0020 EVENT EXTERNAL;               /* global event */
```

Each declaration of a new event variable causes allocation of a new event control block.

6.2.2 Event Control Block

An event control block of 5 bytes is associated with each event in the system.

From this control block, Task Manager can ascertain:
- whether or not the event has occurred
- whether any task is awaiting this event
- whether anything abnormal has happened (exception)

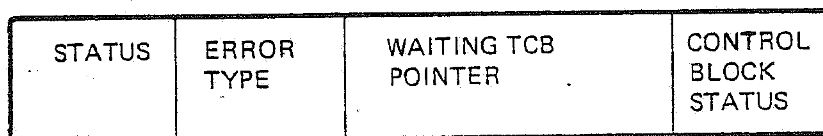


Fig. 6.1 Event Control Block (ECB) layout

The data fields in the event control block have the following meaning:

STATUS	-	This field contains a status byte, which can be read by the user via the built-in function STATUS (event) (see the SPL manual).
ERROR TYPE	-	This field can also be read by the user via built-in function ERRORTYPE (event).
WAITING TCB POINTER	-	The address of a waiting TCB.

CONTROL BLOCK STATUS

Bit 7 6 5 4 3 2 1 0	
X	Post
X (bit 6)	Task waiting
X (bit 5)	Exception
X X X X X (bits 4-0)	Not used

PRIOS uses the STATUS and ERROR TYPE fields in a special way. See section on Printer Functions.

6.3 MULTITASKING FUNCTIONS

The task is the SPL implementation of a logical process. Processing is defined by means of SPL statements for multitasking functions. The general concepts of multitasking are discussed in Section 1.

6.3.1 Task declaration

Each procedure that is intended to be executed as a task must have the TASK option in its declaration.

Example:

```
MY_PROC:
        PROCEDURE OPTIONS (TASK);
        .
        .
        .
        END MY_PROC;
```

This procedure can be attached as a subtask to the attaching procedure. (The attaching procedure is often declared as a task itself.)

Example:

```
MAIN_PROC:
        PROCEDURE;
        DECLARE TP TASK;        /* declare reference to the task */
        .
        .
        CALL MY_PROC TASK(TP) PRIORITY(-1);   /* attach task TP */
        .
        .
        WAIT TP;                /*wait for termination of task TP */
        .
        .
END MAIN_PROC;
```

The attach and wait functions are further discussed below.

6.3.2 Task Control Block (TCB)

The activation of a declared task causes allocation of a task control block of 64 bytes in main storage.

The four first bytes in the task control block can be regarded as an event control block (ECB).

The task control block is used by the Task Manager to store and change information about
- the task state (active, ready, waiting or inactive)
- the absolute task priority
- which other task (if any) is awaiting the termination of this task

The task control block also contains an internal stack to store information needed to permit continued processing after an interrupt or a jump to a subroutine.

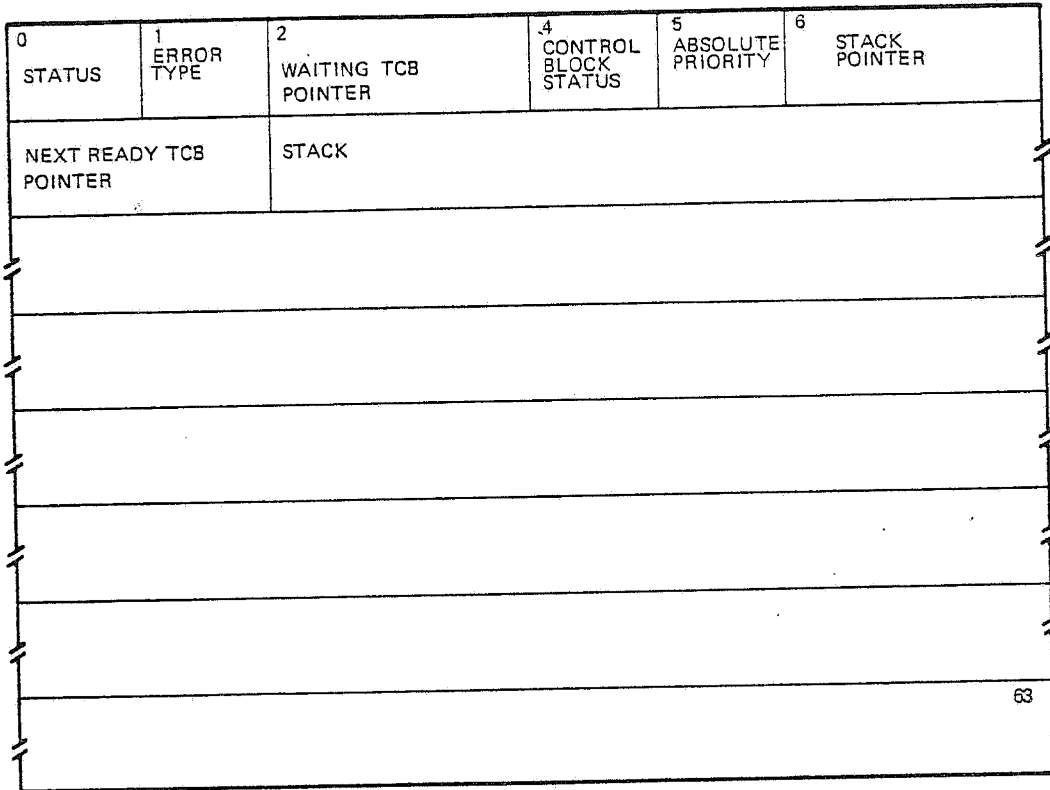


Fig. 6.2 Task Control Block (TCB) layout

The different fields in the task control block have the following meaning:

Field	Meaning
STATUS	This field contains a status byte, which can be read by the user via the built-in function STATUS (task) (see the SPL manual).
ERROR TYPE	This field can also be read by the user via built-in function ERRORTYPE (task).
WAITING TCB POINTER	The address of a waiting TCB.
CONTROL BLOCK STATUS	A byte with the following meaning when bits set:

Bit 7	6	5	4	3	2	1	0	Meaning
X								Post (Task return termination)
	X							Task waiting
		X						Exception (Task exit termination)
			X					Not used
				X				Suspended
					X			Not used
						X		Waiting
							X	Ready

Field	Meaning
ABSOLUTE PRIORITY	The absolute task priority, a number between 0 and 255.
STACK POINTER	Address pointer to the internal stack in the TCB.
NEXT READY TCB POINTER	Address pointer to the next ready TCB.
STACK	A 54 byte field used as internal stack for the task.

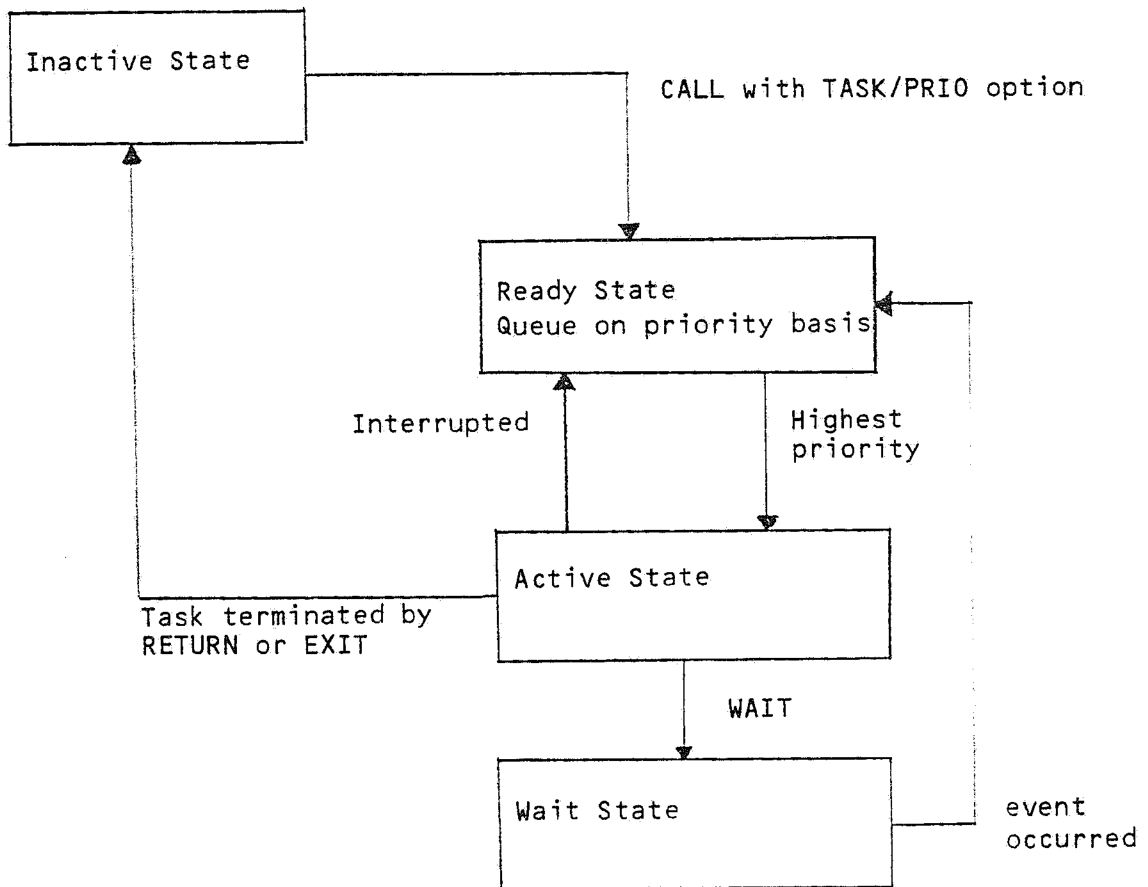


Fig. 6.3 Task States

6.3.3 Attach Task

When a task is declared, it enters the inactive state.

A task is activated (attached) by another procedure issuing a CALL with the TASK and PRIORITY options.

Example:

```
CALL MY_PROC TASK (TP) PRIORITY (-1);
```

The procedure MY_PROC is executed as a task. The task is identified by the task variable TP. The priority of the task is in this case lower than the attaching task's priority.

6.3.4 WAIT Event

A task can be forced to await a certain event in the system. The scope of the event declaration must comprise the procedure in question.

Example:

```
WAIT CODE_IN;
```

CODE_IN must be declared as en event variable. When the wait statement is executed, the event status is checked. If the event has occurred, the execution is immediately continued. If the event has not yet occurred, the executing task enters the waiting state until the event is posted.

Execution of the WAIT statement does not affect the status of the event variable.

6.3.5 POST Event

The POST statement is used to indicate the occurrence of a declared event.

Example:

```
POST CODE_IN;
```

CODE_IN must be a declared event variable. The execution of the POST statement causes an OS request interrupt. If any task was awaiting this event, this task is entered into the ready state.

The task in the Ready state queue having the highest priority continues execution after the interrupt.

6.3.6 ASSIGN Event

The ASSIGN statement is used to reset an event variable to indicate that the event has not occurred.

The ASSIGN statement is often used immediately before or after a WAIT statement, to prevent a looping task from reexecuting on the same event several times.

Example:

```
ASSIGN CODE_IN;
WAIT CODE-IN;
"EXECUTE_CODE"
"LOOP AND WAIT FOR A NEW CODE"
```

CODE_IN must be declared as an event variable.

6.3.7 WAIT Task

The WAIT (task) statement is used when one task is dependent on the completion of another task's execution.

Example:

```
DECLARE TP TASK;
.
.
CALL MY_PROC TASK (TP) priority (-1);
.
.
WAIT TP;
```

In the example above, the execution of task TP is not started until the WAIT statement is executed, since the task TP has a lower relative priority than the attaching task (provided that there is not more than one processor available).

6.3.8 Task Termination

When a task executes the RETURN statement, the task terminates.

The task termination causes the implicitly declared termination event to be posted in the task control block. The posting of this event causes an OS request interrupt. If any task was awaiting the termination, the task is entered into the ready state.

The task in the ready state queue having the highest priority continues execution after the interrupt.

6.3.9 CANCEL Event

Execution of the CANCEL (event) statement causes an abnormal event posting.

An EXCEPTION is marked in the event control block, and an OS request interrupt is obtained.

6.3.10 CANCEL Task

By means of the CANCEL (task) statement, a superior task can force a subtask to terminate abnormally.

An EXCEPTION is marked in the task control block of the terminated task, and the task termination event is posted.

Example:

```
DECLARE TP TASK;
.
.
IF "something wrong"
THEN
  CANCEL TP;
```

6.3.11 EXIT

A task can terminate itself abnormally by executing the EXIT statement.

An EXCEPTION is marked in the task control block, and the task termination event is posted.

For a detailed description of the SPL statements, refer to the SPL Reference Manual.

6.4 INTERRUPT HANDLING

The concept of asynchronous processing is based on interrupt handling.

The interrupt handling is basically performed in six steps:

- The MPU receives an interrupt signal
- The current main program statement is excecuted
- The register contents of the MPU and the program counter are saved on a stack
- The address of the appropriate interrupt routine is loaded into the program counter from an interrupt vector (see below)
- The interrupt routine is executed (the execution can comprise attaching of subtasks)
- The main program is continued

The details of the interrupt handling are hardware dependent. See the Technical Description.

The general concepts are presented below.

The microprocessor reacts to four different interrupts:

- Reset Interrupt
  Power is turned on or the Reset button is pushed

- Non Maskable Interrupt (NMI)
  A non maskable interrupt must always be accepted by the processor. Used differently in different units. See the Technical Description.

- Software Interrupt (SWI)
  This interrupt is generated by program statements such as POST, CALL and RETURN via the OS Request Handler.

- Interrupt Request (IRQ)
  This interrupt type is used for interrupts from peripheral circuits.

The different interrupt types are further discussed below.

6.4.1 Reset Interrupts

The POWER ON interrupt is used to initiate program loading and initialization of the system units. A POWER ON interrupt is initiated by the hardware when power is turned on.

When the external RESET pushbutton is depressed, a bit is set in the hardware in a MIC register to indicate RESET condition and then the POWER ON interrupt is initiated.

The start/restart interrupt handling routines are stored in PROM. A description of these functions is presented in section on Initialization and Logon.

6.4.2 Software Interrupts

Software interrupts are generated in order to access the OS Request Handler.

The OS Request Handler is invoked when SPL statements such as POST, RETURN or CALL are executed.

The OS Request Handler does not perform any processing itself. The processing is carried out in Task Manager and FDIOS, and depends on the parameters issued in the call. The OS Request Handler serves as a "procedure switch" which calls procedures to carry out the desired functions.

6.4.3 Interrupts from Peripheral Circuits

When an interrupt request (IRQ) is obtained from a peripheral circuit (e.g. a keyboard), a certain interrupt procedure is immediately invoked.

The interrupt procedures have a higher absolute priority than all ordinary tasks in the system.

Each type of peripheral interrupt is assigned its own interrupt procedure. The addresses of the interrupt procedures are stored in tables called interrupt vectors.

Interrupts from different peripheral circuits can be assigned various relative priorities. These priorities are called IRQ levels.

IRQ levels, interrupt procedures and interrupt vectors are further discussed below.

6.4.4 Interrupt Levels

Reset, NMI and SWI are non maskable interrupts, i.e. interrupts on these levels cannot be inhibited.

The interrupt type IRQ is divided into eight priority levels, denoted 7-0 where 7 has the highest priority.

The IRQ levels 7-1 are denoted KERNEL, and the lowest level (IRQ LEVEL 0) is denoted INTERRUPT.

The IRQ interrupts are maskable, i.e. interrupts on these levels can be inhibited by means of instructions to the processor. Interrupts can be inhibited in two ways:

- All maskable interrupts inhibited. This is implemented by the SPL statement LOCK(KERNEL). (Machine code SEI for M6800.)
- Only the lowest priority level inhibited. This is implemented by the SPL statement LOCK(INTERRUPT).

The KERNEL levels (7-1) are relative priorities assigned to the different hardware devices. If several IRQ interrupts are waiting simultaneously, they are handled on a priority basis.

6.4.5 Interrupt Procedures

When an interrupt request (IRQ) is obtained from a peripheral circuit, an interrupt procedure is immediately invoked. The interrupt procedures have higher absolute priority than all ordinary tasks.

The interrupt procedures in the operating system are declared with the KERNEL or INTERRUPT options.

Example:

```
KERNPROC:
   PROCEDURE OPTIONS (KERNEL);
   .
   .
   .
   END KERNPROC;
```

```
INTPROC:
   PROCEDURE OPTIONS (INTERRUPT);
   .
   .
   .
   END INTPROC;
```

The KERNEL procedures are automatically invoked via the interrupt vector. INTERRUPT procedures are invoked by calls issued from the KERNEL procedures.

While interrupt procedures on levels KERNEL 7-1 are being carried out, additional KERNEL interrupts are inhibited (LOCK(KERNEL)). This inhibition is carried out automatically by the processor. Interrupt handling on level KERNEL must thus be kept short (<200 micro seconds). In situations where longer processing is needed, it must be carried out on the INTERRUPT level. See the example below.

The execution of interrupt procedures on the INTERRUPT level are carried out while additional interrupts on the KERNEL 7-1 levels are permitted, but while additional interrupts on the INTERRUPT level are inhibited (LOCK(INTERRUPT) and UNLOCK(KERNEL)).

There are a number of restrictions imposed on the KERNEL and INTERRUPT procedures.

A KERNEL procedure must not:

- o Carry out UNLOCK(KERNEL) or LOCK/UNLOCK(INTERRUPT) (LOCK(KERNEL) has no meaning, see above).
- o Require too much time (>200 micro seconds). If a need for an OS Request arises, an INTERRUPT procedure must be initiated which can execute the OS Request.

An INTERRUPT procedure must not:

- o Carry out UNLOCK(INTERRUPT).
- o Require so much time that the queue for processing on the INTERRUPT level becomes full (several tens of milli seconds in exceptional cases).

The operating system contains two procedures which administer all processing on the INTERRUPT level:

- o BLG_00200 which maintains the queue of procedures on the INTERRUPT level and generates IRQs on level 0.

- o BLG_00010 which calls the INTERRUPT procedures in the sequence in which they are queued by BLG_00200. When the queue is empty, the IRQ on level 0 is removed.

See the example at the end of this section.

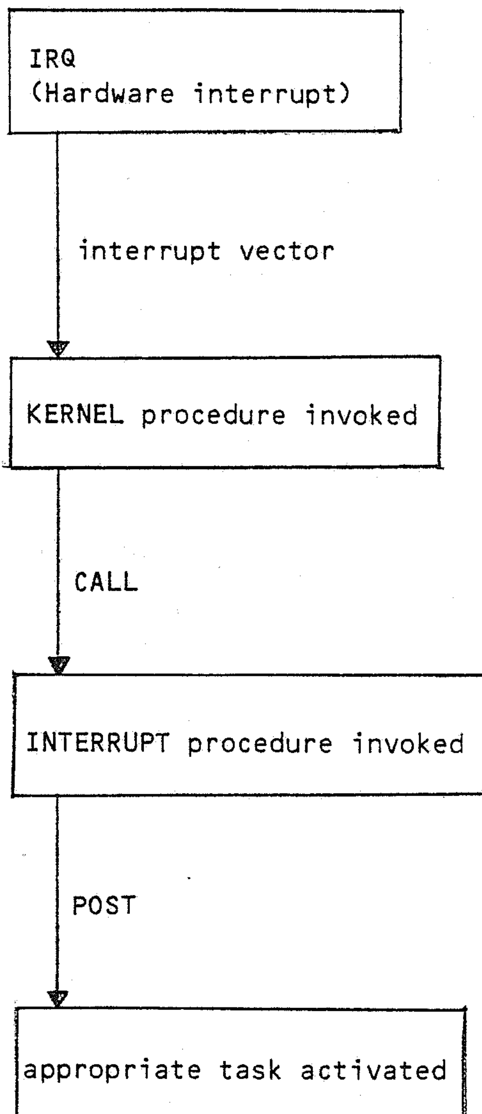


Fig. 6.4 Example of IRQ Interrupt sequence in Alfaskop System 41

6.4.6 Interrupt Vectors

Interrupt vectors are used to enable the processor to start the appropriate interrupt routine when an interrupt occurs.

Each interrupt vector points to a table containing one address for each interrupt type and level. When an interrupt occurs, the processor preserves the current processor status in a stack. This

status consists of the contents of the accumulators, the index register, the status register (condition code register) and the program counter. After that, the interrupt vector for the correct interrupt level is entered into the program counter, i.e. a jump to the interrupt routine is generated.

The interrupt vectors are stored in PROM, and since an interrupt routine must be able to lie anywhere in storage, a 2-stage jump arrangement is used. See Fig. 6.5. The PROM vectors for POWER ON and RESET can point directly to the appropriate interrupt routine since it is also stored in PROM. The vectors for the other interrupt levels refer to fixed addresses in RWM where the jump instructions are stored.

After program loading, the addresses in the jump instructions point to a dummy procedure in PROM. Before an interrupt on a given level can be permitted, the address part of the jump instruction must thus be initialized with the start address for the interrupt procedure in question. The method of storing interrupt addresses in RWM permits a number of interrupt routines (based on previous events) to be used on each interrupt level.

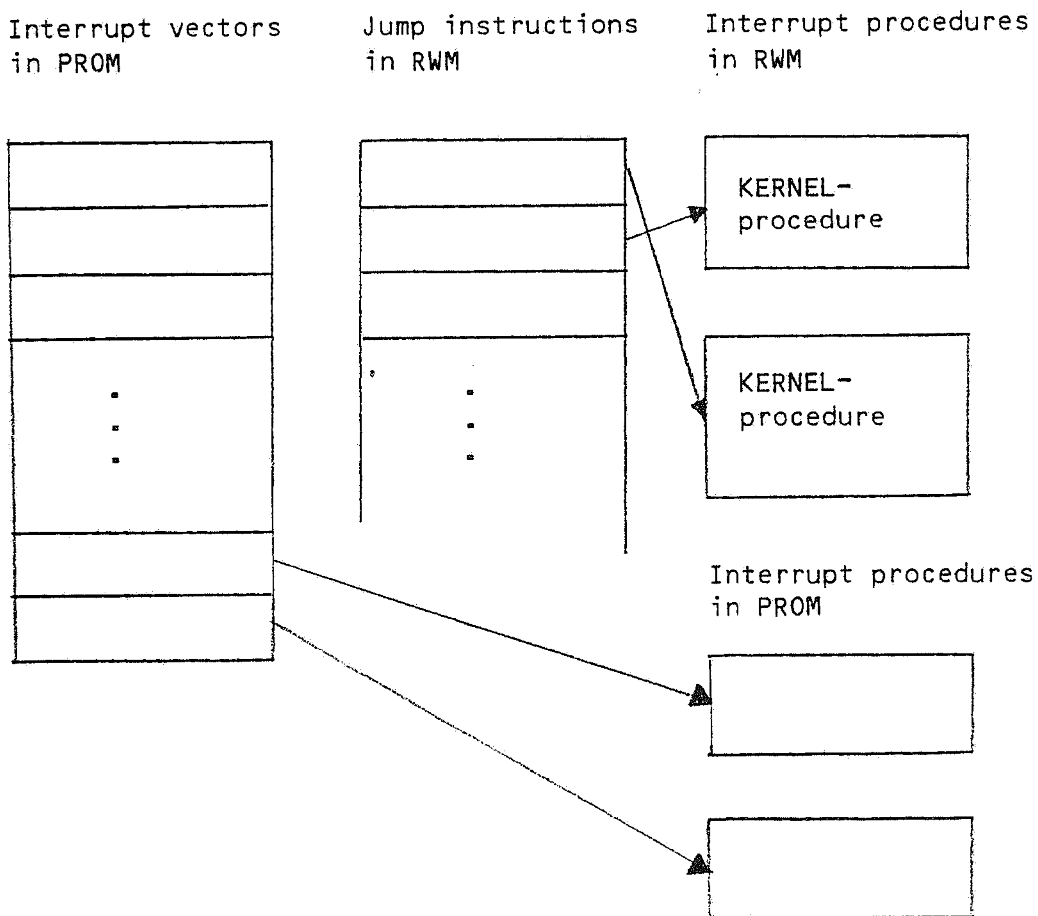


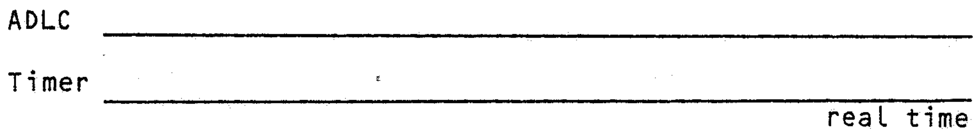
Fig. 6.5 Interrupt vector organization in DU 4110

6.4.7 Examples of Interrupt Handling

Example 1.

This example describes the handling of two different interrupt requests (IRQ). The first peripheral circuit is a timer, the second is an ADLC circuit (Advanced Data Link Controller). An idle loop is executed when no interrupt is present.

Peripheral circuits IRQ (HW interrupts)



Procedures executing

KERNEL level:

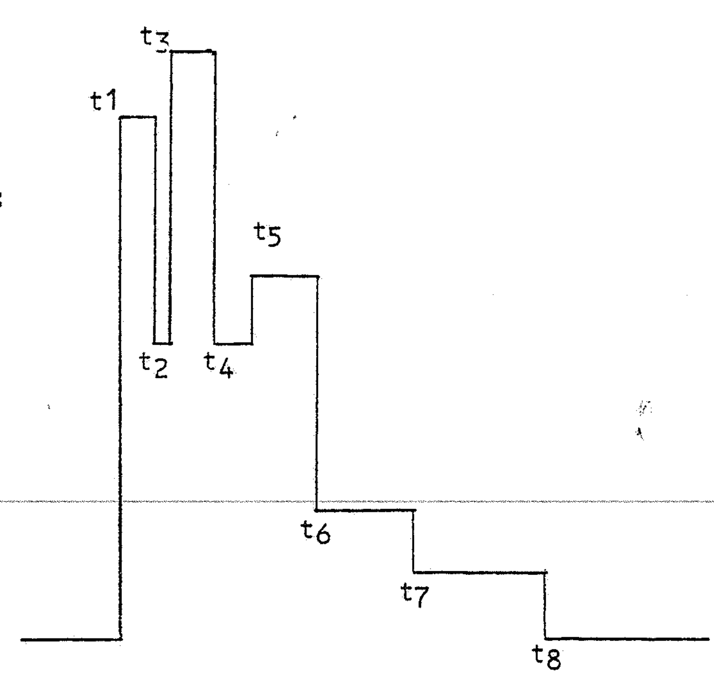


Fig. 6.6 Example of IRQ handling

See the notes on next page

Real time	Notes to example 1
t_1	A timer IRQ occurs. Via an interrupt vector, a KERNEL procedure is invoked. A KERNEL procedure cannot be interrupted by another IRQ.
t_2	When the execution of the KERNEL procedure is completed, an INTERRUPT procedure is called.
t_3	During the execution of the timer's INTERRUPT procedure, an ADLC IRQ occurs. The ADLC KERNEL procedure is immediately invoked.
t_4	When the execution of the ADLC KERNEL procedure is completed, its INTERRUPT procedure is called. However, the queue of waiting INTERRUPT procedures is not empty. The timer's suspended INTERRUPT procedure must be executed first.
t_5	When the timer's INTERRUPT procedure is executed, an event is posted to the timer's task. After that, the ADLC INTERRUPT procedure is executed. When this procedure is completed, another event is posted to the ADLC task.
t_6	Two tasks are now ready to be executed. In this case the ADLC has been assigned a higher priority, and is thus activated first.
t_7	The timer's task is activated as soon as the ADLC task is completed.
t_8	Both IRQs are handled, and the idle loop is executed again.

Example 2

This example shows how a communication procedure can be implemented.

See also notes on the last page of this section.

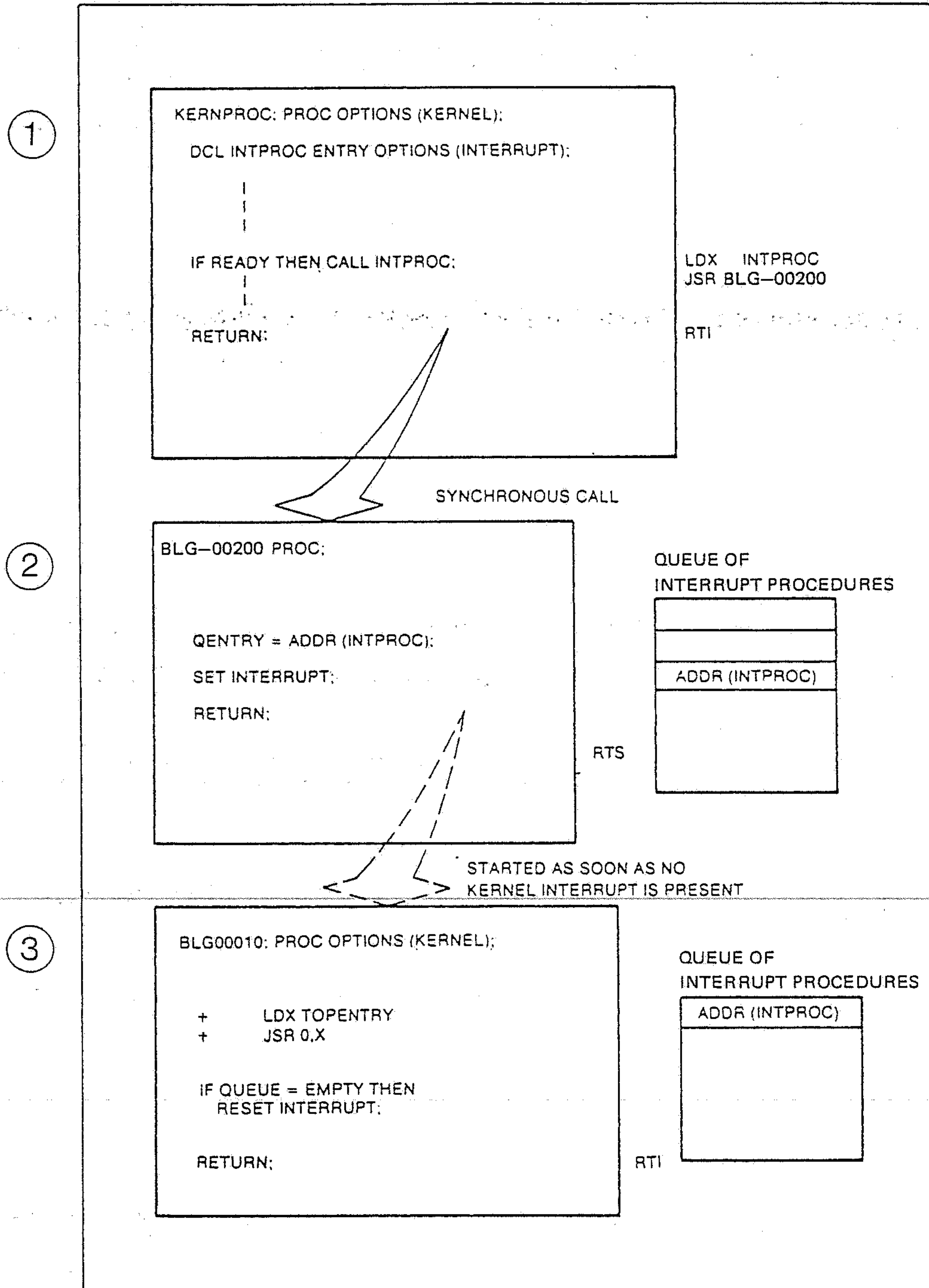


Fig. 6.7 Example of interrupt handling (continued on next page)

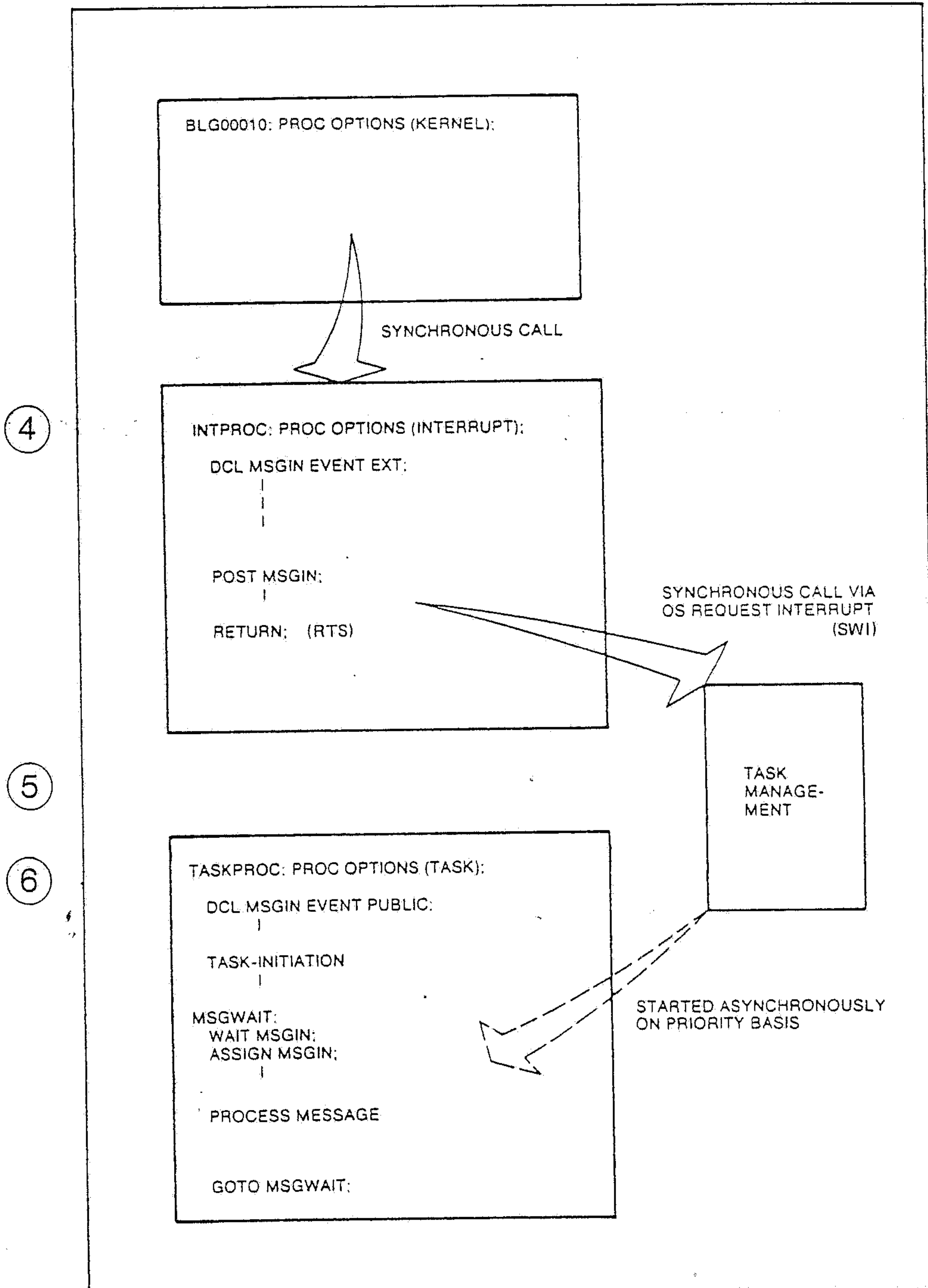


Fig. 6.8 Example of interrupt handling (continued)

Notes (Example of interrupt handling)

1 KERNPROC is assumed to be an interrupt routine on the KERNEL level. This procedure receives a message character by character. When the entire message has arrived, further analysis is to be performed by the procedure INTPROC on level INTERRUPT.

2 The call for the procedure INTPROC causes an implicit call for the OS procedure BLG_00200 which queues the call for INTPROC, and initiates an IRQ interrupt on the INTERRUPT level.

3 When no inrerrupt on higher priority levels are present, the OS procedure BLG_00010 is called. BLG_00010 initiates the first procedure in the INTERRUPT procedure queue. If the queue is empty, the IRQ interrupt on the INTERRUPT level is removed.

4 If INTPROC is the first in queue, it is now executed. When the received message is analysed and checked, the event MSGIN is posted. The posting of the event causes an OS request interrupt (SWI).

5 The OS request interrupt is handeled by the Task Manager as described in section on Multitasking Functions.

6 If any task was waiting for the event MSGIN to be posted, the task is now entered into the Ready state queue.

Contents

7.1	COMMUNICATION CONCEPTS	3
7.1.1	Communication Software	3
7.1.2	Communication Channels	5
7.1.3	Sessions	5
7.1.4	Session Control Block	6
7.1.5	Status Lists	6
7.2	SYSTEM ADDRESSES	7
7.2.1	Physical Addresses	7
7.2.2	Logical Addresses	7
7.3	GENERAL MESSAGE FORMAT	8
7.3.1	LEADING FLAG byte	8
7.3.2	TODEV Byte	8
7.3.3	FRDEV Byte	9
7.3.4	DSA Byte	9
7.3.5	MSGTYP/STATUS Byte	10
7.3.6	MESSAGE CONTENT Field	10
7.3.7	CRCC Byte	10
7.3.8	TRAILING FLAG Byte	10
7.4	MESSAGES	11
7.4.1	Poll Messages	11
7.4.2	Answer to Poll Messages	13
7.4.3	Communication Control Messages	16
7.4.4	Data messages	19
7.5	COMMUNICATION EXAMPLES	21
7.5.1	General Communication Example	21
7.5.2	Poll and Answer to Poll	25
7.5.3	CP IPL Session	27
7.5.4	DU IPL Session	28
7.5.5	File Update Session	30
7.5.6	User Interface Sequence	36
7.5.7	Printer Communication Sequence	37-38

7.1 COMMUNICATION CONCEPT

Communication between the different terminal units (CP, DU, FD, PCU) of Alfaskop System 41 is carried out via SS3 bus, i.e. either shilded twisted pairs or coaxial cable. Transmission is balanced. Data is transmitted serially in an HDLC frame format.

There are four main types of messages used in Alfaskop System 41 Internal communication.

- o Poll
- o Answer to poll
- o Communication control messages (session control)
- o Data messages.

7.1.1 Communication Software

The software handling the internal communication consists of the following main modules:

	Used in:
o Communication Handler	CP DU FD
o User Interface Module (UIM)	CP DU
o Supervisors e.g.	
- Input/output Manager (IOM)	FD
- FD I/O Supervisor	CP DU
- Printer I/O Supervisor	DU
- IPL Supervisor	CP
- Console Mode Supervisor	DU
- Utility Supervisor	CP DU FD

The modules used in PCU are the same as used in DU.

The Communication Handler handles the channel hardware, connects terminal units to the channels and handles poll sequences. It also handles the transmission and reception of messages.

UIM provides the interface between the system modules and the Communication Handler. See the next main section.

For details of the hardware and physical message format see Technical Description. See also Fig. 7.1

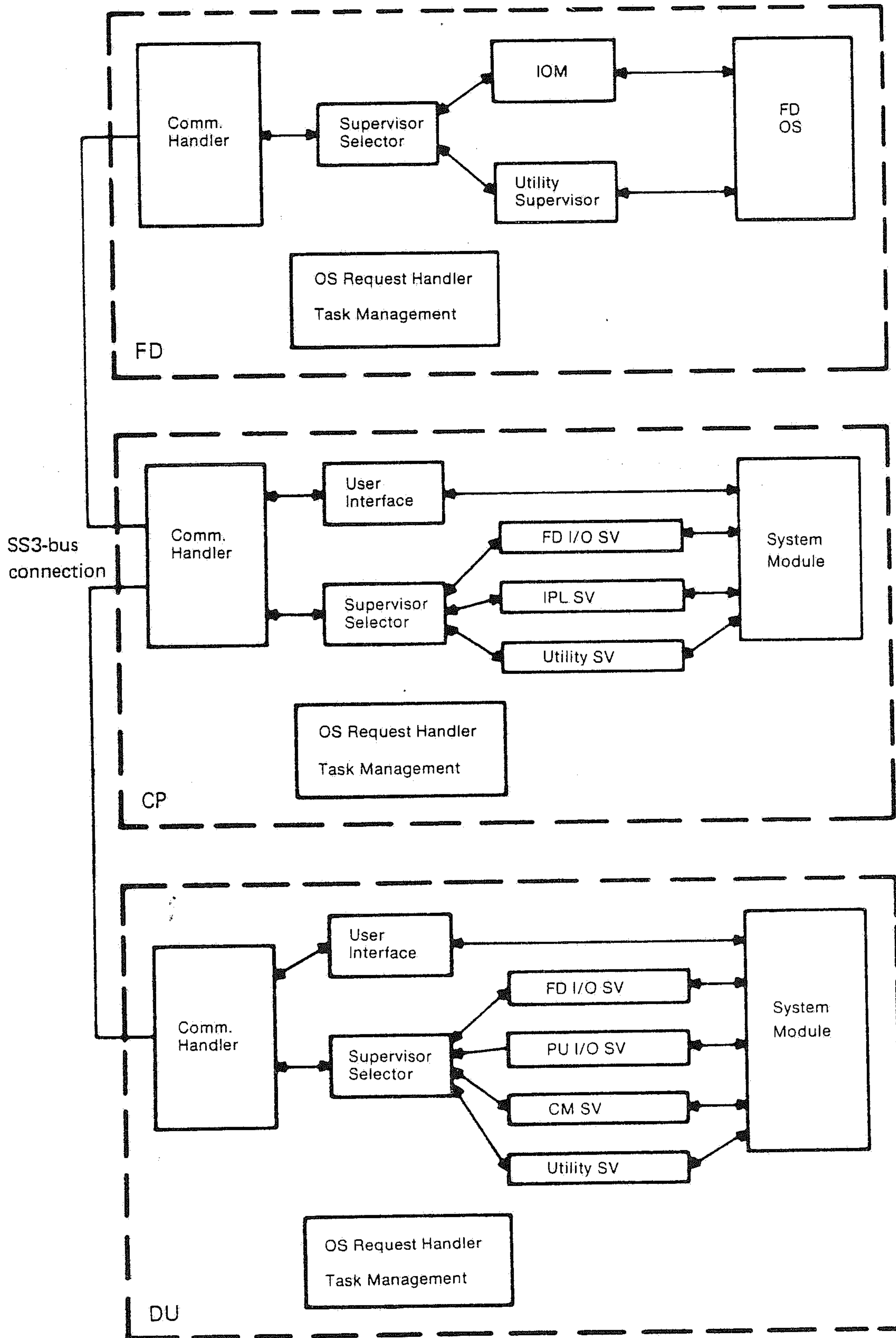


Fig. 7.1 Internal Communication Software

7.1.2 Communication Channels

Four communication channels are available in a cluster configuration. The transmission rate is 300 kbps for each channel. The four channels are numbered 0-3 and are assigned the following communication functions:

Channel	Function
0	Reserved for internal poll sequences
1	Communication between the CP Supervisor and a Supervisor in another terminal unit. If not used for this type of communication, it is a spare for TP-TP communication.
2	User interface (first emulation).
3	User interface (second emulation). If no second emulation exists, it is a spare for TP-TP communication.

For each channel there are two 64 bytes memory buffers, one for the last message received and one for the next message to be transmitted. Users of the internal communication may also define their own data areas for reception and transmission of data.

7.1.3 Sessions

In every terminal configuration, one terminal unit controls the physical communication. This unit is called configuration master. In a cluster configuration the CP is always configuration master (system master).

The physical communication within the cluster is implemented as a poll/connect system. The CP polls the devices for output messages. When a polled device issues a negative response, i.e. has got nothing to transmit, the next device in the poll list is polled.

Besides the physical communication, there is a logical concept of communication. The logical concept of an internal communication is called a session. When a unit has given a positive answer to a poll, a session is opened and is then regarded as going on until it is concluded by a special command. A session may consist of one or several sequences of physical communication. The channel is disconnected after each sequence of physical communication and may thus be used by other units, while processing is done by the first units.

The device which called for the opening of a session is session master. The other unit is session slave. Any unit in a terminal configuration may be session master. The session mastership may be changed several times during a session. Note that the configuration master is not always session master.

7.1.4 Session Control Block

A session is identified by a Session Control Block (SCB) in each of the communicating units, including the configuration master. The SCB is registered until the session is closed or aborted. The SCB is identified by a session number and sometimes a supervisor number. This identification is contained in the DSA byte of each internal message. See section 7.3.4.

One SCB is always reserved for the User Interface Module.

7.1.5 Status Lists

The operating system module in the CP maintains a list of all connected units, indicating their current status. The status is updated after each poll response.

The list contains four bytes for each port, i.e. one byte for each DU, PU, FD and PCU on the port.

The start address of the list in the CP is 0400 hexadecimal.

Each status byte in the status list is interpreted as follows:

Bit 7	6	5	4	3	2	1	0	Meaning
1								Device not connected on two-wire
	1							Device down
		1						IPL request
			1					Host reservation
				1				Session table full (Max 64)
					1			Master lock
						1		Secondary host in dual host config.
							1	Primary host

Note. If bit No. 6 is set for a device in the poll list, CP issues slow polls to this device.

The TP-status list in OS is not identical to the emulation status table maintained by the system module in the CP. See section 8.9.

In each DU, OS maintains a status area indicating the current status of the DU/PU and the status byte in the last poll from CP. This area is called BCG_70113. See also section 8.11.

7.2 SYSTEM ADDRESSES

7.2.1 Physical Addresses

The communication processor provides for up to 32 ports for internal cluster communication. To each port can be connected a chain of up to four devices all of which must be of different types.

Device type

0	Display Unit
1	Printer Unit connected to the DU
2	Flexible Disk unit
3	Peripheral Control Unit (with printer connected).

The physical address is constituted by the port No. (0-31) and the device type.

The format of the physical address byte is as follows:

Bit 7 6 5 4 3 2 1 0	
1 1 1 1 1 1 0 1	System FD (FD hexadecimal)
1 1 1 1 1 1 1 0	CP (FE)
X X X X X X 0 0	DU
X X X X X X 0 1	PU, PCU with printer connected
X X X X X X 1 0	FD
X X X X X X 1 1	PCU

where XXXXXX denotes the port number.

The physical address is used in the TODEV and FRDEV address bytes in the internal communication protocol. See section 7.3.2.

7.2.2 Logical Addresses

Alfaskop System 41 provides for up to 32 logical addresses to be used internally. The logical addresses are assigned to the system units during customizing.

The logical addresses are constituted analogously to the physical addresses, but the port number is replaced by a number assigned by the user.

In this manual, the logical addresses are symbolically denoted DU01, PU05 etc.

See also the document on Terminal Console Functions.

7.3 GENERAL MESSAGE FORMAT

The message format used in communication between Alfaskop System 41 terminal units is as follows:

- o Word length 8 bits
- o Message length 4-4096 bytes, depending on type of message, FLAGs and CRCC excluded.

The general message format is shown in the figure below.

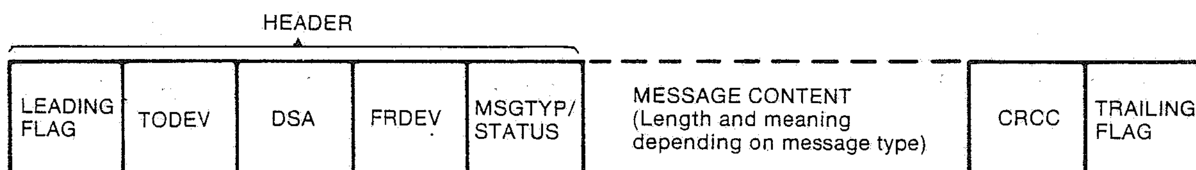


Fig. 7.2 General Message Format

Except in the flag character, which contains six consecutive ones, more than five consecutive ones are not sent in a frame (between flags). This is prevented as the transmitter logic inserts a zero after five ones and the receiver logic always strips a zero following five received ones. If 01111110_{12} anyhow is received that is interpreted as a flag.
If seven or more consecutive ones are received, inside a message, that is interpreted as an abort indication telling the receiver logic that the frame is valid.

7.3.1 LEADING FLAG byte

The LEADING FLAG byte is a control byte used for indicating start of message and for byte synchronization. The flag is added and stripped by hardware logic.

7.3.2 TODEV Byte

The TODEV byte contains the physical address of the unit to which the message is sent. See figure below.

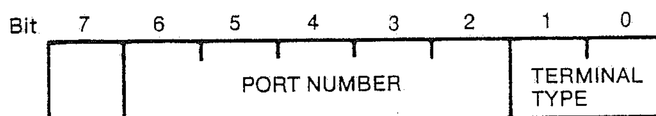


Fig. 7.3 TODEV and FRDEV layout

The fields in TODEV have the following meaning:

- o PORT NUMBER — Number of two-wire connection on the TUA boards of CP (0-31).
- o TERMINAL TYPE (hex) — 00 = DU
  01 = PU
  10 = FD
  11 = PCU

  Furthermore FE_{16} is reserved as CP address and TODEV FD_{16} as a System FD address.

7.3.3 FRDEV Byte

The FRDEV byte contains the physical address of the unit from which the message is sent. The layout of the byte is the same as for the TODEV byte. See above.

7.3.4 DSA Byte

A session is a logical connection between two devices. The two devices are identified by the destination address (TODEV) and the source address (FRDEV).

A session generally consists of several communication sequences, i.e. the physical connection is broken and may be used by other sessions during the time it takes for the device in the first session to prepare its answer. However, the session control block (SCB) remains.

The desired session in the receiving unit is identified by a destination session address, DSA, in each transmitted message.

The DSA layout is shown below.

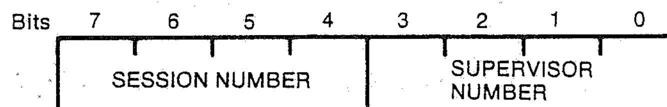


Fig. 7.4 DSA byte layout

The fields in the DSA byte have the following meaning:

- o SESSION NUMBER — Number of the registered session ($\leq$ F). A DU may register up to 15 session including the attached PU. An FD registers up to 15 sessions. If SESSION NUMBER = F_{16}, SESSION NUMBER is asked for (undefined).

o SUPERVISOR

NUMBER		Used in:
0	Input/Output Manager (IOM)	FD
1	Utility Supervisor	DU CP FD
2	FD I/O Supervisor	DU CP
3	Printer I/O Supervisor	DU
4	Console Mode Supervisor	DU
5	IPL Supervisor	CP
E	Host 1 Supervisor (User Interface)	DU CP
(F	Host 2 Supervisor (User Interface)	DU CP) not used yet

In a poll message, DSA can be used to define a drop address on a SS3-bus.

In message types where DSA is not used, the DSA byte has the value FF_{16}.

7.3.5 MSGTYP/STATUS Byte

This byte defines the status of the master in a poll message, and defines the status of the polled device in an answer to a poll. In all other messages the byte defines the message type e.g. acknowledgement, data, abort, connect request, etc.

The MSGTYPE/STATUS byte is further discussed in connection with the various messages below.

7.3.6 MESSAGE CONTENT Field

This field normally contains 0-4092 bytes, depending on the message type. It may even contain more than 4092 bytes, but this is not recommended. The messages are described in Section 7.4 below.

7.3.7 CRCC Byte

The CRCC byte contains a Cyclic Redundancy Check Character, used to check that transmission was correctly performed. The CRCC is calculated and added by the hardware of the sending unit. The same calculation is performed on the message by the hardware of the receiving unit. If the result of this calculation corresponds to the received CRCC, the CRCC is stripped (by hardware logic) and the message is accepted.

7.3.8 TRAILING FLAG Byte

The TRAILING FLAG byte defines the end of the message. The flag is added and strapped by hardware logic.

7.4 MESSAGES

As mentioned above, there are four main types of internal messages in Alfaskop System 41

- o Poll message
- o Answer to poll
- o Communication control message
- o Data message.

They are all transmitted in accordance with the general message format as described in the preceding section.

The differences between the various messages appear in the MSGTYP/STATUS byte and, of course, in the MESSAGE CONTENT field.

7.4.1 Poll Messages

A poll message consists of four bytes. There is no information in the MESSAGE CONTENT field. The DSA field has always the content FF since no session control block is used. The FRDEV byte has always the content FE, since the configuration master always performs the poll.

All connected units are polled by the configuration master using a certain frequency. When a unit gives a negative answer, the next unit is polled, and so on until all units in the cluster are polled. If a unit gives a positive answer, the unit is addressed again until it gives a negative answer, then the next unit is polled.

In a poll message, the MSGTYP/STATUS byte of the message header is interpreted as STATUS.

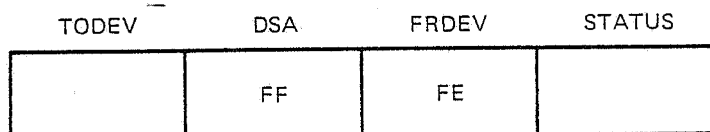


Fig. 7.5 Poll message layout

In a poll message, the various bits in the STATUS byte have the following meaning (if set):

Bit 7	6	5	4	3	2	1	0
1	RETR.	LEADING POLL	ODD	ABORT PENDING SESSIONS	RESERVED FOR EM.	CP	SYSTEM READY

Fig 7.6 STATUS byte in a poll message

Bit 7 — Always 1.

RETR. — Poll retransmission. A poll retransmission is performed when the configuration master has not got an answer on the previous poll before a timeout. If no answer is obtained to the poll retransmission, the polled unit is regarded as "down" and polled more seldom (slow poll).

LEADING POLL — The first poll in a series of polls to a terminal unit. A unit is polled as long as it has got something to transmit. When it has got nothing to transmit it sends a negative poll answer and the next poll to the unit will then be a leading poll.

ODD — Odd poll. The first and then every second poll to a unit in a sequence of polls.

ABORT PENDING SESSIONS — Delete all session control blocks.

Bit 2 — Reserved for emulation (System Reset in IBM Local emulation).

CP — When bit set secondary CP is in control, when reset primary CP is in control.

SYSTEM READY — Host communication possible on modem line No. 1. (System ready set).

7.4.2 Answer to Poll Messages

The figure below shows the various answers to poll messages.

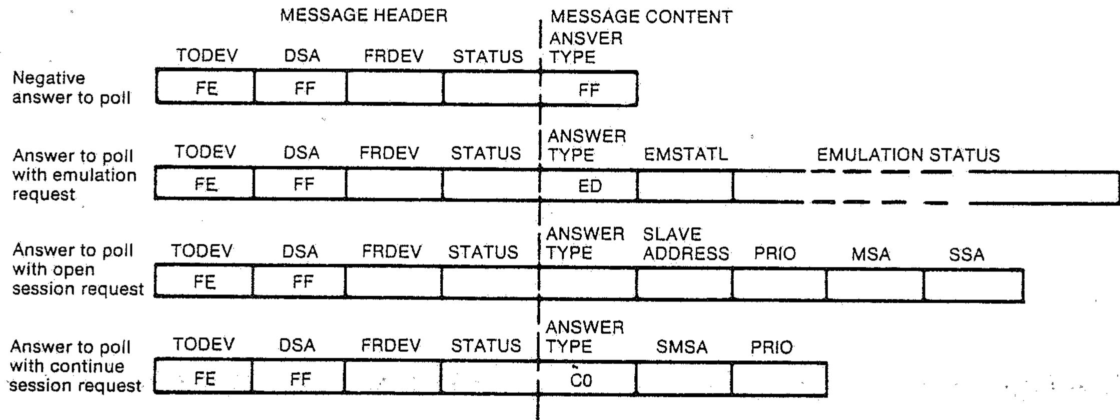
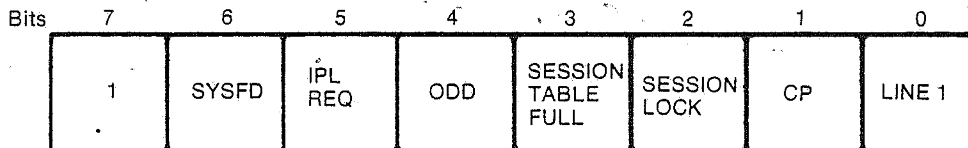


Fig. 7.7 Answer to poll messages

In all answers to poll messages, TODEV = FE, since the answer is sent to configuration master, and DSA = FF since session number is still undefined.

In an answer to a poll message, the MSGTYP/STATUS byte is interpreted as STATUS.



~~Fig 7.8 STATUS byte in answer to poll.~~

In an answer to a poll, the various bits in the STATUS byte have the following meaning:

Bit 7	Always 1.
SYSFD	Polled unit carries the system diskette.
IPL REQ	Request for initial program loading.

ODD — Odd poll reponse in a serie of polls to the same unit.

SESSION TABLE FULL — The polled unit cannot open any more sessions.

SESSION LOCK — Slave connection of unit not accepted.

CP — When bit set secondary CP is in control. When reset primary CP in control.

LINE 1 — Unit is able to communicate with host.

The bytes in the MESSAGE CONTENT field in an answer to poll message have the following meaning:

ANSWER TYPE — The type of answer to the poll.

- o In a "negative answer to poll" the ANSWER TYPE has the value FF and is the only byte in the MESSAGE CONTENT field.
- o In an "answer to poll with emulation request" it also indicates the host system number.
- o In an "answer to poll with open session request" the ANSWER TYPE value may be 0 or 80, thereby marking the address type of the following SLAVE ADDRESS byte. 0 means physical device address (see Section 7.2.1), 80 means logical device address, (see Section 7.2.2).
- o In an "answer to poll with continue session request" the ANSWER TYPE has always the value 'C0'.

EMSTATL — The number of bytes in the following EMULATION STATUS field. Max 58 bytes.

EMULATION STATUS — A field used for fast transmission of emulation defined status to CP, e.g. "ENTER key pressed". See Section 8.9.

MSA — Master session address, gives the session identification of the transmitting device, i.e. the master in the session that is to be established.

SSA — Slave session address (≥ F0), gives the supervisor identification of the slave in the session that is to be established. If the session number is not yet defined, this byte contains 0F.

SLAVE ADDRESS — The address of the session slave. The preceeding byte indicates if the address is physical or logical.

PRIO

A session is always assigned a certain priority. The PRIO byte in an answer to poll message with open or continue session request contains this priority. It decides in which queue in the configuration master the request will be entered.

The PRIO byte can contain the values 00 - 0E, where 0E means highest priority.

SMSA

System Master Session Address in the configuration master. The system master session address contains information on each opened session. SMSA is a number between 00 and FF. This number is transmitted from the configuration master when the session is opened. When a session master wishes to continue a session, the poll response must contain this number.

7.4.3 Communication Control Messages

Communication control consists of functions for opening and ending of sessions, and functions for connection and disconnection within an open session.

A session is opened when the system master sends a connection message to the session slave (SCONN) and a connection message to session master (MCONN). After the transmission is completed, a disconnection is made (EOT or BREAK) but the session still persists. A new connection within the session is made after a continue session request from the current session master of the session.

The session is ended (EOS) when no more transfer between the session master and the session slave is needed.

The device that has called for a session is appointed session master. The session master ends the session with an EOS message.

The figure below shows the various communication control messages.

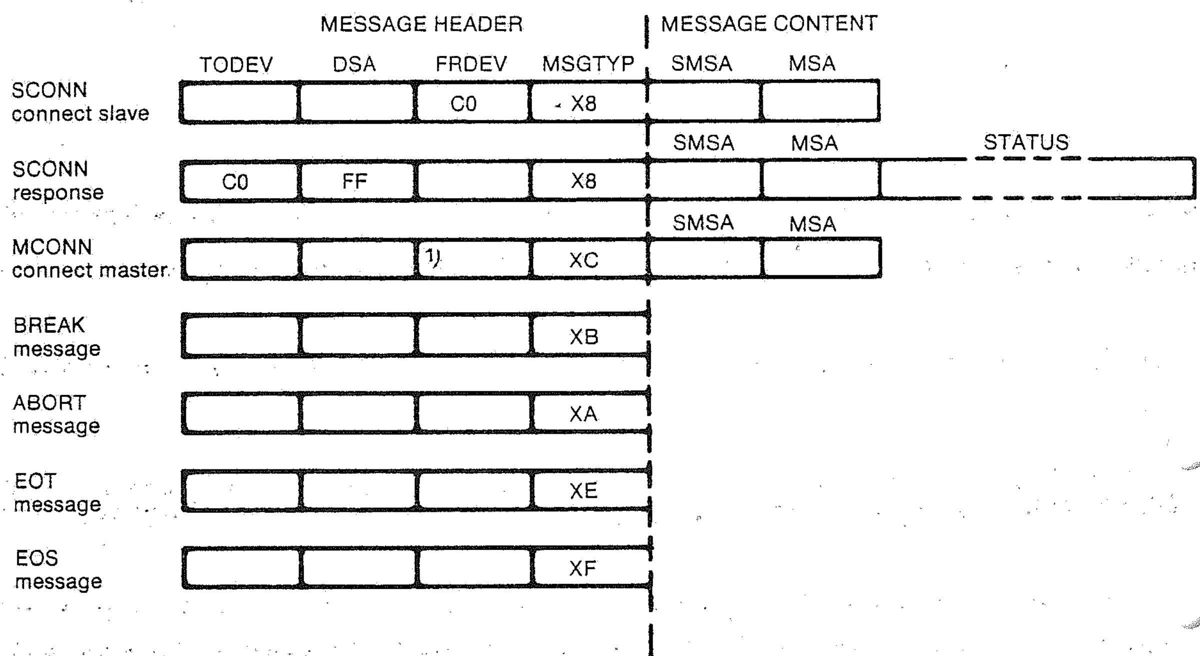


Fig 7.9 Communication control messages

Note 1) This byte defines the address of the session slave.

In all communication control messages, the MSGTYP/STATUS byte is interpreted as MSGTYP. The MSGTYP byte is explained below

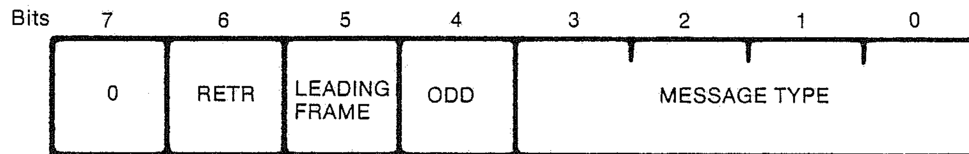


Fig. 7.10 MSGTYP byte layout

Bits 7-4 in the MSGTYP byte in a communication control message have the following meaning:

Bit 7	Always 0.
RETR	Retransmission, used by session or configuration master only, when it has not got an answer to a message before a timeout.
LEADING FRAME	Used by configuration master only in first frame of a communication sequence.
ODD	Odd message, used by slave when received together with RETR, to check whether it has already received the message or not.

Bits 3-0 in MSGTYPE byte can have the following meanings in a communication control message:

MESSAGE TYPE	Mnemonic	Meaning
8 (hex)	SCONN	Connection message to a session slave from configuration master, or connection acknowledgement from session slave to configuration master.
A	ABORT	Abort message, containing information on which session to abort. The message is sent from configuration master to session slave. It can also be sent from session master to configuration master in response to a MCONN message.
B	BREAK	Change of session mastership. Sent from session slave to session master. Also sent from session master to configuration master which results in a disconnection.

MESSAGE TYPE	Mnemonic	Meaning
C	MCONN	Connection message from configuration master to session master.
E	EOT	End of transmission. Sent by session master to system master to order disconnection without mastership change but not ending of session.
F	EOS	End of session. Sent by session slave to session master and from session master to configuration master to order disconnection and ending of session.

The MESSAGE CONTENT field is explained in section on Answer to poll above.

7.4.4 Data messages

Data transfer between devices in the system can take place when a master is connected to a slave within a session.

Messages too long for the received buffer must be preceded by a data header, DH, or a information data header, INFO DH, message. In DH and INFO DH messages, the length of the data message is specified.

Apart from the normal four byte header, both the DH and INFO DH frames contain

- o The first four bytes of the data to be transferred
  These are sent separately, in order to save memory area. At next DMA-transfer an entire frame (including header) is written into the RWM. The header is after a complete transfer replaced in the RWM by the first four data bytes of the preceding frame.

- o Two bytes defining the length of the data message

When the DH message is acknowledged by the receiving unit with a RTRD-message the data message is transferred to the non-standard buffer in the receiving device. The DATA and INFO messages contain

- o The normal four byte header

- o Byte 5 and following of the data to be transferred.

A maximum of 60 information bytes can be transmitted in an INFO message. Up to 4 kbytes can be transmitted in a DATA message.

The figure below shows the various data messages.

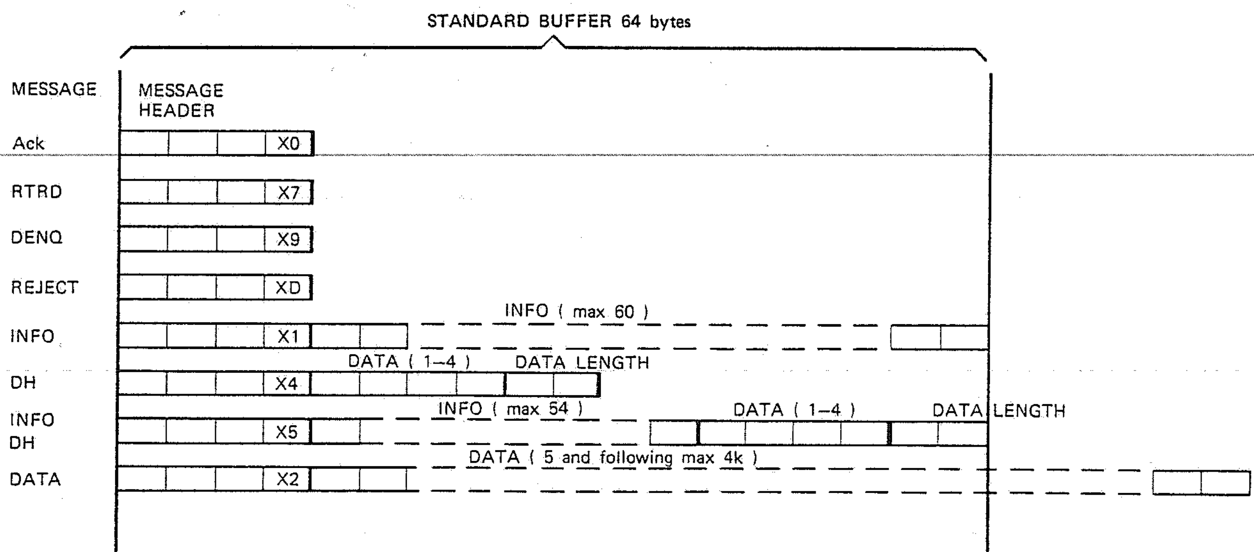


Fig 7.11 Data messages

In a data message the MSGTYP/STATUS byte is interpreted as MSGTYP. Bits No. 7-4 in the MSGTYP byte have the same meaning as described in Section 7.4.3 above.

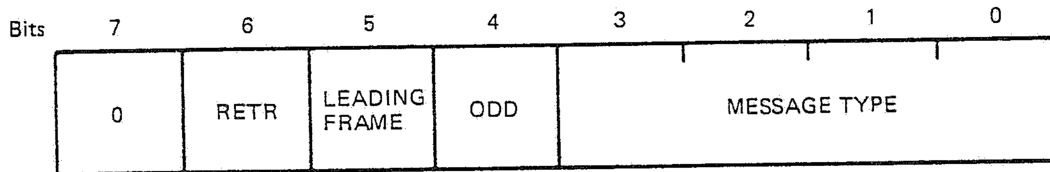


Fig. 7.12 MSGTYP byte layout

Bits 3-0 in the MSGTYPE byte can have the following meanings in a data message:

MESSAGE TYPE	Mnemonic	Meaning
0 (hex)	ACK	Acknowledge to INFO, INFO DH, DATA. Other messages may also work as acknowledge, e.g. BREAK or EOS.
1	INFO	Information frame, max 60 bytes of information.
2	DATA	A long data message. This message must be preceded by a DH message, and is always sent after reception of an RTRD message.
4	DH	Data header, first 4 bytes data bytes are coming in this frame. The rest of the data in next frame (DATA).
5	INFO DH	Information and first 4 data bytes in this frame. The rest of the data in next frame (DATA).
7	RTRD	Acknowledge and ready to receive data. Used as acknowledgement to DH and INFO DH messages.
9	DENQ	Data enquiry. Sent by session master when no answer obtained within a certain time to a DATA message.
D	REJECT	Sent from a unit which is not presently belong able to handle a request. Session master queues up a new request.

In a data message the MSGTYP/STATUS byte is interpreted as MSGTYP. Bits No. 7-4 in the MSGTYP byte have the same meaning as described in Section 7.4.3 above.

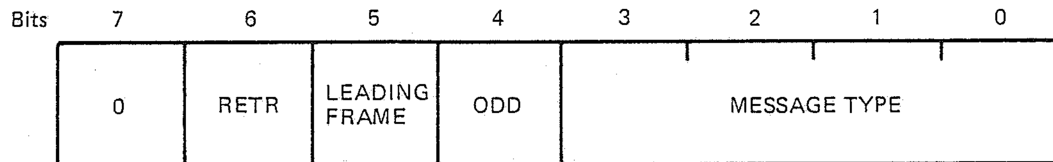


Fig. 7.12 MSGTYP byte layout

Bits 3-0 in the MSGTYPE byte can have the following meanings in a data message:

MESSAGE TYPE	Mnemonic	Meaning
0 (hex)	ACK	Acknowledge to INFO, INFO DH, DATA. Other messages may also work as acknowledge, e.g. BREAK or EOS.
1	INFO	Information frame, max 60 bytes of information.
2	DATA	A long data message. This message must be preceded by a DH message, and is always sent after reception of an RTRD message.
4	DH	Data header, first 4 bytes data bytes are coming in this frame. The rest of the data in next frame (DATA).
5	INFO DH	Information and first 4 data bytes in this frame. The rest of the data in next frame (DATA).
7	RTRD	Acknowledge and ready to receive data. Used as acknowledgement to DH and INFO DH messages.
9	DENQ	Data enquiry. Sent by session master when no answer obtained within a certain time to a DATA message.
D	REJECT	Sent from a unit which is not presently belong able to handle a request. Session master queues up a new request.

The MESSAGE CONTENT field of a data message can have the following meaning:

INFO	For an INFO message this is a 60 byte field (max) containing user defined information. For an INFO DH message the field contains 54 bytes (max). This can also be regarded as a short data message.
DATA	Data is transmitted in two messages. The first four bytes are transmitted in DH or INFO DH message and the rest of the data in a data message.
DATA LENGTH	A two byte field containing the length of the data sent in the following DATA message.

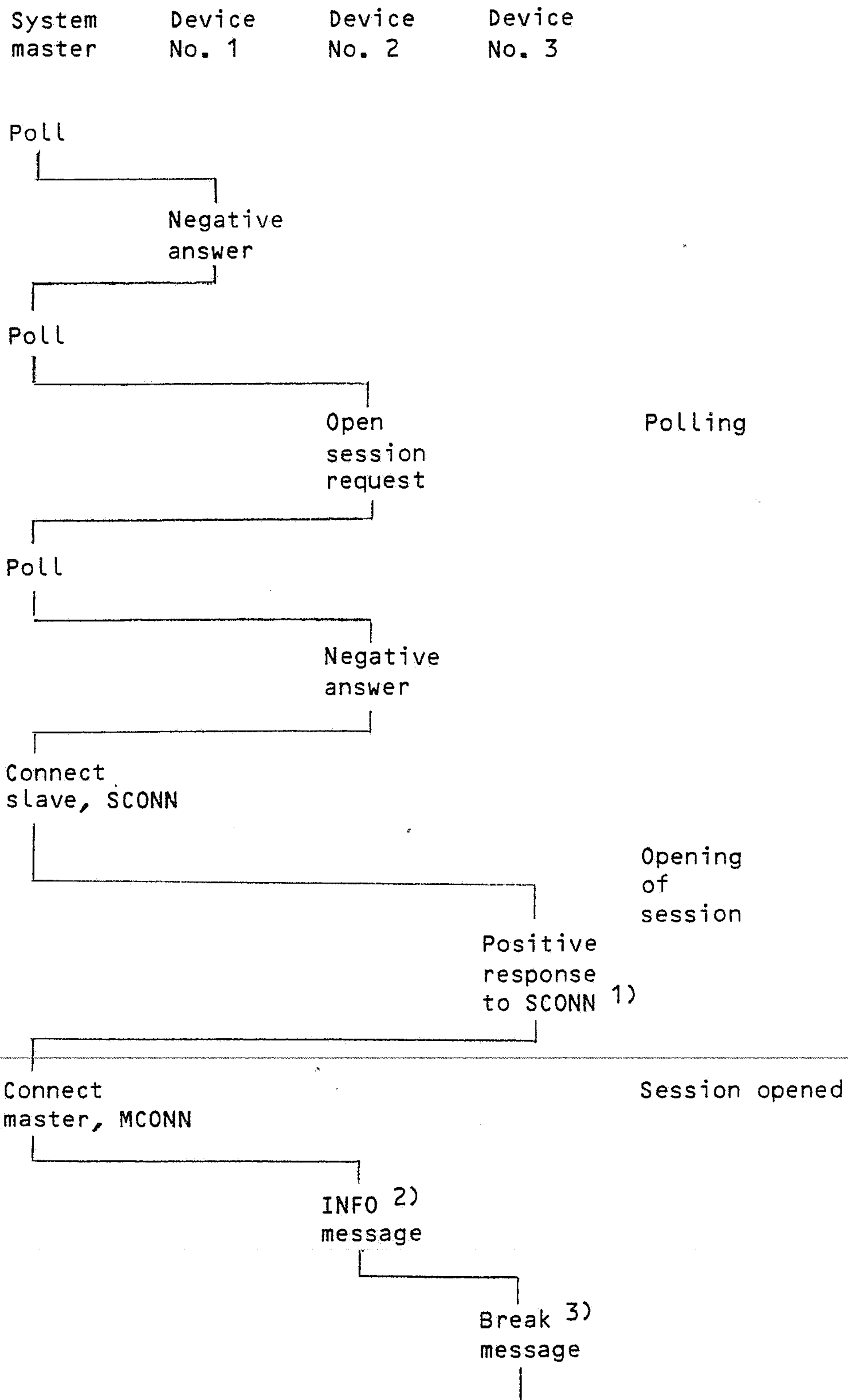
7.5 COMMUNICATION EXAMPLES

The following examples show a number of typical internal cluster communication sequences.

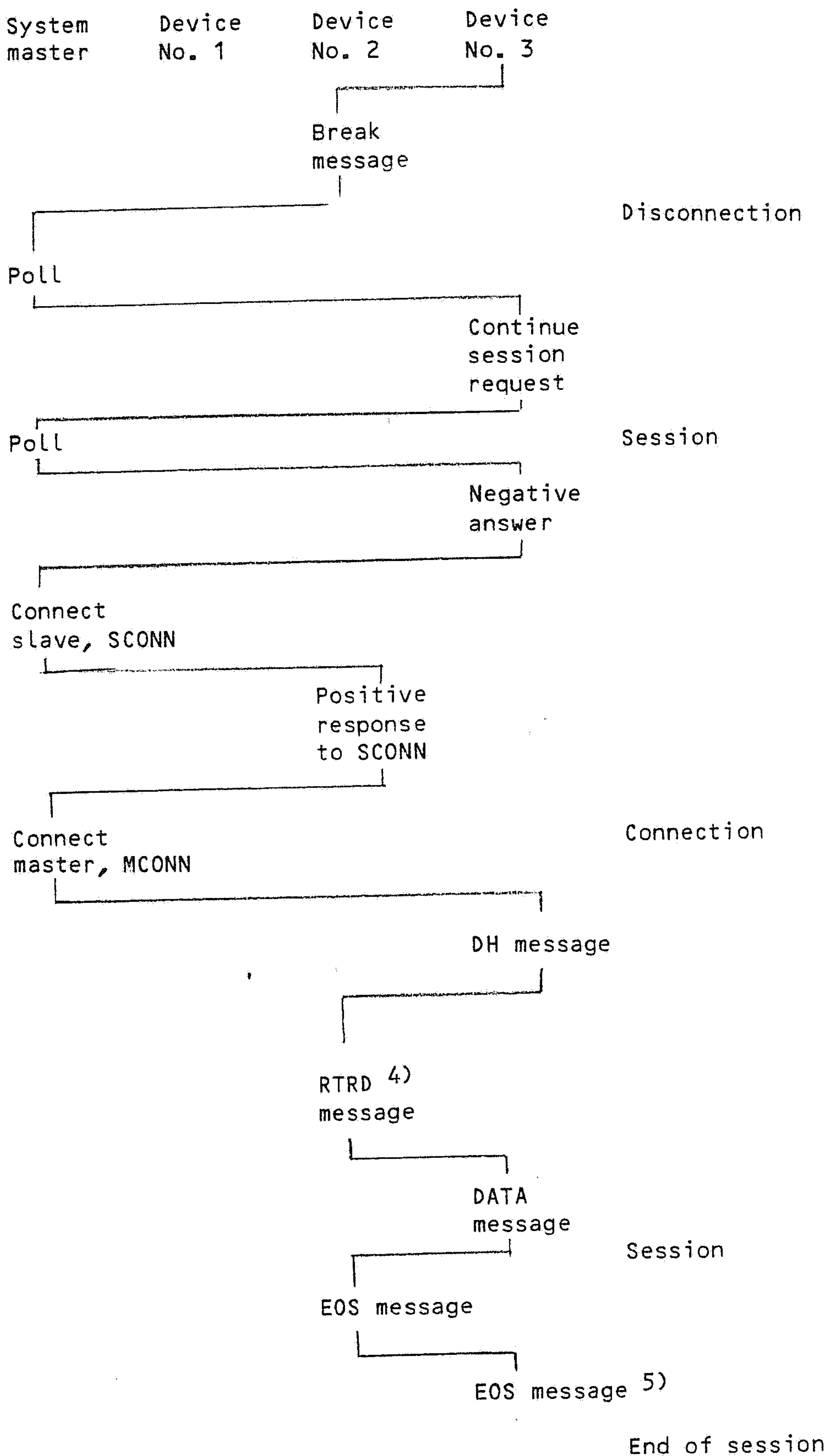
7.5.1 General Communication Example

The following example shows a typical transmission sequence between device 2 and device 3.

The sequence is described in detail below the figures.



(Cont.)


System master
Device No. 1
Device No. 2
Device No. 3
Break message
Disconnection
Poll
Continue session request
Session
Poll
Negative answer
Connect slave, SCONN
Positive response to SCONN
Connection
Connect master, MCONN
DH message
RTRD 4) message
DATA message
Session
EOS message
EOS message 5)
End of session

1) If the device cannot handle the request immediately, information about that is included in the status byte.

2) Other messages that can be sent to the slave in this situation are:

   - DH message
   - INFO DH message and to the system master
   - Abort message

3) The DH message can be sent directly from slave to session master instead of the sequence started with the BREAK message. It depends on the slave status.

4) All other answers than RTRD mean negative acknowledgement.

5) If a data message is not acknowledged the session master sends DENQ message to the slave.

The system master polls first device 1. Device 1 has nothing to send so a negative answer is sent to the system master.

The system master continues with a poll message to device 2. Device 2 wants data from device 3. Therefore device 2 sends the Open session request answer to system master. The answer includes the address of device 3 (slave address). The system master polls device 2 until the device 2 answer is negative.

The system master has now to establish a session with the two devices 2 and 3 involved. Device 2 is session master. The device that has made the session request is always given the role of session master.

The system master sends the SCONN message to device 3. Device 3 sends a positive response to SCONN, which means that device 3 takes the role of slave within the session. The system master sends the MCONN message to device 2. Device 2 takes the session master role. Now the session is opened.

Device 2, the session master, sends a short data message to device 3. In this example the message includes information on which data device 3 should send to device 2. If data not is immediately available device 3 sends a BREAK for disconnection.

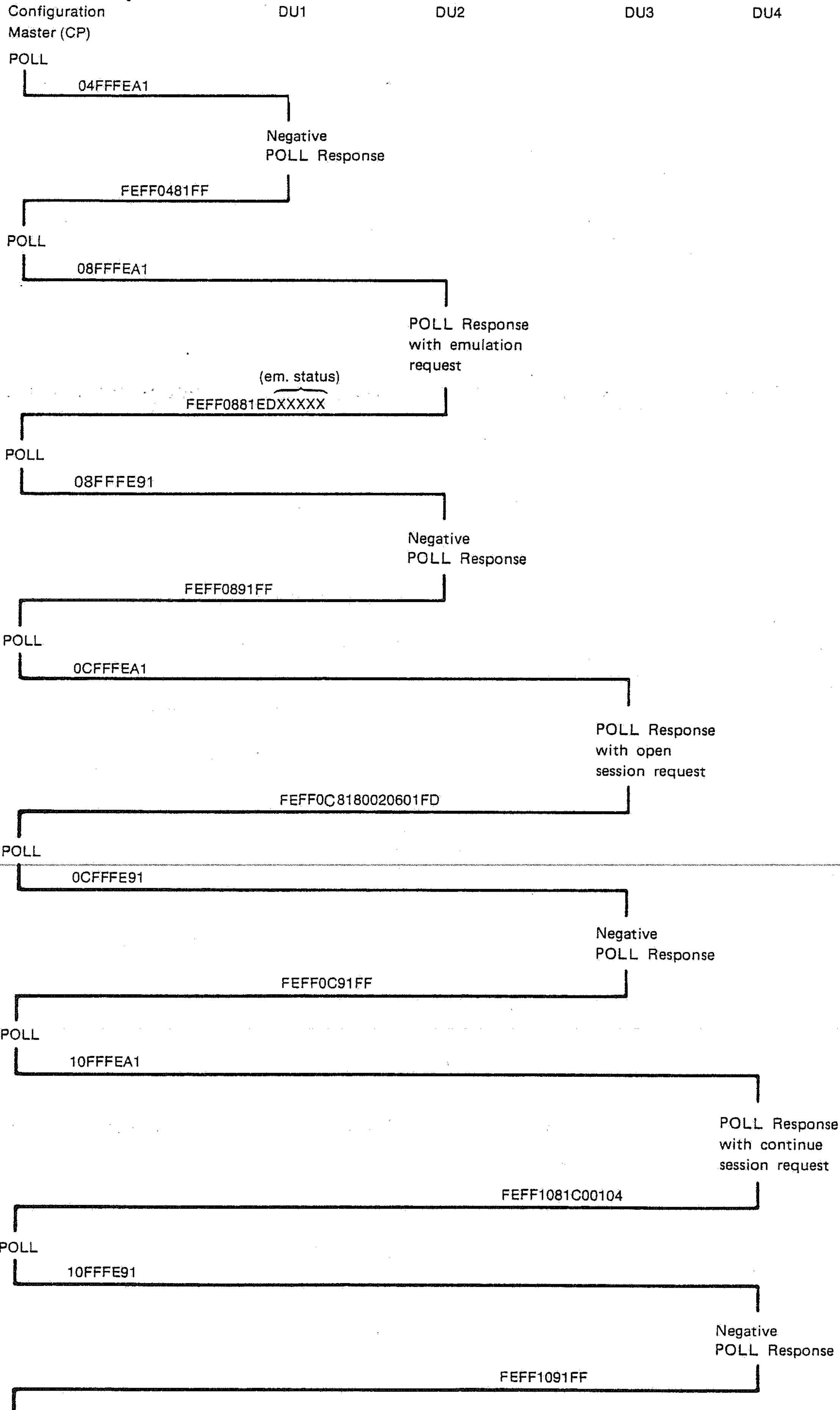
After that device 3 has prepared its data message to device 2, device 3 answers with a continue session request when it is polled.

The system master then connects device 2, session slave, to device 3, session master. If the data message is a long data message, which means that the receiver buffer is too small, a data header, DH, message must precede the data message in order to let the session slave define the receiver buffer to be used.

7.5.2 Poll and Answer to Poll

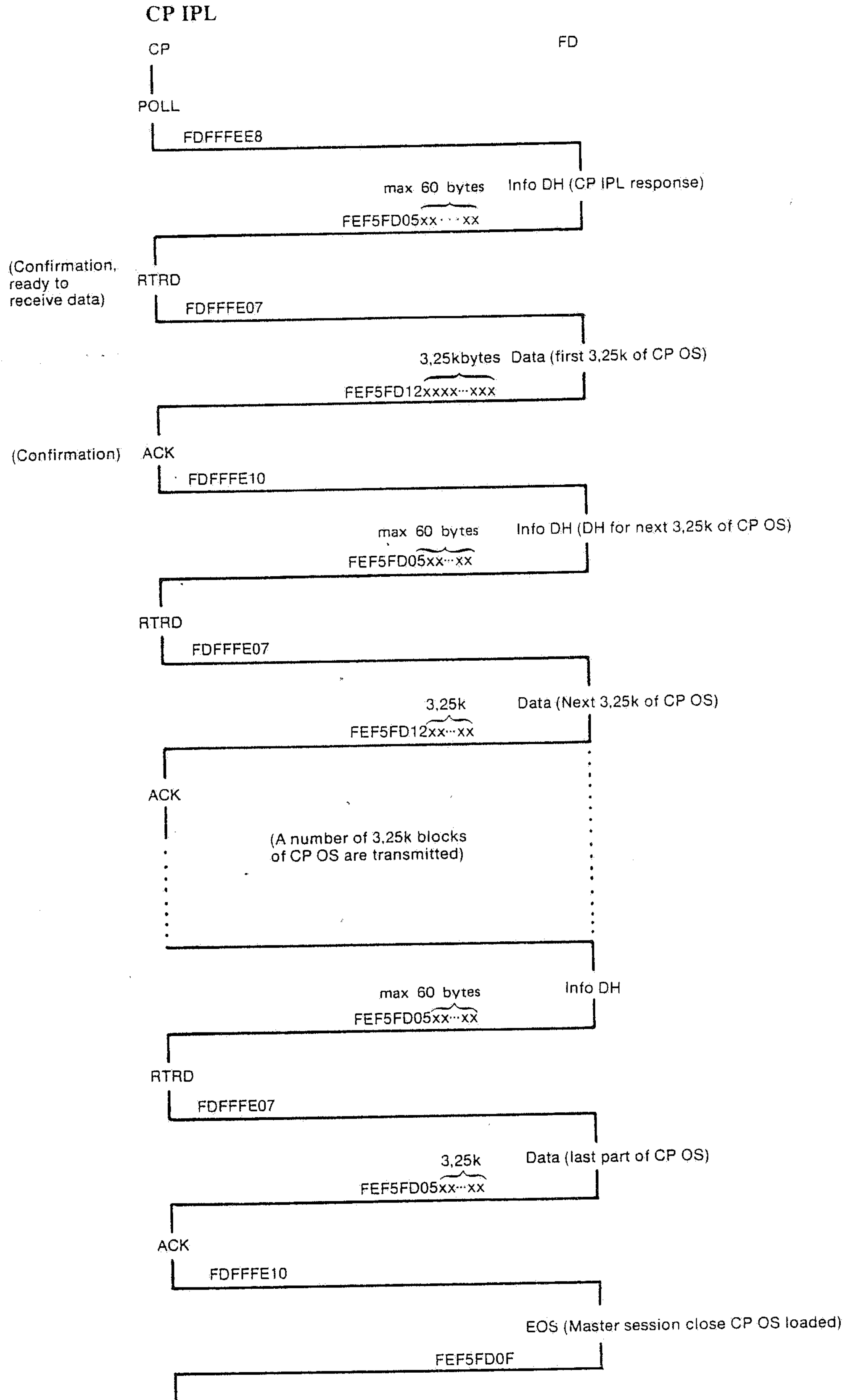
In the example below, 4 display units in a cluster configuration are polled. The following assumptions are made:

CP: Host communication possible on host line 1.
DU1: has got nothing to transmit and gives a negativ poll answer. The emulation is logged on.
DU2: wants to communicate with the host on host line 1.
DU3: wants to communicate with the system FD and therefore initiates an open session request. The master session address is assumed = 1.
DU4: wants to continue a print session. SMSA is assumed = 1.


Configuration Master (CP)
DU1
DU2
DU3
DU4
POLL
04FFFEA1
Negative POLL Response
FEFF0481FF
POLL
08FFFEA1
POLL Response with emulation request
(em. status)
FEFF0881EDXXXXX
POLL
08FFFE91
Negative POLL Response
FEFF0891FF
POLL
0CFFFEA1
POLL Response with open session request
FEFF0C8180020601FD
POLL
0CFFFE91
Negative POLL Response
FEFF0C91FF
POLL
10FFFEA1
POLL Response with continue session request
FEFF1081C00104
POLL
10FFFE91
Negative POLL Response
FEFF1091FF

7.5.3 CP IPL Session

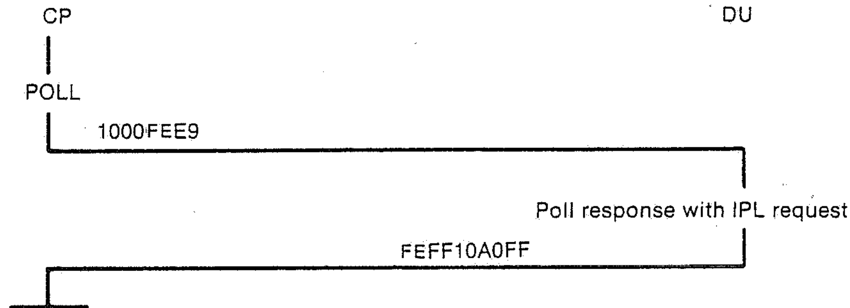
The following example shows how CP requests IPL and gets OS loaded from the system FD. At CP IPL, CP and FD have the initial addresses FE and FD.



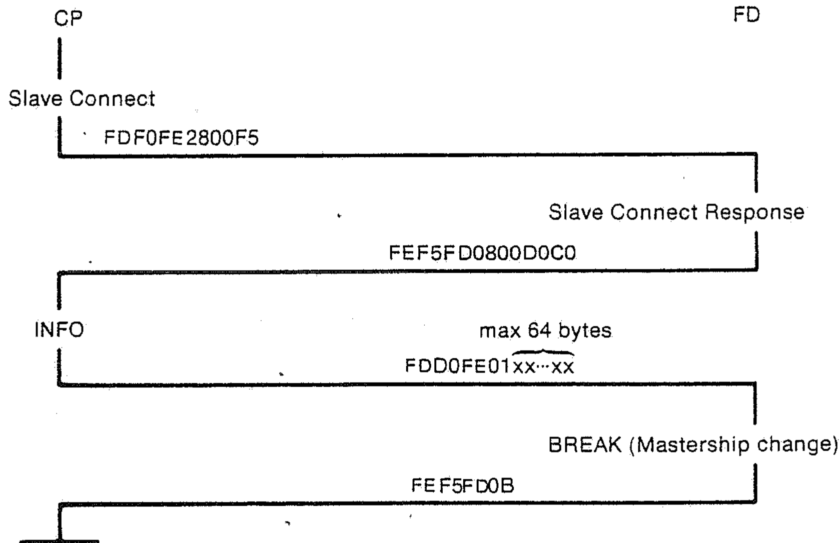
7.5.4 DU IPL Session

DU IPL

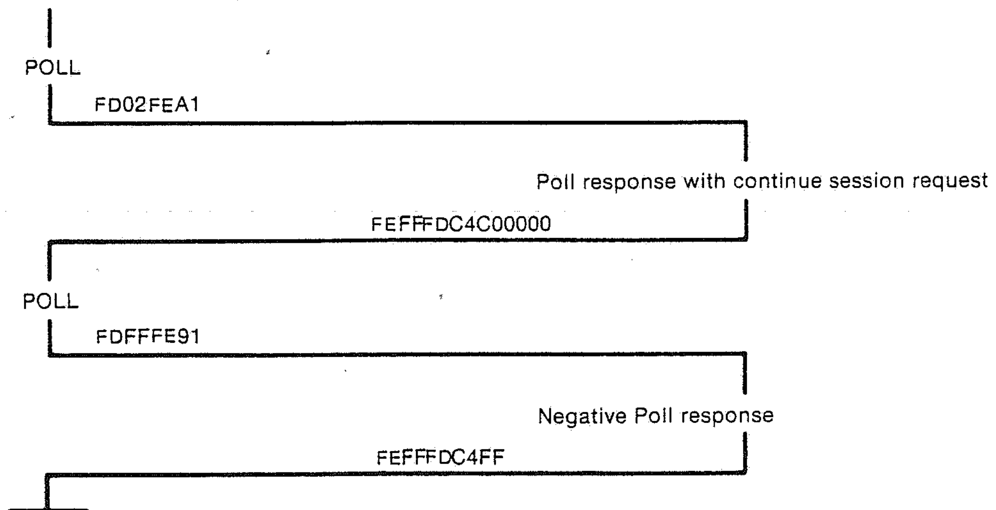
Poll of DU

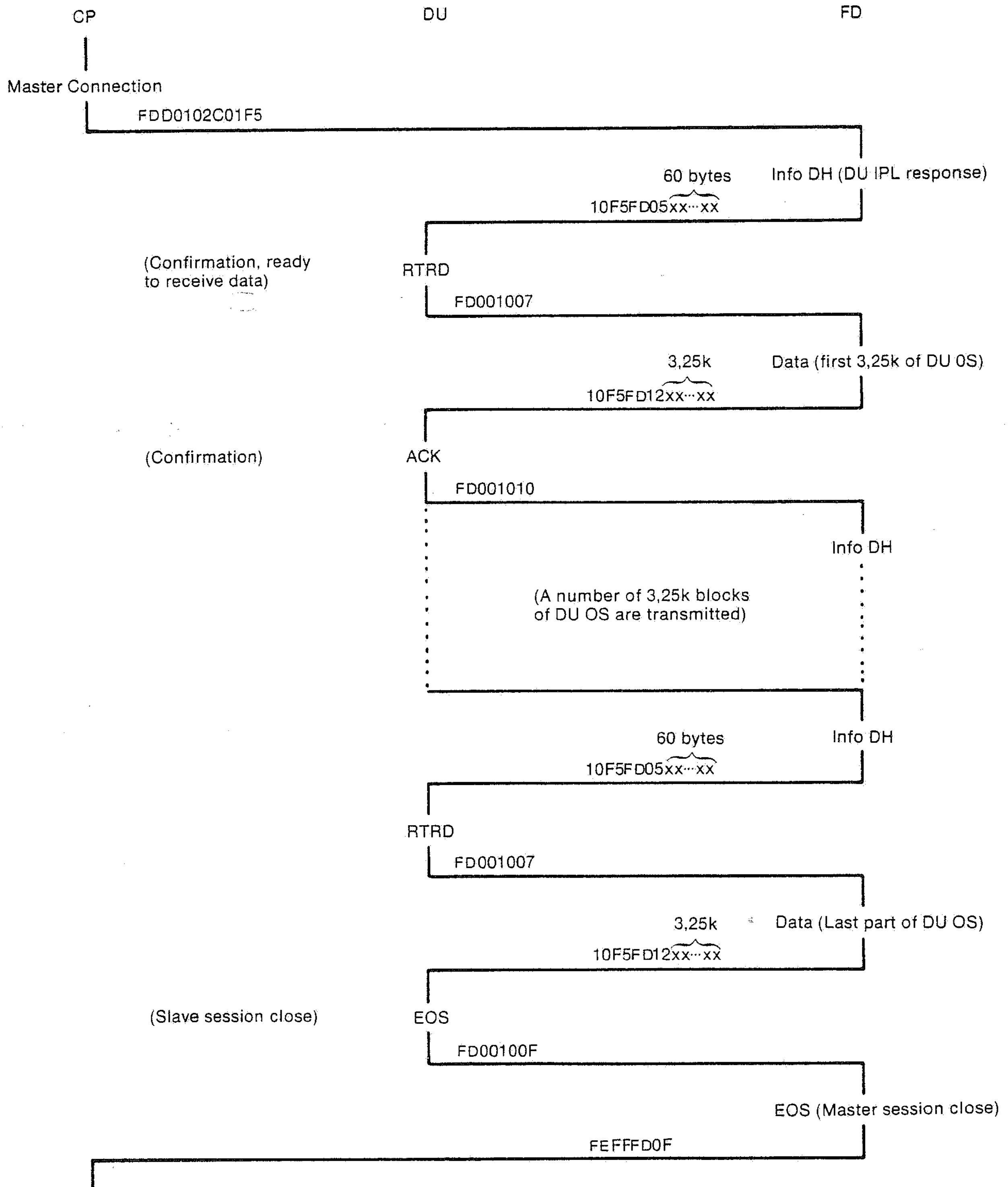


DU IPL session initiation



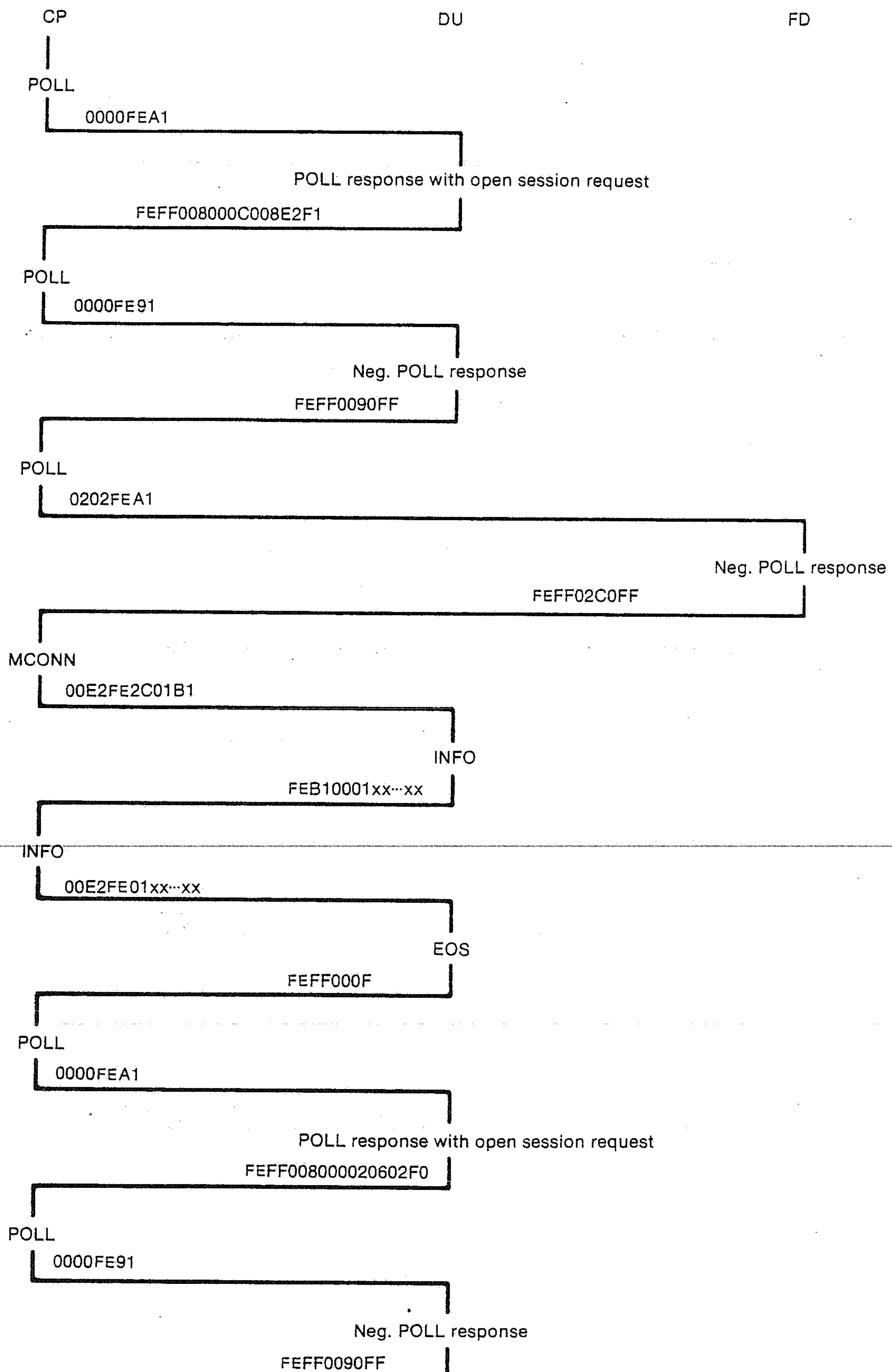
DU IPL session continue request

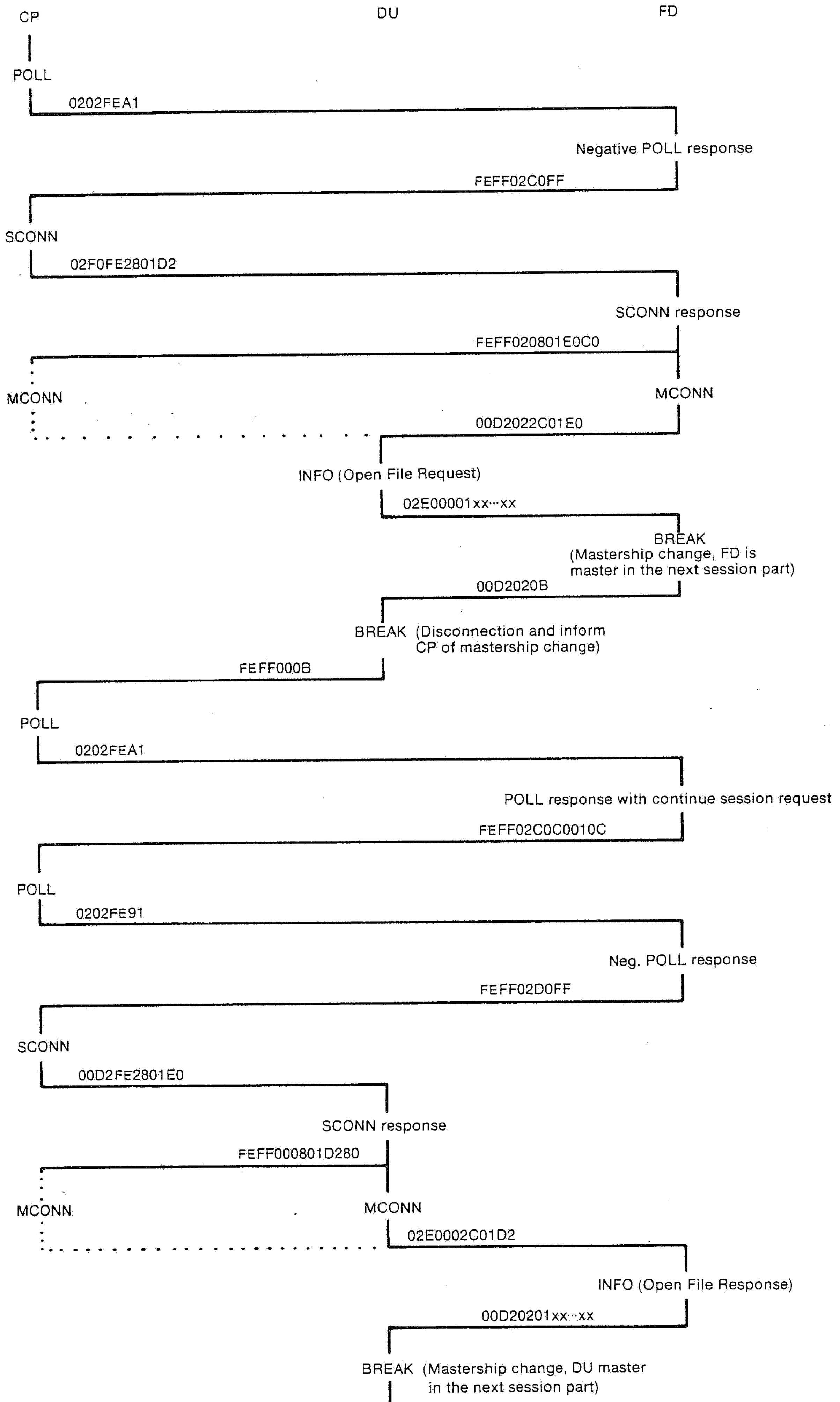


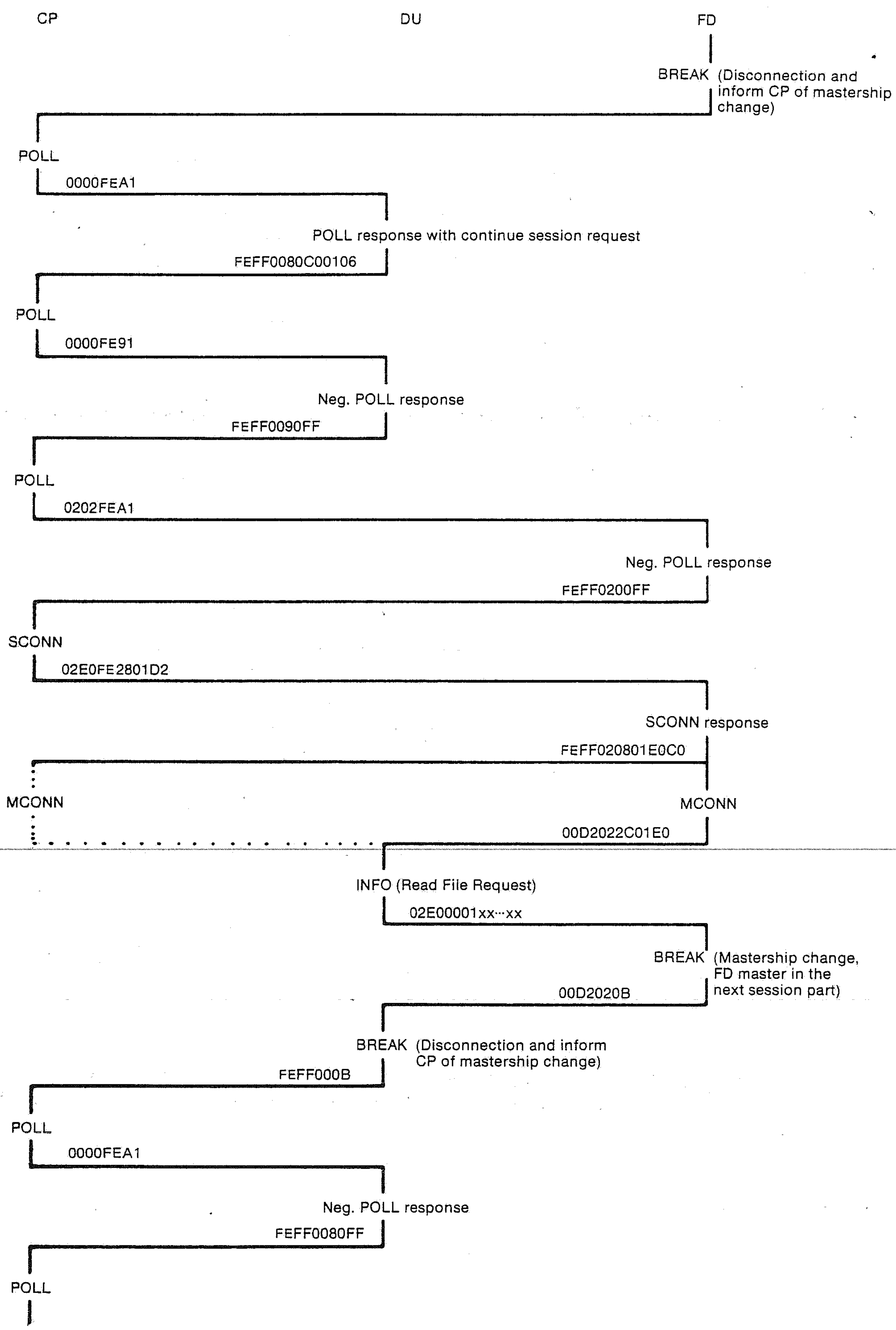

CP
DU
FD
Master Connection
FDD0102C01F5
60 bytes
Info DH (DU IPL response)
10F5FD05xx···xx
(Confirmation, ready to receive data)
RTRD
FD001007
3,25k
Data (first 3,25k of DU OS)
10F5FD12xx···xx
(Confirmation)
ACK
FD001010
Info DH
(A number of 3,25k blocks of DU OS are transmitted)
60 bytes
Info DH
10F5FD05xx···xx
RTRD
FD001007
3,25k
Data (Last part of DU OS)
10F5FD12xx···xx
(Slave session close)
EOS
FD00100F
EOS (Master session close)
FEFFFD0F

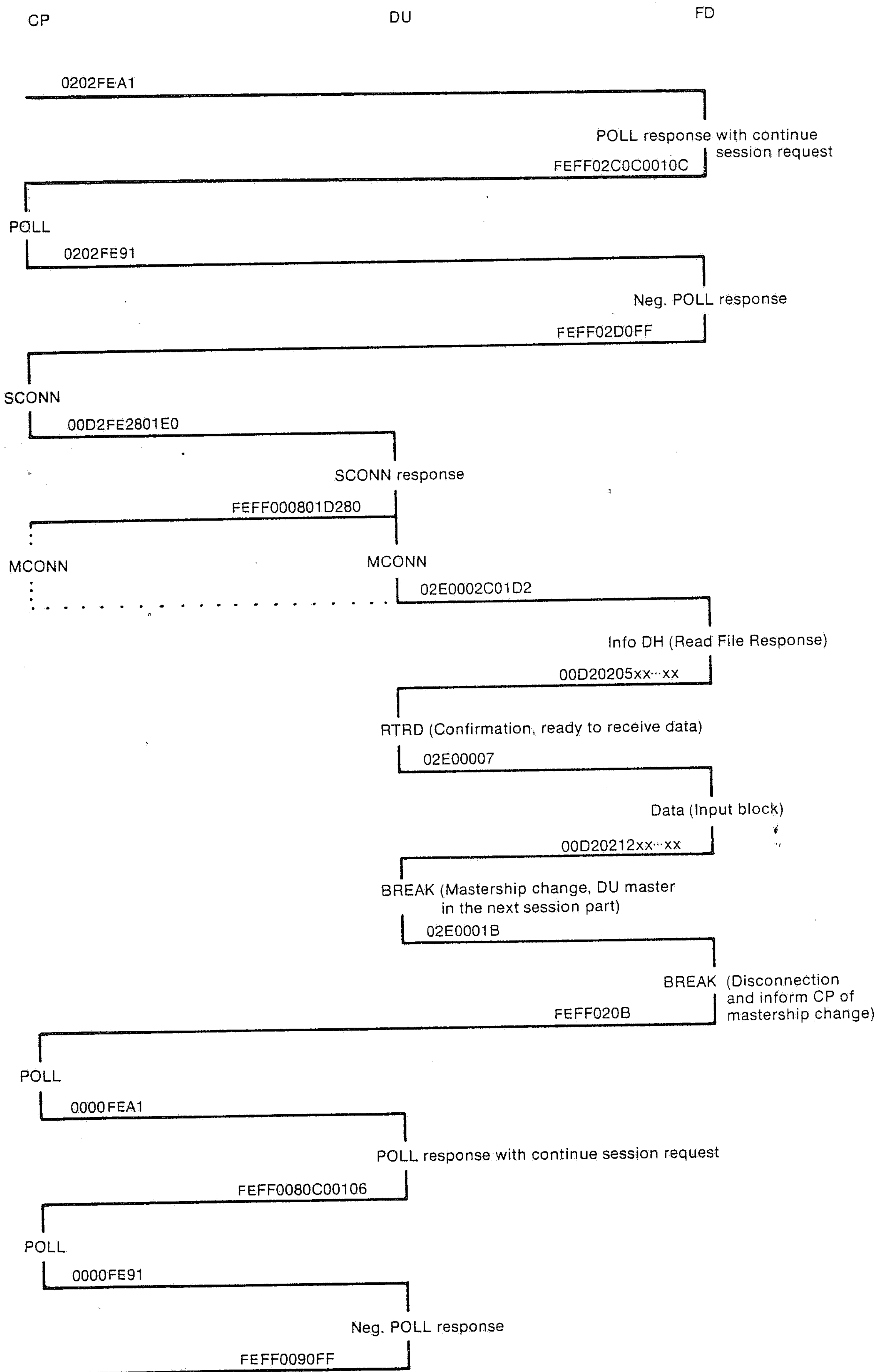
7.5.5 File Update Session

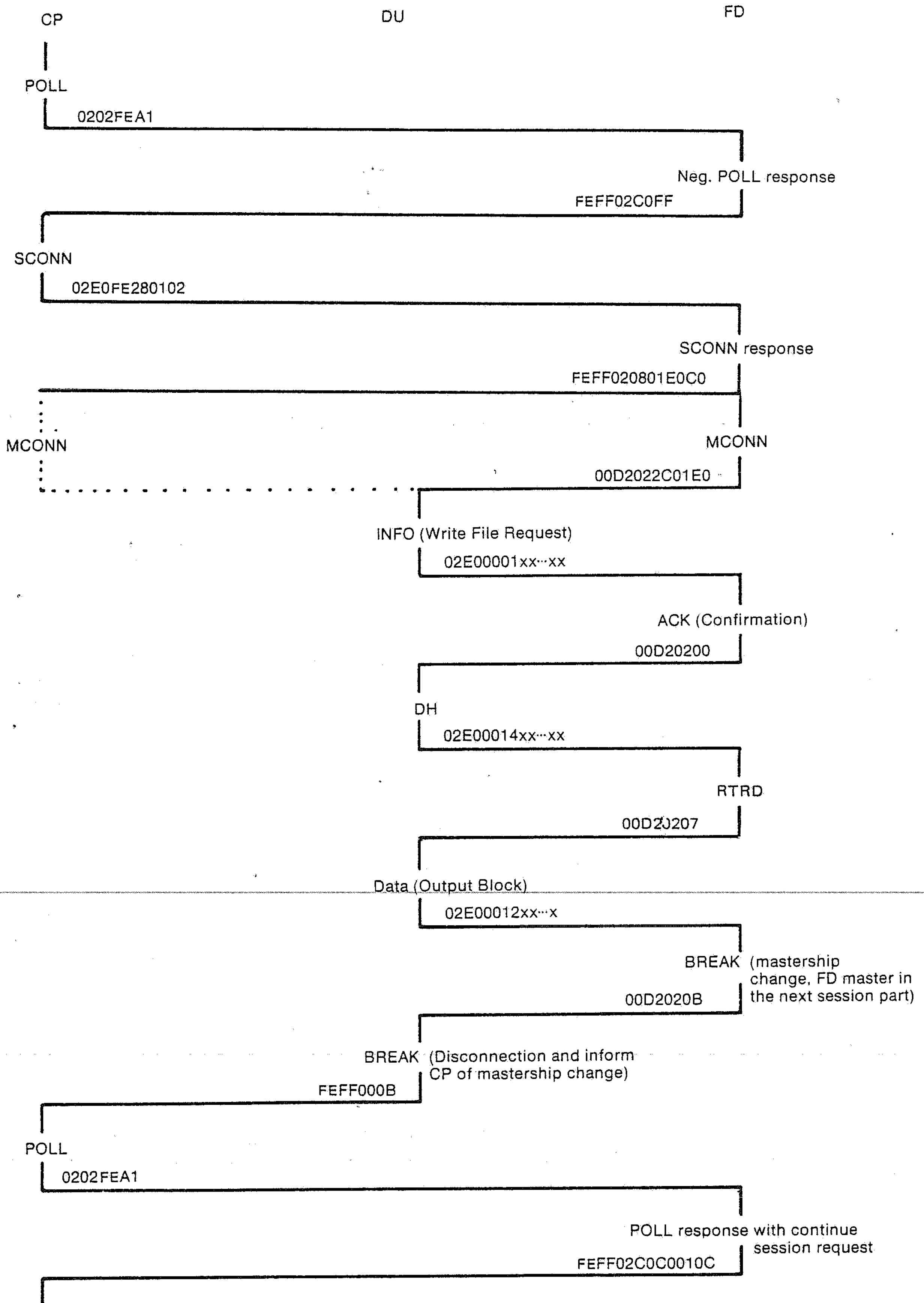
This example shows how DU No. 0 reads a block from a data-FD and then writes it back. All poll sequences are not shown in this example.

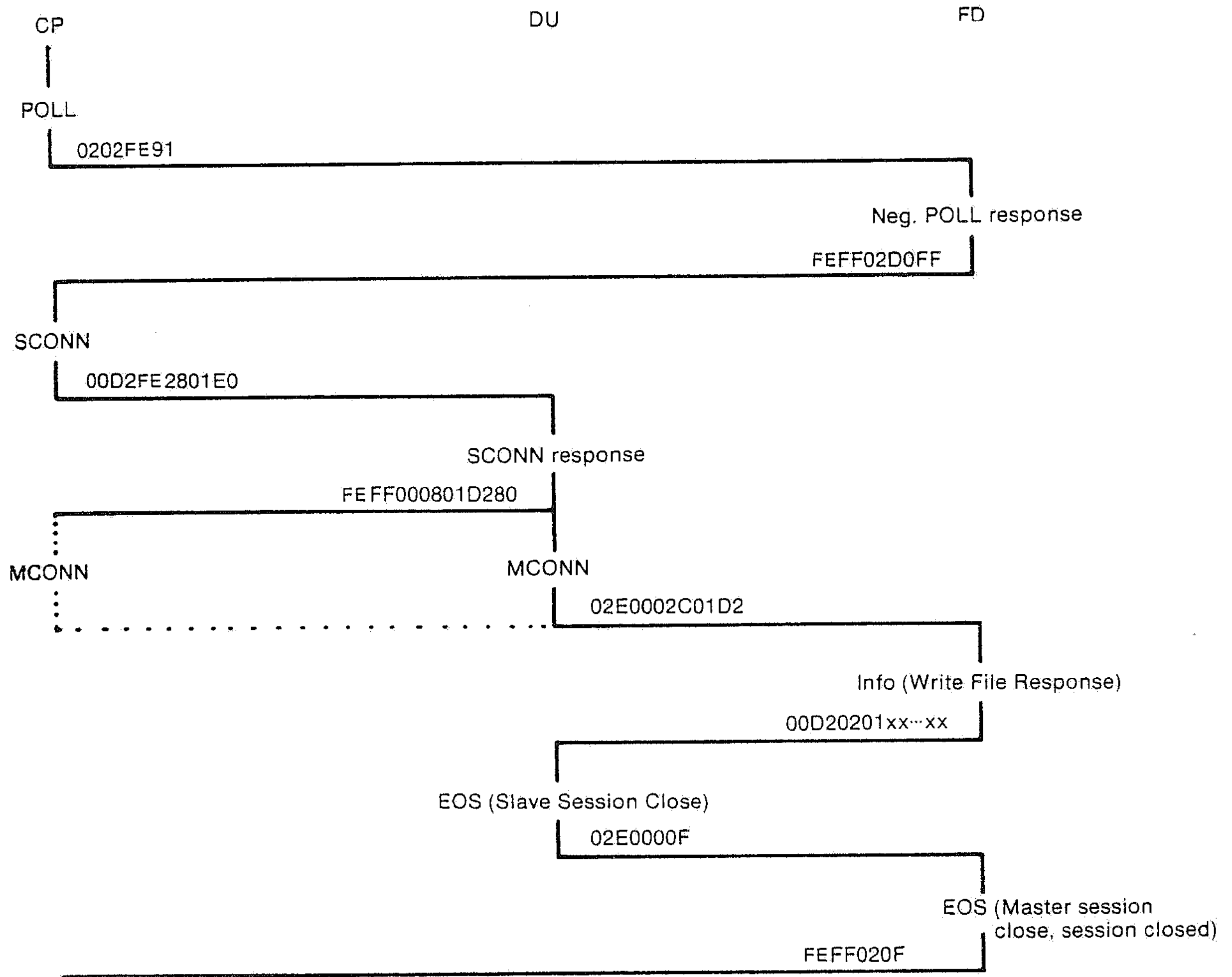



CP
DU
FD
POLL
0202FEA1
Negative POLL response
FEFF02C0FF
SCONN
02F0FE2801D2
SCONN response
FEFF020801E0C0
MCONN
MCONN
00D2022C01E0
INFO (Open File Request)
02E00001xx···xx
BREAK
(Mastership change, FD is
master in the next session part)
00D2020B
BREAK (Disconnection and inform
CP of mastership change)
FEFF000B
POLL
0202FEA1
POLL response with continue session request
FEFF02C0C0010C
POLL
0202FE91
Neg. POLL response
FEFF02D0FF
SCONN
00D2FE2801E0
SCONN response
FEFF000801D280
MCONN
MCONN
02E0002C01D2
INFO (Open File Response)
00D20201xx···xx
BREAK (Mastership change, DU master
in the next session part)


CP
DU
FD
BREAK (Disconnection and inform CP of mastership change)
POLL
0000FEA1
POLL response with continue session request
FEFF0080C00106
POLL
0000FE91
Neg. POLL response
FEFF0090FF
POLL
0202FEA1
Neg. POLL response
FEFF0200FF
SCONN
02E0FE2801D2
SCONN response
FEFF020801E0C0
MCONN
MCONN
00D2022C01E0
INFO (Read File Request)
02E00001xx···xx
BREAK (Mastership change, FD master in the next session part)
00D2020B
BREAK (Disconnection and inform CP of mastership change)
FEFF000B
POLL
0000FEA1
Neg. POLL response
FEFF0080FF
POLL


CP
DU
FD
0202FEA1
POLL response with continue session request
FEFF02C0C0010C
POLL
0202FE91
Neg. POLL response
FEFF02D0FF
SCONN
00D2FE2801E0
SCONN response
FEFF000801D280
MCONN
MCONN
02E0002C01D2
Info DH (Read File Response)
00D20205xx···xx
RTRD (Confirmation, ready to receive data)
02E00007
Data (Input block)
00D20212xx···xx
BREAK (Mastership change, DU master in the next session part)
02E0001B
BREAK (Disconnection and inform CP of mastership change)
FEFF020B
POLL
0000FEA1
POLL response with continue session request
FEFF0080C00106
POLL
0000FE91
Neg. POLL response
FEFF0090FF

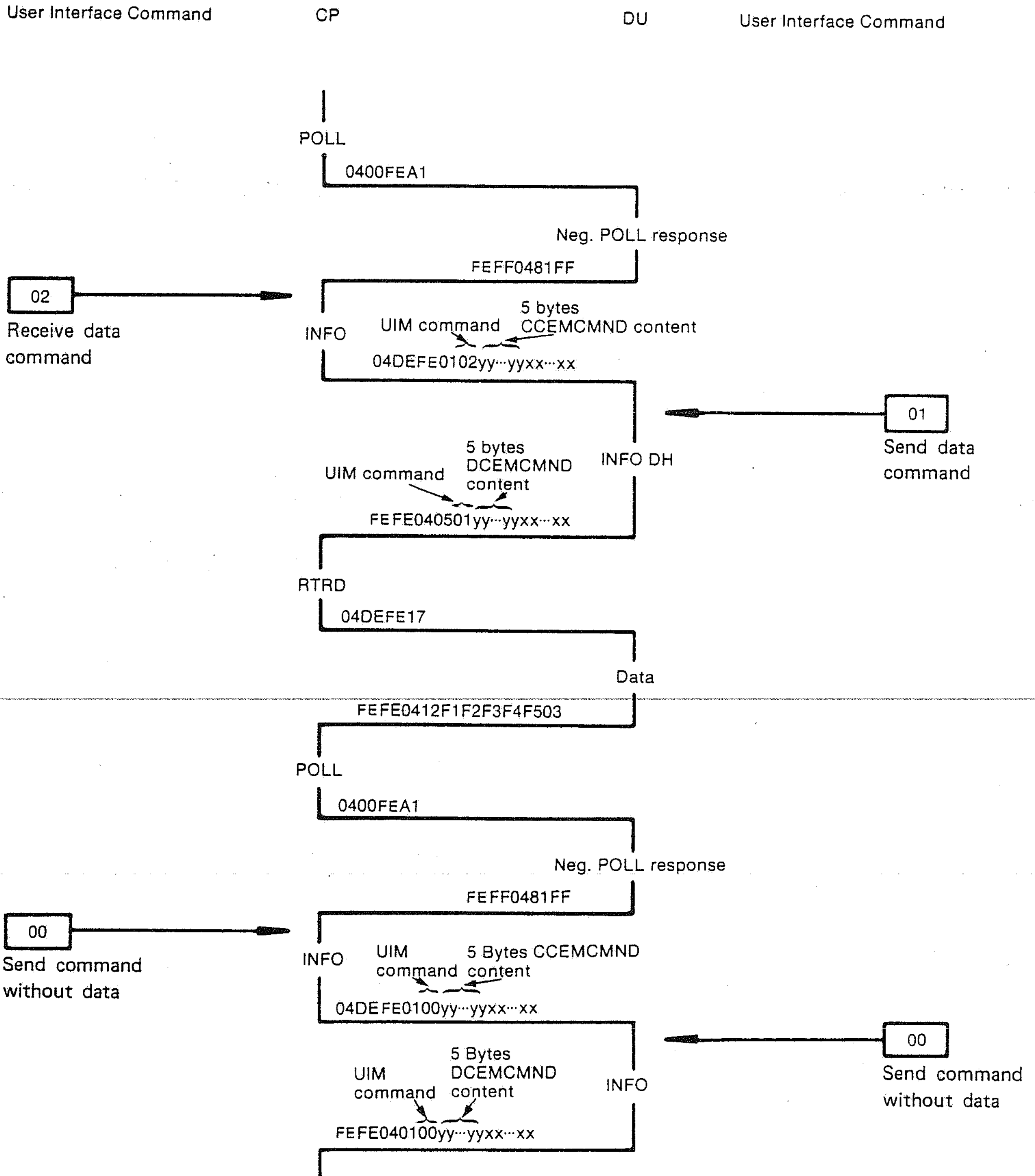

CP
DU
FD
POLL
0202FEA1
Neg. POLL response
FEFF02C0FF
SCONN
02E0FE280102
SCONN response
FEFF020801E0C0
MCONN
MCONN
00D2022C01E0
INFO (Write File Request)
02E00001xx···xx
ACK (Confirmation)
00D20200
DH
02E00014xx···xx
RTRD
00D20207
Data (Output Block)
02E00012xx···x
BREAK (mastership change, FD master in the next session part)
00D2020B
BREAK (Disconnection and inform CP of mastership change)
FEFF000B
POLL
0202FEA1
POLL response with continue session request
FEFF02C0C0010C


CP
DU
FD
POLL
0202FE91
Neg. POLL response
FEFF02D0FF
SCONN
00D2FE2801E0
SCONN response
FEFF000801D280
MCONN
MCONN
02E0002C01D2
Info (Write File Response)
00D20201xx···xx
EOS (Slave Session Close)
02E0000F
EOS (Master session
close, session closed)
FEFF020F

7.5.6 User Interface Sequence

This example shows how a Read DU command in the User Interface is implemented in the two-wire protocol. The CP reads the characters 1, 2, 3, 4 and 5 from DU. After the READ DU sequence, an exchange of System Module Command areas is performed. The User Interface is further described in section 8.

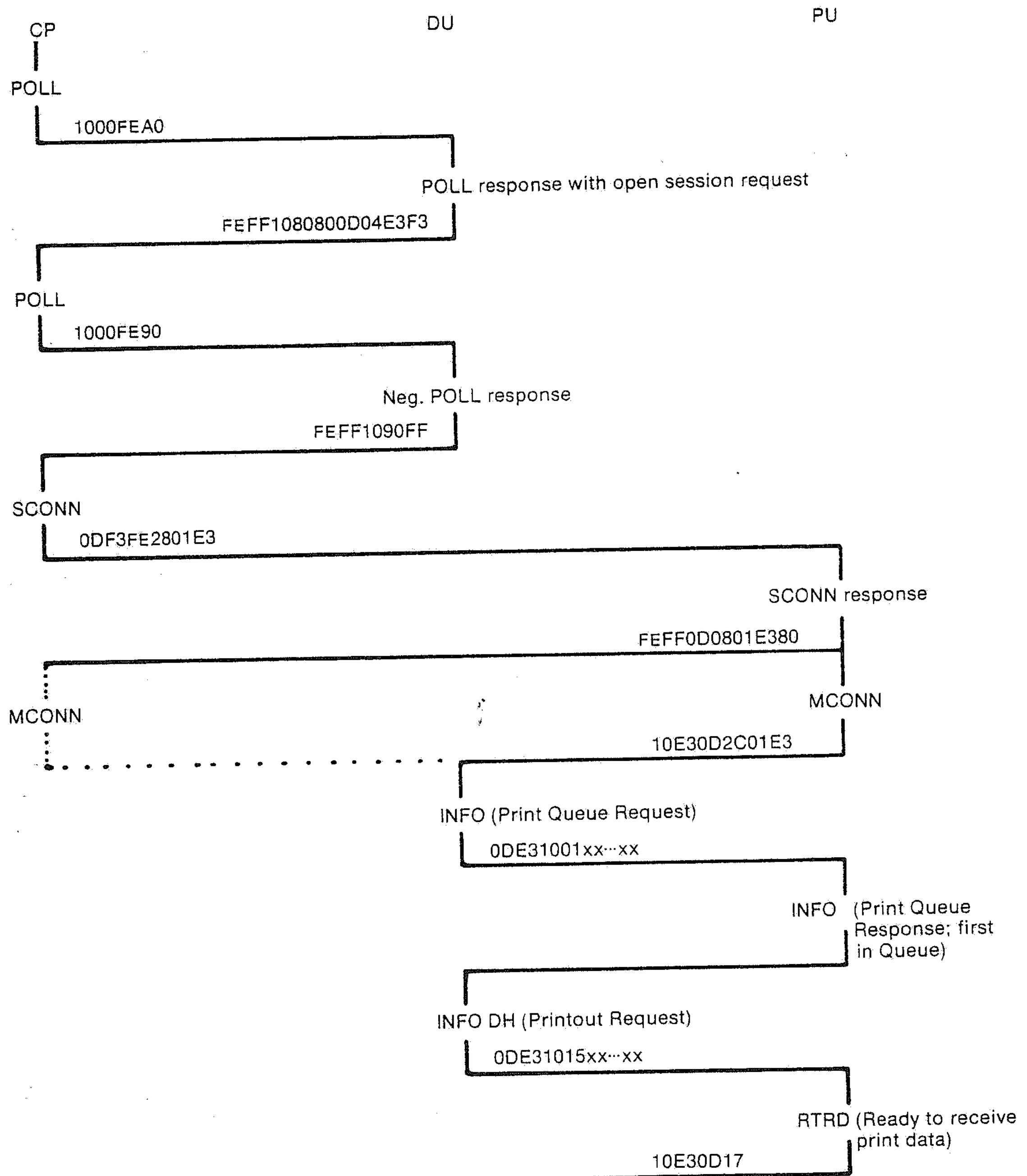
Note that the connect/disconnect procedure is reduced in a User Interface sequence.

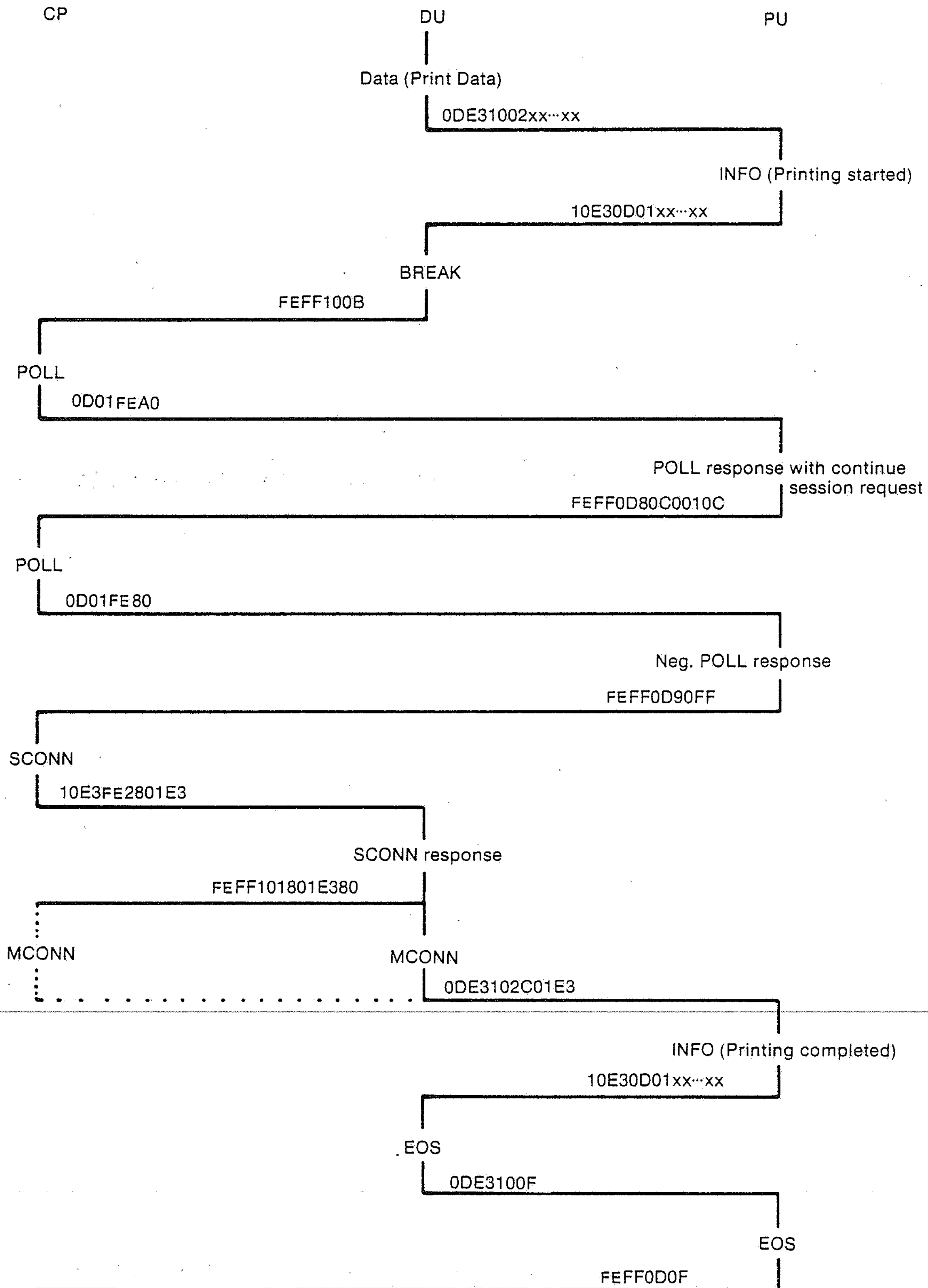


7.5.7 Printer Communication Sequence

The following example shows the communication between a DU without a printer and a DU with printer.

Printer handling is further described in section on Printer Unit Functions.




CP
DU
PU
Data (Print Data)
0DE31002xx···xx
INFO (Printing started)
10E30D01xx···xx
BREAK
FEFF100B
POLL
0D01FEA0
POLL response with continue session request
FEFF0D80C0010C
POLL
0D01FE80
Neg. POLL response
FEFF0D90FF
SCONN
10E3FE2801E3
SCONN response
FEFF101801E380
MCONN
MCONN
0DE3102C01E3
INFO (Printing completed)
10E30D01xx···xx
EOS
0DE3100F
EOS
FEFF0D0F

8 USER INTERFACE FUNCTIONS

List of Contents

8.1	GENERAL	3
8.2	USER INTERFACE MODULE	5
8.3	USER INTERFACE CONTROL BLOCK	6
8.4	INITIALIZATION OF UIM	8
8.4.1	Functional Description	8
8.4.2	Interface in DU	8
8.5	CONNECT/DISCONNECT	9
8.5.1	Functional Description	9
8.5.2	Connect Interface in CP	9
8.5.3	Disconnect Interface in CP	10
8.6	SEND COMMAND WITHOUT DATA	11
8.6.1	Functional Description	11
8.6.2	Interface in CP	11
8.6.3	Interface in DU	11
8.7	WRITE DU	12
8.7.1	Functional Description	12
8.7.2	Interface in CP	13
8.7.3	Interface in DU	13
8.7.4	WRITE DU with Emulation Control	14
8.8	READ DU	15
8.8.1	Functional Description	15
8.8.2	Interface in CP	16
8.8.3	Interface in DU	16
8.9	EMULATION STATUS UPDATING	17
8.9.1	Functional Description	17
8.9.2	Interface in CP	17
8.9.3	Interface in DU/PU	18
8.10	EMULATION ABORT	20
8.10.1	Functional Description	20
8.10.2	Interface in CP	20
8.11	HOST SYSTEM STATUS	21
8.11.1	Functional Description	21
8.11.2	Interface in CP	21
8.11.3	Interface in DU/PU	22

8.12	APPLICATION EXAMPLES	23
8.12.1	Initialization	23
8.12.2	Send Request from DU	23
8.12.3	Poll from Host Computer	24
8.12.4	Ack from Host Computer	25
8.12.5	Text from Host Computer	26
8.12.6	Session Concluded by Host Computer	26-27

8.1 GENERAL

In a cluster configuration system, the system modules must be able to communicate with each other via the two-wire connection.

In order to relieve the system modules from the necessity of handling intercommunication line procedures, a User Interface module (UIM) is provided in CP and in DU/PU. The User Interface module utilizes the Communication Handler.

The User Interface module provides the possibility of transmitting data and commands within the cluster without having to care about the details of the internal communication protocol. Communication can thus be performed by means of the UIM and a protocol defined by the emulation.

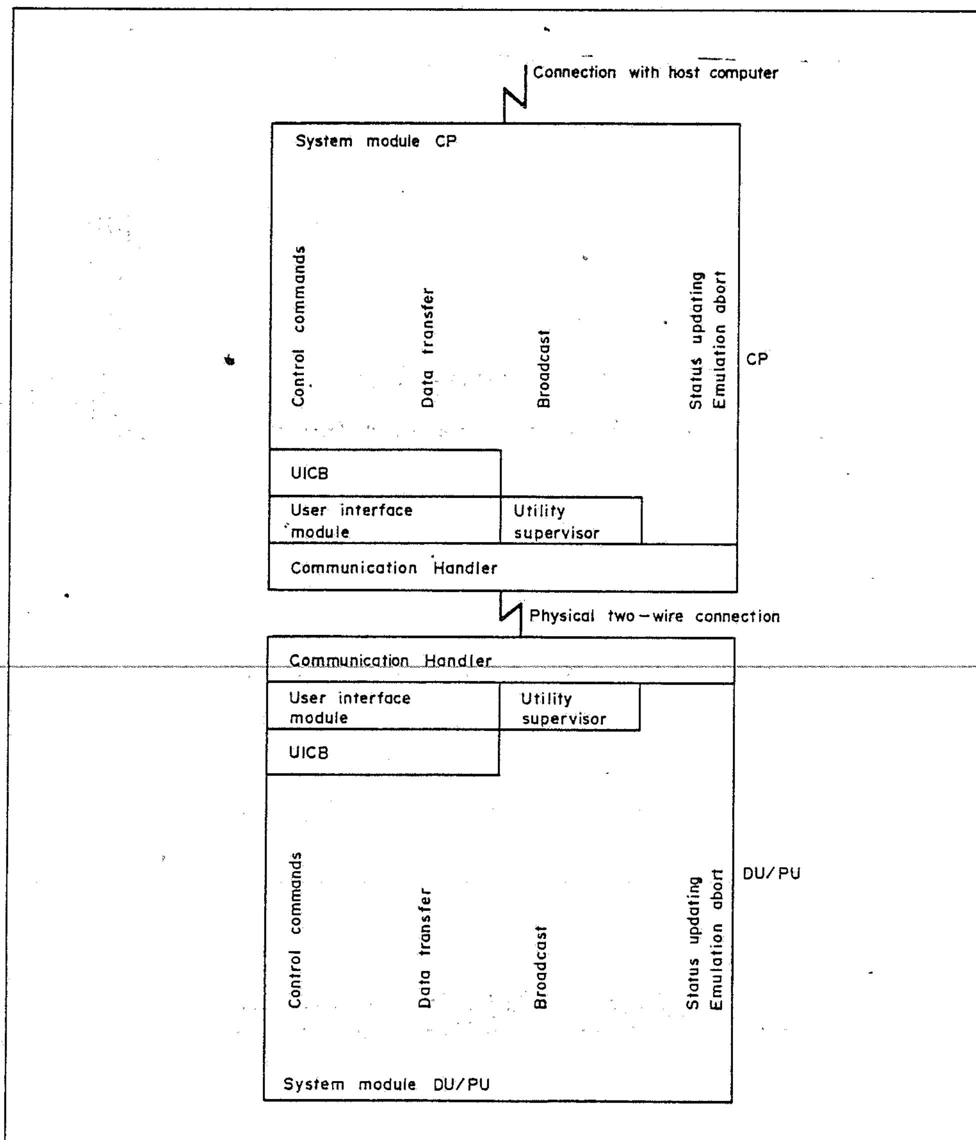


Fig. 8.1 User Interface modules in cluster configuration

The system modules in CP can also acquire fast updating of the status of the other terminal units directly via the Communication Handler. Information about aborted units is received directly via the Communication Handler.

8.2 USER INTERFACE MODULE

The User Interface Module is a procedure which carries out the requested communication sequences within the cluster.

A communication session is always initiated by CP, which connects other system units. The other units (DU/PU) cannot initiate any transmission. They can only raise a connection request indication in the poll response.

The User Interface Module in one unit communicates, via the Communication Handler, with the corresponding User Interface Module in another unit.

UIM in CP is called BCC_08000 and
UIM in DU and PCU is called BCG_08000.

The UIM carries out various communication functions. The desired function is specified in the User Interface Control Block (UICB).

UIM communication is carried out on a special communication channel. See section 7.1.2.

After a call to UIM, the calling unit's UIM is in waiting mode until a response is received.

All UIM execution is carried out under LOCK(INTERRUPT).

8.3 USER INTERFACE CONTROL BLOCK

Communication between the system module and the User Interface Module is controlled by a control block, UICB. In this control block, the system module specifies the function which is to be executed. After the execution, the results of the operation can be obtained from the same control block.

UICB consists of a structure named BCC_70107 in CP and BCG_70107 in DU. The structure is available both before and after a call to the User Interface module.

Declaration of the User Interface Control Block:

```
DCL 1 BCx_70107 EXTERNAL,          /* x = C in CP and x = G in DU */
      5 yCUICMND   BYTE,           /* y = C in CP and y = D in DU */
      5 yCEMCMND   CHAR(5)
      5 yCSDSTRT   POINTER,
      5 yCSDLEN    BINARY(15)
      5 yCRDSTRT   POINTER,
      5 yCRDLEN    BINARY(15),
      5 yCDEVNR1   BYTE,
      5 yCDEVNR2   BYTE,
      5 yCDSTAT    BYTE,
      5 yCUIMSTAT  BYTE,
      5 yCBUFSTAT  BYTE,
      5 yCTIMEFAC  BYTE,
      5 CCCONNECT  BYTE,                /* In CP only */
      5 CSTAUPPR   POINTER,             /* In CP only */
      5 CRECLEN    BIN;                 /* In CP only*/
```

Parameter	Meaning (all codes are hexadecimal)
yCUICMND	Type of command to UIM
	$(20): Initialize User Interface module (only DU). UIM is initialized and put in wait mode.
	$(10): Wait (only DU). UIM sets yCUIMND=10 when a command has been executed or when a transmission error occurs. In case of transmission error the system module may call UIM in order to recover.
	$(81): Connect one DU to CP.
	$(C0): Disconnect.
	$(00): Send command without data. The command is transmitted in the YCEMCMND area. See below.
	$(01): Send data.
	$(02): Receive data.
	$(06): Check source (only from CP). Always automatic connect/disconnect.

Parameter	Meaning (all codes are hexadecimal)
yCEMCMND	Emulation command area, used by the emulation modules. This command area contains the emulation dependent codes for polls, answer to polls, acknowledgements etc. (Refer to the appropriate emulation reference manual.) The emulation command area is transmitted between the communicating units at each call to UIM.
yCSDSTRT	Start address of send-buffer area in sending unit.
yCSDLEN	Length of send-buffer area in sending unit.
yCRDSTRT	Start address of receive-buffer area in receiving unit.
yCRDLEN	Length of receive-buffer area in receiving unit.
yCDEVNR1	Logical address of the unit to which the command is issued. See section 7.2.1.
yCDEVNR2	Not used.
yCDSTAT	Error status for UICB. $(00): No error. $(80): Transmission error. $(FF): Connection error.
yCUIMSTAT	CP: Not used. DU: $(00): UIM in waiting mode $(FF): UIM busy.
yCBUFSTAT	Used only in DU. See section 8.7.4.
yCTIMEFAC	Timeout for internal communication. Always set by UIM to $(FF), 5 sec.
CCCONNECT (only CP)	Automatic connect/disconnect $(00): No connect/disconnect $(01): Automatic connect before command. $(02): Automatic disconnect after command. $(03): Combination of $(01) and $(02).
CSTAUPPR (only CP)	Pointer to emulation status updating routine. Set by emulation initialization routine. If no status updating routine exists CSTAUPPR=0.
CRECLEN (only CP)	Received data length. After a Read DU command, the length of the received data is stored in this parameter.

8.4 INITIALIZATION OF UIM

8.4.1 Functional Description

For CP there is no initialization to be done by the calling program.

For DU, the initialization command $(20) must be sent. Before the call to UIM, address and length of the receive buffer must also be set in UICB. After the call, UIM will be in waiting mode.

At transmission and connect errors, UIM can be reset to the waiting mode by means of command $(10). In some cases also the receive buffer parameters must be reset.

Note that when a command has been executed, UIM always sets the command entry in UICB to $(10). This command is only valid in DU. If the error occurred in CP, a disconnect will be automatically performed.

8.4.2 Interface in DU

Declarations:

```
DECLARE 1 BCG_70107 EXTERNAL,          /* UICB IN DU          */
.
.
.            As set forth in
.            Section 8.3
.
.
.
DECLARE BCG_08000 ENTRY;               /* UIM PROC IN DU      */
```

Call format:

```
DCUICMND = $(20);                      /* INITIALIZE UIM      */
DCRDSTR = "receive-buffer start address";
DCRDLEN = "receive-buffer length";

DUWAIT:
CALL BCG_08000;                        /* EXECUTE             */
IF DCDSTAT ] = 0                       /* ERROR STATUS OF UICB */
   THEN DO;
        /*HANDLE COMMUNICATION ERROR*/
        .
        .
        .
        DCUICMND = $(10);              /* RECOVER ERROR       */
        GOTO DUWAIT;
        END;
.
.
```

8.5 CONNECT/DISCONNECT

8.5.1 Functional Description

Connection/Disconnection can be performed either by using the command byte in UICB or automatically by using the CCCONNECT byte in UICB.

All connection/disconnection is performed in CP.

After disconnection, control is left back to the calling program. Note that disconnection should not be ordered if a communication error occurred.

8.5.2 Connect Interface in CP

Declarations:

```
DECLARE 1 BCC_70107 EXTERNAL,          /* UICB IN CP           */
.
.
.          As set forth in
.          Section 8.3
.
.
.
DECLARE BCC_08000 ENTRY;               /* UIM PROCEDURE        */
```

Call format; connection via connect command:

```
CCUICMND = $(81);                      /* CONNECT ONE DU TO CP */
CCDEVNR1 = "DU to be connected";
CALL BCC_08000;                        /* EXECUTE              */
```

Call format; automatic connection via CCCONNECT byte:

```
CCUICMND = "suitable command";
CCDEVNR1 = "DU to be connected";
CCCONNECT = $(01) (or $(03));          /* AUTOMATIC CONNECTION */
CALL BCC_08000;
```

If CCDSTAT in UICB contains $(00), the operation was successful.

8.5.3 Disconnect Interface in CP

Declarations:

```
DECLARE 1 BCC_70107 EXTERNAL,              /* UICB IN CP          */
.
.          As set forth in
.          Section 8.3
.
.

DECLARE BCC_08000 ENTRY;                    /* UICM PROCEDURE      */
```

Call format; disconnection via disconnection command

```
CCUICMND = $(C0);
CCDEVNR1 = "DU to be disconnected";
CALL BCC_08000;                             /* EXECUTE             */
```

Call format; automatic disconnection:

```
CCUICMND = "suitable command";
CCDEVNR1 = "DU to be disconnected";
CCCONNECT = $(02) (or $(03));
CALL BCC_08000;                             /* EXECUTE             */
```

8.6 SEND COMMAND WITHOUT DATA

8.6.1 Functional Description

When no data (text) is included in the internal message to be transmitted, only the emulation command area (yCEMCMND) has to be transferred. This is accomplished by setting the command byte in UICB to $(00).

The send command without data is used for acknowledgement from a unit which has received data, but has no data to transmit in response to the receiption.

8.6.2 Interface in CP

Declarations:

```
DECLARE 1 BCC_70107 EXTERNAL,          /* UICB IN CP            */
.
.
.           As set forth in
.           Section 8.3
.
.
DECLARE BCC_08000 ENTRY;               /* UIM PROCEDURE         */
```

Call format:

```
CCUICMND = $(00);                /* SEND COMMAND WITHOUT DATA   */
CCEMCMND = "suitable command";         /* COMMAND TO BE SENT    */
CCDEVNR1 = "logical device number";    /* DESTINATION           */
CALL BCC_08000;                        /* EXECUTE               */
```

8.6.3 Interface in DU

Declarations:

```
DECLARE 1 BCG_70107 EXTERNAL,          /* UICB IN DU            */
.
.
.           As set forth in
.           Section 8.3
.
.
DECLARE BCG_08000 ENTRY;               /* UIM PROCEDURE         */
```

Call format:

```
DCUICMND = $(00);                /* SEND COMMAND WITHOUT DATA   */
DCEMCMND = "suitable command";         /* COMMAND TO BE SENT    */
CALL BCG_08000;                        /* EXECUTE               */
```

8.7 WRITE DU

8.7.1 Functional Description

When data is to be sent to DU, CP issues a send data command.

Before the call to UIM, address and length of the send buffer, as well as command area and physical unit number must be set in UICB.

UIM in DU must be in wait mode to receive the send data command. UICB must contain the address and length of the receive buffer.

Note that UIM does not check if the send buffer is greater than the receive buffer length. If the receive buffer is to small, program code may be overwritten.

When the data has been received, control is left with the system module. Data transmitted is then available in the receive buffer. The system module then sets the send command without data in UICB and calls UIM to acknowledge the reception. See Fig. 8.2.

Note that the emulation command area is also transferred in a data transmission.

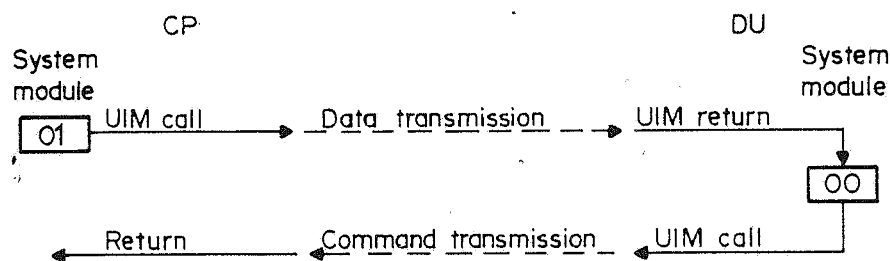


Fig. 8.2. Write DU sequence

8.7.2 Interface in CP

```
Declarations:
DECLARE 1 BCC_70107 EXTERNAL,          /* UICB IN CP          */
.
.
.            As set forth in
.            Section 8.3
.
.
DECLARE BCC_08000 ENTRY;               /* UIM PROCEDURE IN CP */

Call format:
CCUICMND = $(01);                      /* SEND DATA           */
CCSDSTRT = "send buffer address";      /* DATA TO BE SENT     */
CCSDLEN = "send buffer length";
CCEMCMND = "suitable command";         /* COMMAND TO BE SENT  */
CCDEVNR1 = "logical device number";    /* DESTINATION         */
CALL BCC_08000.                        /* EXECUTE             */
```

8.7.3 Interface in DU

```
Declarations:
DECLARE 1 BCG_70107 EXTERNAL,          /* UICB IN DU          */
.
.
.            As set forth in
.            Section 8.3
.
.
DECLARE BCG_08000 ENTRY;               /* UIM PROCEDURE IN DU */

Call format:
/*PREVIOUS UIM CALL*/
CALL BCG_08000;
/*RETURN FROM UIM AFTER DATA HAS BEEN RECEIVED*/
/*PROCESS RECEIVED DATA*/

DCUICMND = $(00);               /* SEND COMMAND WITHOUT DATA   */
DCEMCMND = "suitable command";  /* ACKNOWLEDGEMENT FROM DU     */
CALL BCG_08000;                        /* EXECUTE             */
```

8.7.4 WRITE DU with Emulation Control

In the UICB in DU, the variable DBUFSTAT contains the status of the DU buffer. If bit 7 is set in this byte, the emulation will be invoked each time CP issues the send data command (01). At this moment, UICB is updated, and can thus be used by the emulation.

The emulation should respond by a receive data command (02) when the DU buffer is ready to receive the text.

The purpose of this procedure (short write) is to make it possible for CP to send a Write command to DU, wihtout having to check if the DU buffer is ready to receive the text.

The emulation in DU can delay the data transmission. The maximum delay time is depending on the variable CTIMEFAC, the command transmission time and the time required in DU to respond to the command. CTIMEFAC is by OS set to $(FF), which corresponds to 5 seconds.

The variable DBUFSTAT should be handeled with care. It is advisable to manipulate the byte only when the session is established and control is handed over from UIM to the emulation.

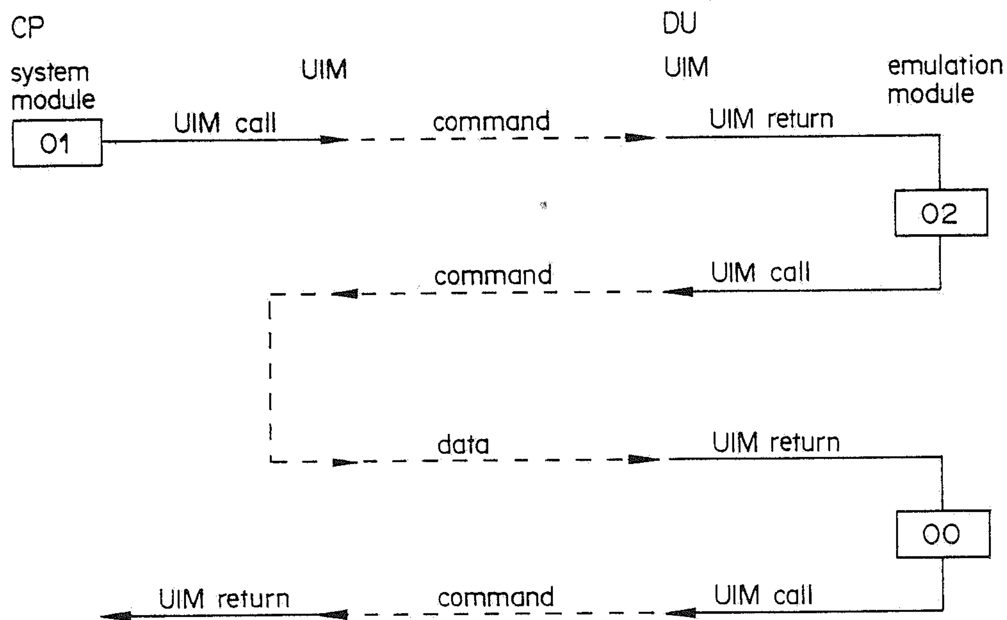


Fig. 8.3 Write DU with emulation control

8.8 READ DU

8.8.1 Functional Description

When data is to be sent from DU, the CP system module sets receive buffer address and length, physical device number, command area and CCUICMND = receive data, before UIM is called.

If DU has data to transmit, send buffer address and length has to be set before UIM is called with the send data command set. If DU has no data to transmit, it responds with send command without data.

Note that UIM does not check if the send buffer length is greater than the receive buffer length. If receive buffer is to small, program code may be overwritten. See Fig. 8.4.

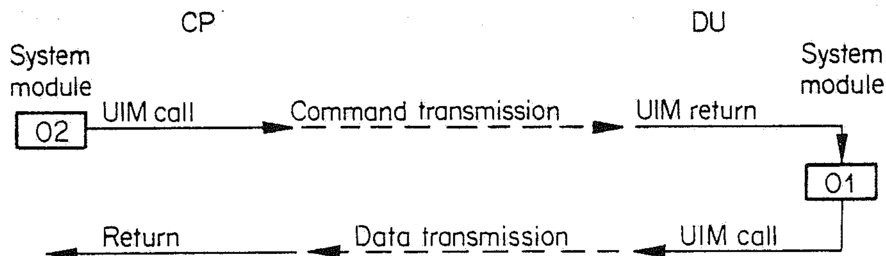


Fig. 8.4. Read DU sequence

8.8.2 Interface in CP

Declarations:

```
DECLARE 1 BCC_70107 EXTERNAL,              /* UICB IN CP            */
.
.
.          As set forth in
.          Section 8.3
.
.
DECLARE BCC_08000 ENTRY;                   /* UIM PROCEDURE in CP   */
```

Call format:

```
CCUICMND = $(02);                          /* RECEIVE DATA          */
CCEMCMND = "suitable command";
CCRDSTRT = "receive buffer address";
CCRDLEN = "receive buffer length";
CCDEVNR1 = "logical device number";        /* DESTINATION           */
CALL BCC_08000;                            /* EXECUTE               */
```

8.8.3 Interface in DU

Declarations:

```
DECLARE 1 BCG_70107 EXTERNAL,              /* UICB IN DU            */
.
.
.          As set forth in
.          Section 8.3
.
.
DECLARE BCG_08000 ENTRY;                   /* UIM PROCEDURE IN DU   */
```

Call format:

```
/*PREVIOUS UIM CALL*/
CALL BCG_08000;
/*RETURN FROM UIM*/

DCUICMND = $(01);                          /* SEND DATA             */
DCEMCMND = "suitable command";
DCSDSTRT = "send buffer address";
DCSDLEN = "send buffer length";
CALL BCG_08000;                            /* EXECUTE               */
```

8.9 EMULATION STATUS UPDATING

OS maintains a status list in CP for all physical addresses. This status list is updated in each poll response from the units.

The emulation can also define its own status list in CP, where the status of the devices is defined according to some suitable principle.

The emulation status can comprise states like "ENTER key pressed" etc.

The following describes how this emulation-defined status list can be continually updated.

8.9.1 Functional Description

When the status of any system unit is changed, the new status is saved in a status area in the unit. The external pointer in BCG_70104 is set to refer to this status area in DU/PU.

In the next internal poll response, a flag is set to indicate that the emulation status list is to be updated. The content of the event status area is transferred to a corresponding device status area in CP in connection with the poll. The external pointer BCC_70105 is set to refer to this device status area in CP.

The external event BCD_70112 indicates to DU that CP has received the new status.

When the new status has been transferred to the status area in CP, a user-written routine is invoked. The reference to this procedure is stored in the pointer CSTAUPPR in UICB. If no status updating routine is provided, CSTAUPPR = 0.

The user-written routine is executed on interrupt level.

8.9.2 Interface in CP

Declarations:

```
DECLARE CSTAUPPR PTR EXTERNAL;  /* POINTER TO USER WRITTEN PROC */
DECLARE BCC_70105 PTR EXTERNAL; /* POINTER TO DEVICE AREA       */
DECLARE 1 DEVAREA BASED BCC_70105,
            5 DEVNR BYTE,       /* PHYSICAL DEVICE NUMBER       */
            5 DEVLEN BYTE,      /* LENGTH OF DEVSTAT            */
            5 DEVSTAT(n) BYTE,  /* RECEIVED DEVICE STATUS       */
DECLARE STATUS(x) CHAR(n);      /* EMULATION STATUS TABLE       */
```

x = number of units in the configuration.

The number of bytes (n) in the status table depends on the application in question. STATUS must be assigned initial values by the user.

User written routine in CP system module:

```
STATUPD: PROCEDURE;
              .
              .
              LOCK (INTERRUPT);
              STATUS(DEVNR)= DEVSTAT;
              UNLOCK (INTERRUPT);
              .
              .
END STATUPD;
```

Call format, initialization:

```
              CSTAUPPR = ADDR(STATUPD);
```

The Communication Handler will now see to it that the emulation STATUS table is kept continuously updated.

8.9.3 Interface in DU/PU

Declarations:

```
DECLARE 1 BCG_70104 EXTERNAL, (BCP-70104 for PU)
               5 CONREQ BYTE,             /* FLAG                    */
               5 STATPTR POINTER;         /* POINTER TO STATAREA     */
DECLARE 1 STATAREA,
               5 STATCNT BYTE,         /* SIZE OF EMSTAT (BYTES)     */
               5 EMSTAT(n) CHAR;          /* EMULATION STATUS        */
DECLARE BCD_70112 EVENT EXTERNAL; (BCP_70112 for PTR) /* STATUS
                                                 RECEIVED BY CP */
```

CONREQ is a byte with the following meaning:
  Bit 7 = 1 Status transmission request
  Bit 6 = 1 Status transmission to CP initiated.

EMSTAT contains the status code which is transmitted to CP. n is the number of characters in the status code (maximum 21).

Call format, initialization:

```
STATPTR = ADDR(STATAREA);
STATCNT = "n";
```

The system module requests status to be sent to CP by setting bit 7 in CONREQ. When status transmission has been initiated, OS sets bit 6 in CONREQ. When CP has acknowledged the reception of the status, OS posts the event BCD_70112 and clears bits 6 and 7 in CONREQ.

Since the status area is also used by the poll answer routine, the status updating routine is executed under LOCK (INTERRUPT).

Call format, send new status:

```
/* LOOP START */
LOCK (INTERRUPT);
IF CONREQ = 0                                      /* NO REQUEST            */
   THEN
      GOTO PROCEED;
IF CONREQ = $(80)               /* STATUS TRANSMISSION REQUEST    */
   THEN                         /* WAS ALREADY INITIATED          */
      IF "skip old status"
            /*OLD STATUS WILL BE OVERWRITTEN*/
            THEN
              DO;
              CONREQ = 0;
              GOTO PROCEED;
              END;
/* IF CONREQ = $(80) OR $(40) */
ASSIGN BCD_70112;               For Printer: ASSIGN BCP_70112;
WAIT BCD_70112;                              WAIT BCP_70112;

PROCEED:
UNLOCK (INTERRUPT);
CONREQ = $(80)                  /* TRANSMISSION REQUEST           */
```

Transmission of the new status will be performed in connection with the next poll from CP.

8.10 EMULATION ABORT

8.10.1 Functional Description

System modules in CP may require information about whether a unit is
- logged on to the emulation
- logged off or down

Information about changes concerning these states can be handed to a user-written abort procedure in CP. The original information about the changes in device status is obtained from the TP-status list in CP OS.

The external pointer BCC_70109 is set to refer to the abort procedure. If no abort procedure is provided, BCC_70109 = 0.

The status byte BCC_70110 is interpreted by the abort procedure as follows:

Bit	7	6	5	4	3	2	1	0	Meaning
	1								Unit down or logged off
	0								Unit logged on and not down
		X	X	X	X	X	X	X	Logical device address

8.10.2 Interface in CP

Declarations:

```
DECLARE "abort proc" ENTRY;
DECLARE BCC_70109 PTR EXTERNAL;       /* PTR TO ABORT PROC        */
DECLARE BCC_70110 BYTE EXTERNAL;      /* STATUS BYTE              */
```

Call format, initialization:

```
BCC_70109 = ADDR("abort proc");
```

The abort proc will now be invoked each time bit No. 7 in BCC 70110 is changed.

Note that the polling of the cluster units is not started until the pointer BCC_70109 is initialized.

8.11 HOST SYSTEM STATUS

8.11.1 Functional Description

As mentioned in section 7.15, OS maintains a status area in each DU/PU. This status area contains the current status of the DU/PU and the status of the CP, which is updated in each received poll.

The system module can be provided with a user-written routine which will be automatically invoked each time any of bits 0-3 in the poll status byte is changed.

This means that the system module can receive and properly handle a simple ON/OFF message included in the poll.

Bits 3-0 in the status byte of the poll are interpreted as follows (see also section 7.4.1):

Bit 3	2	1	0	
1				The device has been in slow poll list
	X			This bit can be used by the emulation. (System Reset in IBM Local emulation)
		1		CP2 in dual host configuration
			1	System ready set.

8.11.2 Interface in CP

Declarations:

```
DECLARE BCC_$0910 BYTE EXTERNAL;
```

Call format, host system status ON:

```
.
.
LOCK(KERNEL);
BCC_$0910 = BCC_$0910   $(04);
UNLOCK(KERNEL);
.
.
```

Call format, host system OFF:

```
.
.
LOCK(KERNEL);
BCC_$0910 = BCC_$0910 & $(FB);
UNLOCK(KERNEL);
.
.
```

8.11.3 Interface in DU/PU

Declarations:

```
DECLARE 1 BCG_70113 EXTERNAL,        /* UNIT STATUS AREA        */
          2 CPSTAT BYTE,             /* CP STATUS IN LATEST POLL */
          2 DUSTAT BYTE,             /* DU STATUS               */
          2 PUSTAT BYTE;             /* PU STATUS               */
DECLARE OSPNTD PTR EXTERNAL;         /* POINTER TO DU - PROC    */
DECLARE OSPNTP PTR EXTERNAL;         /* POINTER TO PU - PROC    */
```

DUSTAT is transmitted from the device in each poll response. Thereby the device status list in CP is continually updated (see section 7.1.5). If the display unit has a PU connected, the PUSTAT byte is included in the poll response.

Call format, initialization:

```
OSPNTD = ADDR("DU - procedure");
OSPNTP = ADDR("PU - procedure");
```

The procedures specified in the initialization will be invoked each time any of bits 0-3 in poll status byte is changed.

8.12 APPLICATION EXAMPLES

8.12.1 Initialization

```
CP:  CSTAUPR = ADDR(STATUPD);         /* SET POINTER TO EMULATION    */
                                      /* STATUS UPDATE PROCEDURE     */

DU:  STATPTR = ADDR(STATAREA);        /* SET ADDRESS OF STATUS AREA  */
     STATCNT = N;                     /* STATUS LENGTH               */
     EMSTAT = INITSTAT;               /* AND INITIALIZE STATUS       */

     DCUICMND = $(20);                /* INITIALIZE USER INTERFACE   */
     DCRDSTRT = ADDR (RECBUF);        /* RECEIVE BUFFER ADDRESS      */
     DCRDLEN = BUFLEN;                /* RECEIVE BUFFER LENGTH       */

     DUWAIT:
     CALL BCD_08000;                  /* EXECUTE INITIALIZATION       */

     IF DCDSTAT ] = 0                 /* ERROR STATUS OF UICB IN DU   */
        THEN
           GOTO ERROR;                /* HANDLE ERROR                 */

     /*WAIT FOR COMMAND FROM CP*/

     ERROR:
     /* HANDLE COMMUNICATION ERROR */
     .
     .
     DCUICMND = $(10);                /* RECOVER ERROR                */
     GOTO DUWAIT;
```

8.12.2 Send Request from DU

```
DU:  /*STORE NEW STATUS*/
     EMSTAT = "send request"
     /*REQUEST STATUS TRANSMISSION*/

     LOCK (INTERRUPT);
     IF CONREQ = 0
       THEN
         GOTO PROCEED;
     IF CONREQ = $(80)                /* TRANSMISSION REQUEST         */
                                      /* WAS INITIATED                */
       THEN
         IF <skip old status>
           THEN
             DO;
             CONREQ = 0;
             GOTO PROCEED;
             END;
```

```
ASSIGN BCD_70112;
WAIT BCD_70112;                   /* WAIT FOR ACK FROM CP OS       */
PROCEED;
UNLOCK (INTERRUPT);
CONREQ = $(80);                   /* TRANSMISSION REQUEST          */
```

(Transmission to CP is carried out in response to the next internal poll.)

The status routine is called by the poll routine in CP:

```
CP:  STATUS (DEVNR) = DEVSTAT;
     /*PROCESS STATUS*/
```

The next poll from CP acknowledges the reception of the status message.

8.12.3 Poll from Host Computer

DU
(CP decides to fetch data from DU)

```
CP:  CCEMCMND = "poll"            /* INTERNAL COMMAND TO DU        */
     CCDEVNR1 = "DU nr";          /* DEVICE ADDRESS                */
     CCRDSTRT = ADDR(CREADBUF);   /* BUFFER ADDRESS                */
     CCRDLEN = RDBUFLEN;          /* AND LENGTH                    */
     CCCONNECT = $(03);           /* AUTOMATIC CONNECT/DISCONNECT  */
     CCUICMND = $(02);            /* RECEIVE DATA COMMAND          */
     CALL BCC_08000;              /* EXECUTE                       */
```

(Command is transmitted to DU)

```
DU:  (Return from UIM)

     IF DCDSTAT ] = 0             /* ERROR STATUS OF UICB          */
       THEN
         GOTO ERROR;

     SELECT(DCEMCMND);
     .
     .
     WHEN "poll" DO;
     /*HANDLE POLL*/
                              .
                              .
                              .
```

```
DCUICMND = $(01);              /*SEND DATA TO CP                    */
DCSDSTRT = ADDR (SENDBUF);     /*DEFINE SEND BUFFER                 */
DCSDLEN = MSGLEN;
DCEMCMND = "answer to poll";   /*COMMAND TO BE SENT                 */
CALL BCG_08000;                /*EXECUTE                            */

(Command and data is transmitted to CP.)

(Return from UIM)

CP:  IF CCDSTAT ] = 0          /*ERROR STATUS OF UICB IN CP         */
       THEN
         GOTO ERROR;
     /*DATA FROM DU IS CONTAINED IN CREADBUF*/
     .
     .
     /*CP TRANSMITS DATA TO HOST COMPUTER*/
```

8.12.4 Ack from Host Computer

```
CP:  /*SEND ACK TO DU WITH SEND COMMAND WITHOUT DATA*/

     /*DEVICE NR IS ALREADY DEFINED*/
     CCEMCMND = "ack";         /* COMMAND TO BE SENT                */
     CCUICMND = $(00);         /* SEND COMMAND WITHOUT DATA         */
     CCCONNECT = $(03);        /* AUTOMATIC CONNECT/DISCONNECT      */
     CALL BCC_08000;           /* EXECUTE                           */

     (Transmission to DU)

DU:  (Return from UIM)
     IF DCDSTAT ] = 0          /* ERROR STATUS OF UICB              */
       THEN
         GOTO ERROR;
     SELECT (DCEMCMND);
     .
     .
     WHEN "ack" DO;
     DCEMCMND = "ack";         /* ACK OF ACKNOWLEDGEMENT            */
     DCUIMCND = $(00);         /* SEND COMMAND WITHOUT DATA         */
     Call BCG_08000;
     (Transmission to CP)

     (Return from UIM)

CP:  IF CCDSTAT ] = 0          /* ERROR STATUS OF UICB
       THEN
         GOTO ERROR;
     /*WAIT FOR NEXT MESSAGE FROM HOST COMPUTER*/
```

8.12.5 Text from Host Computer

```
CP:  CCUIMCMND = $(01);          /*SEND TEXT TO DU                  */
     CCEMCMND = "text";
     CCDEVNR1 = "DU nr";
     CCSDSTRT = ADDR(SENDBUF);   /*DEFINE BUFFER                    */
     CCSDLEN = BUFLEN;
     CCCONNECT = $(03);          /*AUTOMATIC CONNECT/DISCONNECT     */
     CALL BCC_08000;             /*EXECUTE                          */
     (Data transmitted to DU)

DU:  (Return from UIM)
     IF DCDSTAT ] = 0            /*ERROR STATUS                     */
       THEN
         GOTO ERROR;
     SELECT(DCEMCMND);
     .
     .
     WHEN "text" DO:
     /*EDIT TEXT*/
     DCUICMND = $(00);           /*SEND ACK TO CP                   */
     DCEMCMND = "ack";
     DCRDSTRT = ADDR(RECBUF);
     DCRDLEN = BUFLEN;
     Call BCG_08000;             /*EXECUTE                          */
     (Command transmission to CP)

CP:  (Return from UIM)
     IF CCDSTAT ] = 0            /*ERROR STATUS                     */
       THEN
         GOTO ERROR;
     /*SEND ACK TO HOST COMPUTER*/
```

8.12.6 Session Concluded by Host Computer

```
CP:  /*EOT RECEIVED FROM HOST COMPUTER*/
     /* SEND ON TO DU*/
     CCUICMND = $(00);           /*SEND COMMAND WITHOUT DATA        */
     CCEMCMND = EOT;             /*END OF TRANSMISSION              */
     CCCONNECT = $(03);          /*AUTOMATIC CONNECT DISCONNECT     */
     CALL BCC_08000;             /*EXECUTE                          */
     (Send command to DU)

DU:  (Return from UIM)
     IF DCDSTAT ] = 0            /*ERROR STATUS                     */
       THEN
         GOTO ERROR;
     SELECT (DCEMCMND);
     .
     .
     WHEN (EOT) DO;
     /*PROCESS EOT*/
```

```
      DCUICMND = $(00);             /*SEND COMMAND WITHOUT DATA       */
      DCEMCMND = "ack";
      CALL BCG_08000;               /*EXECUTE                         */
      (Command transmitted to CP)

 CP:  (Return from UIM)
      IF STATUS ] = 0               /*ERROR STATUS                    */
        THEN
          GOTO ERROR;
      /*ELSE SESSION CONCLUDED*/
```

9 DISPLAY UNIT FUNCTIONS

List of Contents

9.1	GENERAL DU DESCRIPTION	3
9.2	DISPLAY AREA	4
9.2.1	Functional Description	4
9.2.2	Interface for Default Display Area	4
9.2.3	Character Generation Codes	4
9.2.4	Field Attribute Characters	5
9.3	INITIALIZING THE ADAPTATION CIRCUITRY	6
9.3.1	Functional Description	6
9.3.2	Interface	6
9.3.3	IPL Initialization	10
9.4	CURSOR HANDLING	11
9.4.1	Functional Description	11
9.4.2	Interface	11
9.5	MESSAGE LINE	12
9.5.1	Functional Description	12
9.5.2	Message Line Control Block	12
9.5.3	Interface to Message Line	14
9.6	HARDWARE DEPENDENT FUNCTIONS	15
9.6.1	Hardware Identification	15
9.6.2	Maxram	16
9.6.3	DU 4111 Characteristics	16
9.6.4	DU 4112 Characteristics	16
9.6.5	E241	17
9.6.6	DU 4113 Characteristics	17
9.6.7	DPU 4173 Characteristics (WS 3111)	17
9.7	APPLICATION EXAMPLES	18
9.7.1	Initialization and Cursor Handling	18
9.7.2	Example of Message Line Handling	20-20

9.1 GENERAL DU DESCRIPTION

The following description of the display unit functions is based on DU 4110. However, the main part of the described functions applies to all DU models. The functions that are dependent on a special hardware configuration are presented in the end of this section.

The DU contains a certain memory area, which is called the display area. Each cell in the display area corresponds to a particular character position on the screen. The memory cells contain the code for the character currently being displayed on the screen. A special adaptation circuitry (DIA) continously interprets the content of the display area, and transfers the characters to the screen.

A number of system parameters are used to define the DU functions. These parameters are listed and explained in section on Customizing Data in document on Maintenance.

9.2 DISPLAY AREA

9.2.1 Functional Description

The size of the required display area depends on desired emulation and screen format. The display area may be located anywhere between the addresses 6000 and 7FFF hexadecimal (E002 and F4A2 in DU 4113).

The operating system provides a default display area containing 1920 bytes (3840 in DU 4113). This area can be used for IBM type presentation in a screen format of 25 x 80 including message line.

For other presentation types or other screen formats, the display area must be specified and declared in the system module.

9.2.2 Interface for Default Display Area

The default display area is declared in the structure BED_07200. The following declarations are included in the system module if the default display area is to be used.

DU 4110, 4111 and 4112:

```
DECLARE BED_$7200 CHAR(1919) EXT;        /* DISPLAY AREA        */
DELCARE BED_$7210 CHAR EXT;              /* LAST CHARACTER      */
                                         /* POSITION ON THE     */
                                         /* SCREEN              */
```

DU 4113:

```
DECLARE BED_$7200 CHAR(4000) EXT;
DELCARE BED_$7210 CHAR EXT;
```

The default display area starts at address 7830 (E502 in DU 4113) and ends at address 7FAF (F400). The address of BED_$7210 is thus 7FAF (F401).

9.2.3 Character Generation Codes

Each supported character in the display unit has its own unique character generation code, describing its layout in the cell on the screen.

The translation from the character code (e.g. ASCII) to the character generation code is performed via a character generation table. This table contains each supported character code and its corresponding generation code.

In DU 4110 the character generation table is stored in PROM. In later DU models, the table is loaded from diskette and stored in the RWM.

The character generation tables are listed in Appendix.

See also section on Keyboard Input below.

9.2.4 Field Attribute Characters

Field attribute characters (FAC) are used to define the attributes of a display field. A field can be protected from input, displayed with higher intensity etc. Each FAC can apply either to the end of the line or to the next FAC on the screen.

The interpretation of the FAC codes depend on desired emulation, system parameters and hardware. See the technical description and the reference manuals for the IBM and Univac emulations.

Extended attributes are used in DU 4113 to provide colour and graphic characters. When extended attributes are used every position on the display corresponds to two buffer positions - one for the extended attribute and one for the character to be displayed.

9.3 INITIALIZING THE ADAPTATION CIRCUITRY

9.3.1 Functional Description

To initialize the adaptation circuitry for display purposes, a table of initialization parameters must be defined by the user. The parameters are stored in a variable containing 20 bytes. The address of this variable is handed over to the operating system via the external pointer BED_$7400.

The global procedure BED_07100 carries out the initialization. When this procedure is called, the adaptation circuitry is initialized and the parameter values are then available in BED_07100.

9.3.2 Interface

Declarations:

```
DECLARE INITAB CHAR(20);                  /* USER DEFINED VARIABLE    */
DECLARE 1 BED_$7100 EXTERNAL,
      5 DIARSTRT   PTR,                   /* START OF DISPLAY AREA    */
      5 DIAREND    PTR,                   /* END OF DISPLAY AREA      */
      5 DIARLLEN   BYTE,                  /* LINELENGTH (CH/L)        */
      5 DIARLNR    BYTE,                  /* NUMBER OF LINES EXCL ML  */
      5 DIAPRES1   BYTE;                  /* SEE BYTE 19              */
      5 DIAPRES2   BYTE;                  /* SEE BYTE 20              */

DECLARE BED_$7400 PTR EXTERNAL;           /* PTR TO INITAB            */
DECLARE BED_07100 ENTRY (PTR VALUE);      /* INITIALIZATION PROC      */
```

Before the call to the initializing procedure, INITAB is initialized as follows.

INITAB parameters:

Code (hex) for screen formats including message line:

Byte	13x40	13x80	17x80	25x80	33x80	44x80	Meaning
1	63	63	63	63	63	63	Horizontal total
2	28	50	50	50	50	50	Horizontal displayed (number of characters/line)
3	44	58	58	58*	58*	58	Horizontal sync position
4	06	06	06	06	06	06	Horizontal sync width
5	12	12	12	19	21	2C	Vertical total
6	08	08	08	0A	12	15	Vertical total adjust
7	0D	0D	11	19	21	2C	Vertical displayed (number of lines including message line)
8	0F	0F	12	19	21	2C	Vertical sync position
9	00	00	00	00	00	00	Interlace mode
10	15	15	15	0F	0B	08	Number of sweeps per displayed line minus one

*) 56 in DU 4113 (only format 28 x 35 and 33 x 80 in this unit).

	Bit 7 6 5 4 3 2 1 0	Meaning
Byte 11	1 1	Flashing cursor, 1.5 Hz
	1 0	Flashing cursor, 3 Hz
	0 1	No cursor
	0 0	Steadily glowing cursor
	X X X X X	Sweep No. within line for top edge of cursor
Byte 12	X X X X X	Sweep No. within line for bottom edge of cursor
Bytes 13-14		Absolute start address of display area
Bytes 15-16		Cursor address after initialization
Bytes 17-18		End address of display area, excl. message line ≤7FFF (hex)

Continued on next page.

Byte 19 - DIA PIA control byte A.

DU 4110, 4111 and 4112:

```
Bit
 7 6 5 4 3 2 1 0

 1                  Enable FAC interpretation.
 0                  Disable FAC interpretation. Does not
                    affect interpretation of 5F 00, 20 or
                    MLFAC in DU 4111. See the technical
                    description.
   1                Reset CRTC/Disable presentation.
   0                Enable presentation.
     X              Field underlining codes. See the
                    technical description.
       1            Characters visible through cursor.
       0            Characters covered by cursor.
         X X X      FAC control bits. See the technical
                    description.
               1    Inverse video.
               0    Normal video.
```

DU 4113:

```
Bit
 7 6 5 4 3 2 1 0

 1                  Enable FAC interpretation.
 0                  Disable FAC interpretation. Does not
                    affect interpretation of 5F 00, 20 or
                    MLFAC in DU 4111. See the technical
                    description.
   1                Reset CRTC/Disable presentation.
   0                Enable presentation.
     X              Not used.
       1            Characters visible through cursor.
       0            Characters covered by cursor.
         X X X X    Not used.
```

Continued on next page.

Byte 20 - DIA PIA control byte B

DU 4110, 4111 and 4112:

Bit 7	6	5	4	3	2	1	0	
1								Disable wraparound of FACs
0								Enable wraparound of FACs
	X	X						Number of sweeps in character cell:
	0	0						16 sweeps/cell (25 lines)
	0	1						12 sweeps/cell (33 lines)
	1	0						9 sweeps/cell (44 lines)
	1	1						22 sweeps/cell (< 25 lines)
								Note. If 01 or 10, no space underlining provided.
			1					Video enabled.
			0					Video disabled.
				1				IBM type interpretation of FACs.
				0				UTS type interpretation of FACs.
					0	0		Presentation mode 3.
					0	1		Presentation mode 2.
					1	0		Presentation mode 1.
					1	1		Presentation mode 0. See the technical description
							X	Scope of FACs. See the technical description.

DU 4113:

Bit 7	6	5	4	3	2	1	0	
1								Disable wraparound of FACs
0								Enable wraparound of FACs
	X	X						Number of sweeps in character cell:
	0							12 sweeps/cell
	1							16 sweeps/cell
		X						Not used
								Note. If 01 or 10, no space underlining provided.
			1					Video enabled.
			0					Video disabled.
				X				Not used
					0			Non Base Colour mode
					1			BNase Colour flag effective
						X		Not used
							X	Scope of FACs. See the technical description.

The control registers of the Display Adaptation Peripheral Interface Adapter (DIA PIA) are described in detail in the technical description.

Call format; Initialization of adaptation circuitry:

```
BED_$7400 = ADDR(INITAB);
CALL BED_07100(BED_$7400);
```

After the call, the structure BED_$7100 contains the parameters defined in INITAB.

9.3.3 IPL Initialization

At IPL time, a default initialization table designated BED-$7300 is used. After IPL initialization, BED-$7100 contains the following:

Identifier	Contents (hex)
DIARSTRT	Start address of display area
DIAREND	End address of display area
DIARLLEN	50
DIARLNR	18
DIAPRES1	00
DIAPRES2	11 (10 in DU 4113)

9.4 CURSOR HANDLING

9.4.1 Functional Description

Cursor handling is controlled by means of calls to the procedure BED_07000.

The execution of the procedure is performed in accordance with the parameters initialized in the structure BED_$7002. Initialization must be carried out before each procedure call.

9.4.2 Interface

Declarations:

```
DECLARE 1 BED-$7002 EXTERNAL,
          5 CURADD PTR,
          5 CURSTR BYTE,
          5 CUREND BYTE;
DECLARE BED-07000 ENTRY(BYTE VALUE);
```

Before the procedure call, the structure BED_$7002 is initialized as follows:

Identifier	Bit 7 6 5 4 3 2 1 0	Meaning
CURADD		Absolute memory address of cursor (in display area)
CURSTR	1 1	Flashing cursor, 1.5 Hz
	1 0	Flashing cursor, 3.0 Hz
	0 1	No cursor
	0 0	Steadily glowing cursor
	X X X X X	Sweep No. for top edge of cursor
CUREND	X X X X X	Sweep No. for bottom edge of cursor (≤ 21, 15, 11 or 8 depending on display format)

Call format, cursor handling:

```
CALL BED-07000($(02));
```

Note that the parameter must always be $(02) when the cursor is used on the ordinary display screen. Other values are used by the Message Line Handler.

9.5 MESSAGE LINE

9.5.1 Functional Description

The bottom line on the screen is a system line which is used for messages to the operator from OS and from the emulation. The message line can also be used for simple input from the operator.

The message line is implemented as two message buffers, one for OS and the other for the emulation module. The first five positions of each buffer are always occupied by the buffer indicators *OS* and *EM*.

Two keyboard keys are used only in connection with the message line:
ROLL ML key - to scroll between the two buffers
CU TO ML key - to move the cursor to and from the message line.

The message line can be utilized by the system module by means of calls to the Message Line Handler (MLH). Its functon is specified in the Message Line Control Block (MLCB). An externally declared event variable is used to indicate completed operator input.

The Message Line Handler contains the following main parts:

- A procedure which is called to request input/output on ML.
- A task which handles communication with the Display Area Handler and the Keyboard Handler.

Two field attribute characters (FAC) can be used on the message line: A0, which indicates protected field, and 80, which indicates unprotected field.

The message line is further discussed in the reference manual for the emulation in question.

9.5.2 Message Line Control Block

The Message Line Control Block (MLCB) is a structure which the user declares and utilizes in order to control communication with the message line handler.

The message line control block has the following structure.

```
DECLARE 1 MLDATA,                    /* ML CONTROL BLOCK            */
          5 ID CHAR(2),
          5 MSGPOINT,
            10 FROMADDR POINTER,
            10 TOADDR POINTER,
          5 CURPOS BYTE,
          5 CONTROL BYTE,
          5 MLSTATUS BYTE,
          5 SENDKEYS(4) BYTE;
```

The identifiers in Message Line Control Block have the following meanings:

Identifier	Meaning
ID	Specifies the calling module. OS: Operating system EM: System module
FROMADDR	Start address for message from system module to ML.
TOADDR	Start address for reply message to system module from ML. The entire ML is transferred. TOADDR is specified only if input on ML is requested.
CURPOS	Not used.
CONTROL	Control byte, interpreted as follows:

```
Bit 7 6 5 4 3 2 1 0
    1                 Output and input on ML.
    0                 Output only.
      X X             Not used.
          1           Cursor moved to ML.
          0           Cursor not affected.
            1         Erase ML. If this bit is set, the
                      previous message is replaced by
                      spaces.
              1       Jail cursor. The cursor cannot be
                      moved from ML until input is
                      completed. (CU TO ML is
                      deactivated).
                1     ENTER key concludes input.
                0     Other key concludes input. See
                      SENDKEYS below.
                  1   Update ML. Last written message
                      is rewritten.
```

Identifier	Meaning
MLSTATUS	Always $(FF)
SENDKEYS(4)	Desired input termination key. See bit 1 above.
Byte 1	First byte in keyboard table.
Byte 2	Second byte in keyboard table.
Byte 3	Not used.
Byte 4	Not used.

The keyboard table is described in section on Keyboard Functions.

9.5.3 Interface to Message Line

Declarations:

```
DCL 1 MLDATA,                            /* MLCB, SEE ABOVE            */
      5 ID CHAR(2),
      5 MSGPOINT,
        10 FROMADDR PTR,
        10 TOADDR PTR,
      5 CURPOS BYTE,
      5 CONTROL BYTE,
      5 MLSTATUS BYTE,
      5 SENDKEYS(4) BYTE;
DCL MLDPTR PTR;                          /* POINTER TO MLCB            */
DCL OUTMSG CHAR(75);
DCL INMSG CHAR(75);
DCL BGD-$0020 EVENT EXTERNAL;            /* INDICATES COMPLETED INPUT  */
DCL BGD-00110 ENTRY (PTR VALUE);         /* MESSAGE LINE HANDLER       */
```

Input and output buffers must always be declared as 75-character buffers. The message line has 80 positions, but 5 positions are occupied by the indicators *EM* or *OS*.

The event variable BGD_$0020 is used for system modules. For application modules, the variable BGD_$0030 must be used. The events are posted when the specified input termination key is depressed.

Call format:

```
ID = 'EM';
FROMADDR = ADDR(OUTMSG);
TOADDR = ADDR(INMSG);                    If input is to take place
CONTROL = 'suitable code';
SENDKEYS(1) = 'suitable function code if input is required';
SENDKEYS(2) = 'suitable function code if input is required';
MLDPTR = ADDR(MLDATA);
CALL BGD_00110(MLDPTR);
```

If input is to take place:

```
ASSIGN BGD_$0020;
WAIT BGD_$0020;
```

9.6 HARDWARE DEPENDENT FUNCTIONS

The functions described above apply to display units in general. However, the following hardware characteristics should be considered. The operating system contains internal tests which inquire the current hardware configuration, and adapt its functions accordingly. Please refer to the Technical Description for further details.

9.6.1 Hardware Identification

To make it possible for the system modules to inquire about the hardware configuration, each PROM is assigned a unique identification code.

The identification codes must be declared by the user:

```
DECLARE 1 PROMFUNC EXTERNAL,
          2 FUNC CHAR(4),          /* FUNCTION BIT MASK        */
          2 PROMID BYTE;           /* UNIT TYPE IDENTIFIER     */
```

The PROMID byte is interpreted as follows (hexadecimal):

```
PROMID:     FF = Single DTC + DTC-A, FD 4120 and older FD units
            01 = Cluster DTC + DTC-A
            02 = DU 4111
            03 = DU 4112
            04 = WS 641
            05 = DU 4113
            06 = FD 4122
            07 = PCU 4171 (Printer PCU)
            08 = DPU 4173
```

The FUNC character is interpreted as follows:

```
Bit 7 6 5 4 3 2 1 0
    0                         Internal logic of type DU 4111
      0                       4 colours available
          0                   7 colours available
```

Note that bit set indicates inactive function. Bits 5, 3, 2, 1, 0 are not used.

The following addresses are used for the identifiers:
PROMID: FFE7
FUNC: FFE3
OSFUNC: 02E8 (in OS version 3 only)

The structure OSFUNC describes among other things the keyboard functions. See section on Keyboard Input.

9.6.2 Maxram

The size of the RWM area in DU is stored in the globally declared variable MAXRAM.

MAXRAM can contain one of the following values:

		Totally in unit:
MAXRAM = 7FFC	No MRW	32 K
MAXRAM = BFFC	16 K MRW	32 K + 16 K
MAXRAM = CFFC	20 K MRW	32 K + 20 K
MAXRAM = DFFC	24 K MRW	32 K + 24 K
MAXRAM = EFFC	28 K MRW	32 K + 28 K
MAXRAM = F67C	30 K MRW	32 K + 30 K

The address of MAXRAM is 02C7 hexadecimal in OS version 3.

9.6.3 DU 4111 Characteristics

Display Unit 4111 can be regarded as a cost reduced version of DU 4110. The main characteristics of DU 4111 are listed below. For a detailed description of the hardware, please refer to the Technical Description.

- DU 4111 loads the character generation table from diskette
- The interrupt handling of DU 4111 is implemented in software
- DU 4111 provides no Program Counter error handling
- DU 4111 supports address switching on two-wire dropline (not presently used)
- DU 4111 contains 64 kbyte RWM as a standard feature
- DU 4111 cannot handle light pen

9.6.4 DU 4112 Characteristics

Display Unit 4112 is a 4-colour display unit in which the colour selection is controlled by means of field attribute characters. Apart from the colour features, DU 4112 is similar to DU 4111.

The colour representation is based on the colours green, red, blue and white. In two-colour mode, green and white are used. Changing of the colour representation can be carried out by calling the OS procedure BED_07700 with a byte parameter.

Changing of colour representation from the keyboard requires a keyboard table with appropriate adaptations.

The byte parameter in a call to BED_07700 can have the following values:

0 Set two-colour mode
1 Set four-colour mode
2 Change colour mode

The external byte BED_$7701 indicates the present mode.
If bit No. 3 = 0: two colour mode
If bit No. 3 = 1: four-colour mode

The logon handler initiates DU 4112 to four-colour mode during logon.

9.6.5 E241

E241 is an emulation which is loaded into any Alfaskop display unit. It is intended for communication with a series 2000 minicomputer, via a certain supervisor (sv6).

The XC interface is connected to a series 2000 minicomputer in one end and in the other to a CP. SC is the software in XC that handles two-wire communication. (FD logical address is used.)

9.6.6 DU 4113 Characteristics

Display unit 4113 is a 7-colour display unit that is primarily intended for IBM-emulations.

A printer can not be connected to DU 4113. Nor can this display unit communicate with Keyboard 4140.

DU 4113 supports the IBM Extended Data Stream. Because of this facility two bytes per screen position must be stored in the display memory (see 9.2.4).

Also the character generator handling differs somewhat from that of the other display units in S41.

In other respects DU 4113 is similar to DU 4112.

9.6.7 DPU 4173 Characteristics (WS 3111)

Workstation 3111 is intended for emulation of IBM 3178 only.

WS 3111 is constituted of three units: a 12" display monitor, a display unit processor (DPU 4173) and a keyboard.

A selector pen can not be connected to the WS 3111.

9.7 APPLICATION EXAMPLES

9.7.1 Initialization and Cursor Handling

Note that the table BED_$1000 is a global variable from which the system module can fetch characters that have been entered from the keyboard. The fourth byte contains the converted key code. See also section on Keyboard Input.

```
DECLARE BED-$1000 CHAR(5) EXTERNAL; /* ENTERED CHARACTERS          */
DECLARE 1 BED-$7100 EXTERNAL,       /* DIA INITIALIZATION PARAMETERS*/
            5 DIARSTRT   POINTER,
            5 DIAREND    POINTER,
            5 DIARLLEN   BYTE,
            5 DIARLNR    BYTE,
            5 DIAPRES1   BYTE,
            5 DIAPRES2   BYTE;
DECLARE 1 BED-$7002 EXTERNAL, /* CURSOR INITIALIZATION PARAMETERS   */
            5 CURADD     POINTER,
            5 CURSTR     BYTE,
            5 CUREND     BYTE;
DECLARE DISPLAY CHAR BASED(CURADD);
DECLARE GENCODE CHAR DEFINE BED_1000 POS(4); /* CONVERTED KEY CODE  */
DECLARE WORKPTR POINTER;
DECLARE ERASE CHAR BASED(WORKPTR);
DECLARE BED-07000 ENTRY(BYTE VALUE); /* CURSOR HANDLING PROCEDURE   */
.
.
.
/* INITIALIZE CURSOR TO FIRST POSITION IN SECOND LINE:             */
CURSTR = $(42);
CUREND = $(0A);
CURADD = DIARSTRT + DIARLLEN;
CALL BED-07000($(02));
.
.

/* DISPLAY THE NEXT CHARACTER ENTERED FROM THE KEYBOARD             */
DISPLAY = CHARCODE;
CURADD = CURADD + 1;
.
.
/* CHECK THAT CURADD DOES NOT POINT BEYOND THE                      */
/* DISPLAY AREA                                                     */
.
.
CALL BED-07000($(02));
.
.
```

```
/* MOVE CURSOR TO FIRST POSITION IN NEXT LINE:                  */
WORKPTR = DIARSTRT;
DO WHILE(WORKPTR < CURADD);
WORKPTR = WORKPTR + DIARLLEN;
.
.
/* CHECK THAT WORKPTR DOES NOT POINT BEYOND THE                 */
/* DISPLAY AREA                                                 */
.
.
END;
CURADD = WORKPTR;
CALL BED-07000($(02));

/* ERASE ALL CHARACTERS STARTING AT THE CURSOR POSITION         */
/* UP TO AND INCLUDING THE POSITION IMMEDIATELY BEFORE THE      */
/* NEXT FIELD ATTRIBUTE CHARACTER                               */
/* FIELD ATTRIBUTE CHARACTER IS ASSUMED TO HAVE BIT NO. 7 SET   */
WORKPTR = CURADD;
DO WHILE(ERASE < $(80));
ERASE = $(20);
WORKPTR = WORKPTR + 1;
.
.
/* CHECK THAT WORKPTR DOES NOT POINT BEYOND                     */
/* THE DISPLAY AREA                                             */
.
.
END;
```

9.7.2 Example of Message Line Handling

The following example shows how a message can be entered onto ML and how input from ML can be handled.

```
DECLARE 1 MLDATA,                     /* ML CONTROL BLOCK             */
          5 ID CHARACTER(2),          /* CALL IDENTIFIER OS OR EM     */
          5 MSGPOINT,                 /* BUFFER POINTERS              */
            10 FROMADDR POINTER,      /* ADDRESS OF                   */
                                      /* OUTPUT MESSAGE BUFFER        */
            10 TOADDR POINTER,        /* ADDRESS OF INPUT             */
                                      /* MESSAGE BUFFER               */
          5 CURPOS BYTE,              /* NOT USED                     */
          5 SENDKEYS(4) BYTE;         /* INPUT TERMINATION KEY        */
DECLARE MLDPTR POINTER;               /* POINTER TO MLCB              */
DECLARE OUTMSG CHARACTER(75);         /* OUTPUT BUFFER                */
DECLARE INMSG CHARACTER(75);          /* INPUT BUFFER                 */
DECLARE BGD_$0020 EVENT EXTERNAL;     /* FOR SYSTEM MODULE            */
DECLARE BGD_00110 ENTRY (POINTER VALUE) /* MESSAGE LINE HANDLER       */
.
.
.
FROMADDR = ADDR(OUTMSG);
OUTMSG = 'ERROR MESSAGE ....';
ID = 'EM';
FROMADDR = ADDR(OUTMSG);
CONTROL = $(00);
MLDPTR = ADDR(MLDATA);                /*OUTPUT ONLY                   */
CALL BGD-00110(MLDPTR);
.
.
.
OUTMSG = 'ENTER NAME OF DESIRED MODULE:.....';
TOADDR = ADDR(INMSG);
CONTROL = $(90);                      /*INPUT, CURSOR TO MESSAGE LINE*/
SENDKEYS(1) = 'function code for PF1 first byte';
SENDKEYS(2) = 'function code for PF1 second byte';
MLDPTR = ADDR (MLDATA);
CALL BGD-00100(MLDPTR);               /* LINE WILL BE PRESENTED       */
ASSIGN BGD-$0020;
WAIT BGD-$0020;                       /* WAIT FOR INPUT COMPLETION    */
.
.
```

10 KEYBOARD FUNCTIONS

List of Contents

10.1	GENERAL KEYBOARD DESCRIPTION	3
10.1.1	Keyboard Input Synchronization	3
10.1.2	Keyboard Type Indication	5
10.2	KEYBOARD DATA STRUCTURES	6
10.2.1	Keyboard Table	6
10.2.2	Keyboard Table Header	8
10.2.3	Strap Data for KBU 4140-XXX	10
10.2.4	Normal Strap Data for KBU 4143	11
10.2.5	Extended Strap Data for KBU 4143	12
10.3	OPENING AND CLOSING FOR INPUT	13
10.3.1	Functional Description	13
10.3.2	Interface	13
10.4	KEYBOARD INPUT	14
10.4.1	Functional Description	14
10.4.2	Interface	15
10.5	KEYBOARD REPETITION FREQUENCY	16
10.5.1	Functional Description	16
10.5.2	Interface	16
10.6	CLICK SOUND ACKNOWLEDGEMENT	16
10.6.1	Functional Description	16
10.6.2	Interface	16
10.7	ALARM	17
10.7.1	Functional Description	17
10.7.2	Interface	17
10.8	KEYBOARD LAMPS	18
10.8.1	Functional Description	18
10.8.2	Interface	18
10.9	MAGNETIC ID INPUT	19
10.9.1	Functional Description	19
10.9.2	Interface	20
10.10	SELECTOR PEN INPUT (DU 4110 only)	21
10.10.1	Functional Description	21
10.10.2	Interface	21
10.11	APPLICATION EXAMPLE	22-22

10.1 GENERAL KEYBOARD DESCRIPTION

This section describes the keyboard data structure and the keyboard functions. The MID and selector pen functions are also described here.

Two main types of keyboard units are available in Alfaskop System: KBU 4140-XXX and KBU 4143. The following descriptions apply to both keyboard types, if nothing else is stated.

Communication between the DU and the keyboard, which contains its own microprocessor, is carried out asynchronously in serial form.

The OS procedures which handle operator input are gathered in the program module Keyboard Handler.

All input is transferred by the Keyboard Handler to the global area BED_$1000, where it becomes accessible to the user. This area can contain only one character at a time. Synchronization of input and processing must thus be performed.

10.1.1 Keyboard Input Synchronization

The synchronization of input and processing is accomplished by means of an input buffer and two event variables.

The event BED_$1900 is posted by the system module when it is ready to receive a new character. The event BED_$1910 is posted by the Keyboard Handler when the new character is transferred to the area BED_$1000.

When a keyboard key is pressed, a status byte and a data byte is sent from the keyboard in response to the order (poll) byte from the display unit. An IRQ interrupt is obtained for every byte sent to or from the display unit. This interrupt initiates a chain of activities as described in Fig. 10.1 below.

1. A keyboard key is pressed.

2. During the next KB scanning cycle, the physical key number is sent to the keyboard buffer. The buffer is a FIFO queue, which can contain up to 16 characters.

3. If the event BED_$1900 has been posted by the emulation, the first character in the buffer is transferred to the BED_$1000 area. When this transfer is completed, the event BED_$1910 is posted by the Keyboard Handler.

4. The character in BED_1000 is read and processed by the system module. When the module is ready to receive a new character, the event BED_1900 is posted.

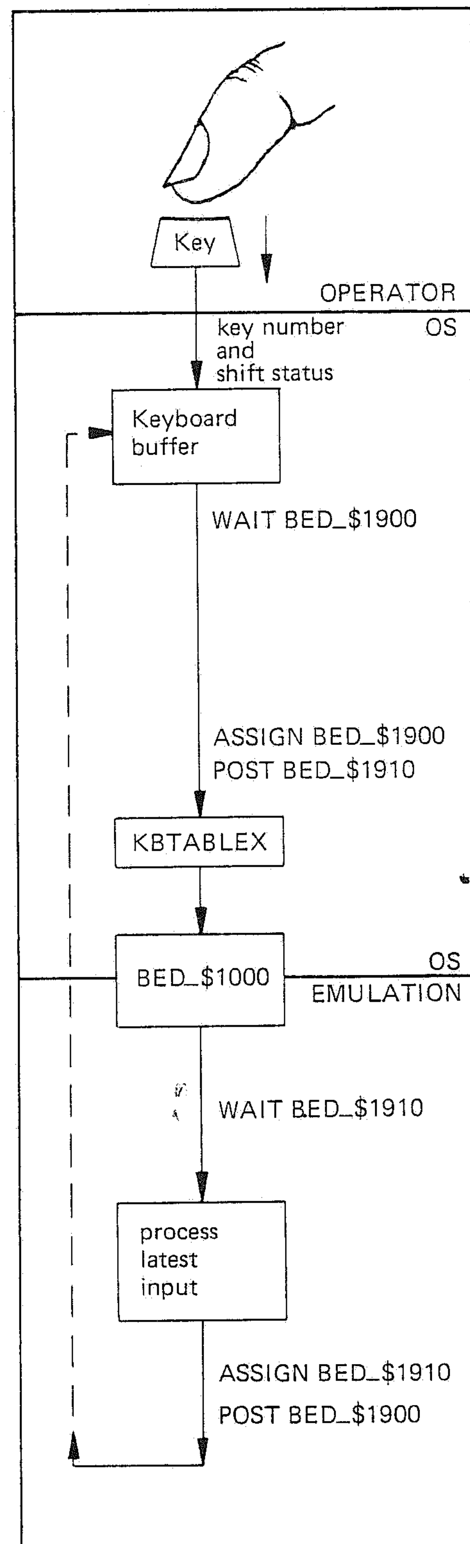


Fig 10.1 Keyboard Input synchronization

10.1.2 Keyboard Type Indication

During initial program loading, the Keyboard Handler inquires what keyboard type is currently being connected to the DU.

In OS version 3 the keyboard type indicator must be declared by the user on address 02E8 hexadecimal.

```
DECLARE 1 OSFUNC EXTERNAL,
          2 OSFUNC1 BYTE,
          2 OSFUNC2 BYTE;
```

OSFUNC1 contains the following information:

```
Bit 7  6 5 4 3 2 1 0
    1                    Character generator in software
    0                    Character generator in PROM
       1                 Keyboard Unit 4143
       0                 Keyboard Unit 4140-XXX
         X X X X X X     Not used
```

10.2 KEYBOARD DATA STRUCTURES

When a keyboard key is pressed, information is transferred through several data structures:

- o To the keyboard buffer
- o To the keyboard table, where the physical key number is translated to an appropriate character or function code
- o To the character generator table, in which this code is translated to a display code, as described in section on DU functions
- o To the globally declared area BED_$1000, where the code becomes available to the user

The keyboard buffer and the common area BED_$1000 are always provided by the operating system.

The character generator table can either be stored permanently in PROM (DU 4110) or be loaded from the system diskette into RWM.

The keyboard tables are further discussed below.

10.2.1 Keyboard Table

The keyboard table provides the translation from the physical key number (and shift state) to the appropriate character or function code.

Up to 5 keyboard tables can be defined in a cluster. All defined tables are stored in library KBLIB or KBLIBA on the system diskette. The keyboard tables are named KBTABLE0, KBTABLE1 ... KBTABLE4.

A maximum of 128 physical keys can be defined in a keyboard table. However, up to 16 different shift states can also be specified. This implies that each physical key can be associated with up to 16 different logical keys.

Each logical key in the keyboard table is defined to be of a certain key type: alphanumeric key, function key, short message key or system key. Thus, two bytes are required to describe each logical key in the table. The first byte defines the key type and the second byte defines the character code.

A keyboard table thus contains 128 x 2 x N bytes, where N is the number of supported shift states. See the figure below.

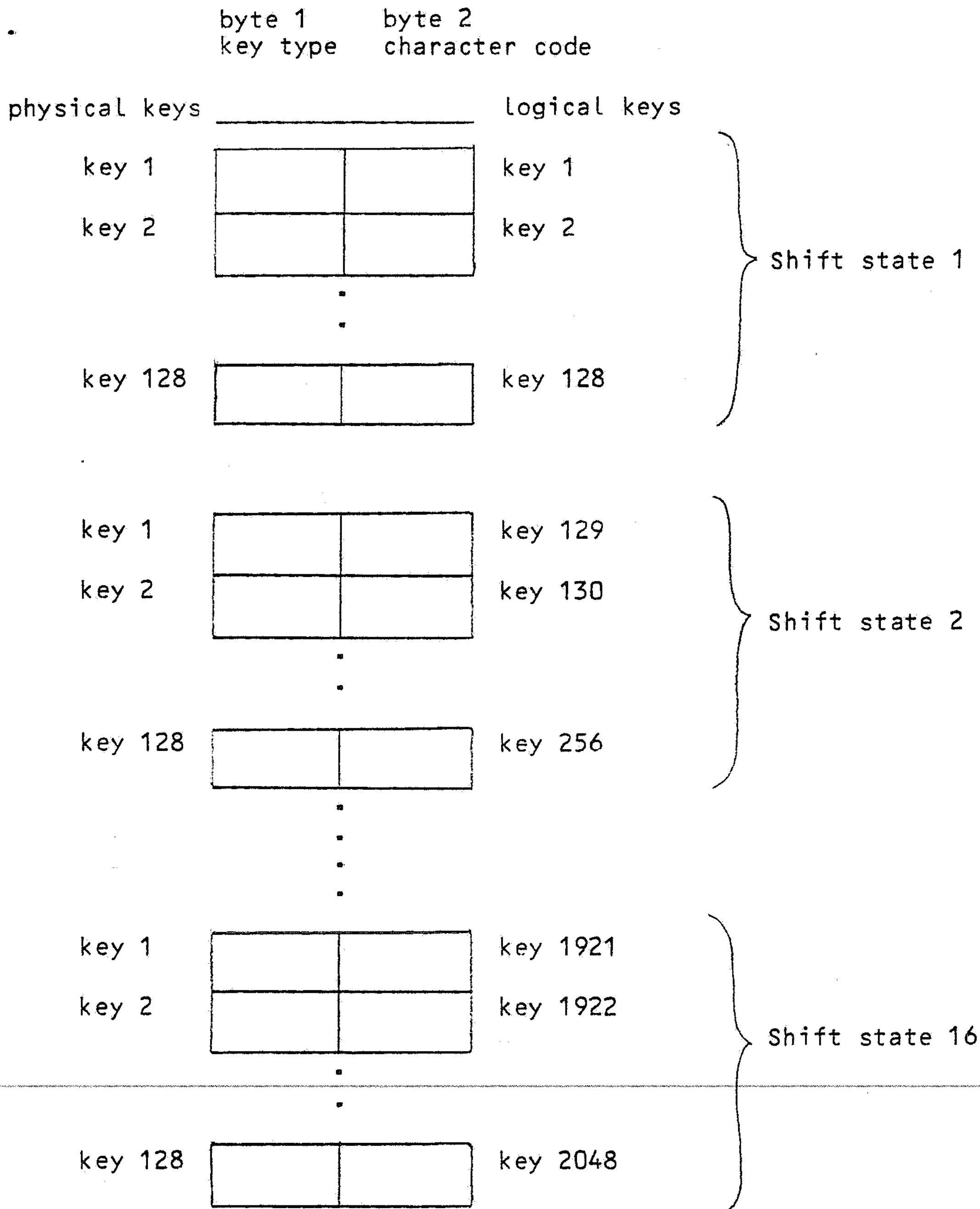


Fig 10.2 Keyboard table structure

Each pair of bytes in the keyboard table has the following meaning.

```
Byte 1                       Key type
Bit 7 6 5 4 3 2 1 0
    1                        System key. Internal use only.
      1                      Repetition ignored
        X X X X              Emulation dependent codes
    X X X X X 1 X X          APL-key in DU 4113
                0 0          Alphanumeric key
                0 1          Short-message key, PA functions
                1 0          Function key
                1 1          Short-message key, PF functions

Byte 2                       Key code, representation depending on key
Bit 7 6 5 4 3 2 1 0          type:
    X X X X X X X X          o Character code if alphanumeric key
                             o Function code if function key
                             o Short-message code if short-message key
                               (PA or PF)
```

10.2.2 Keyboard Table Header

A keyboard table header must be provided by the user. The keyboard table header contains keyboard strap data and 16 pointers to the keyboard subtables.

The buffer in DU for the current keyboard table and its header can be declared as follows.

```
DECLARE 1 KBTABBUF,
          5 KBSTRAP CHAR(13),          /* STRAP DATA        */
          5 KBTABPTR(16) POINTER,      /* POINTERS          */
          5 KBTAB(Nx128) CHAR(2);      /* KB TABLE          */
```

The structure of the buffer is presented in the figure below.

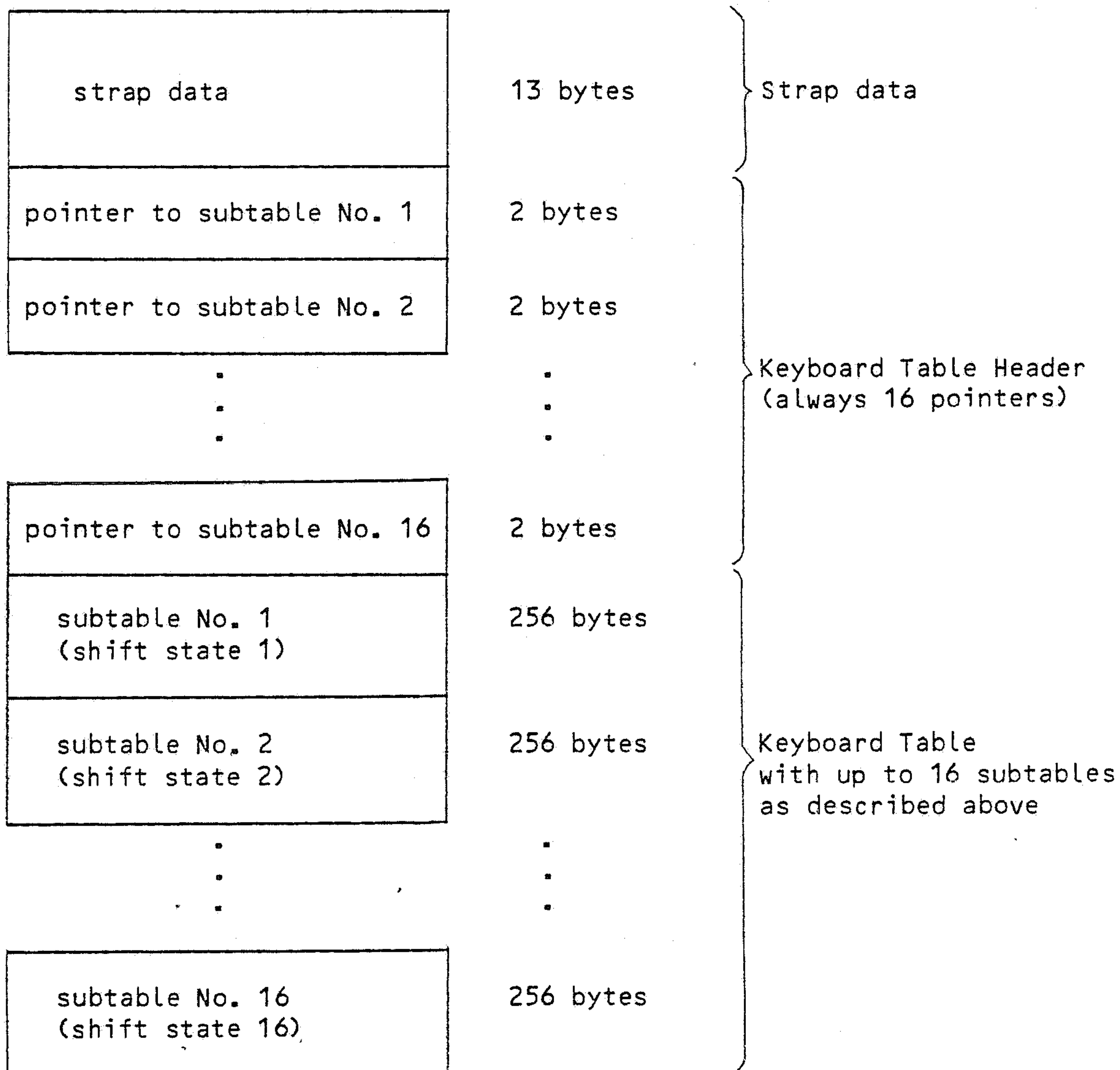


Fig 10.3 Keyboard table with header

The Keyboard Handler maintains a pointer (BED_$1100) to the keyboard table header.

The name of the assigned keyboard table is contained in BAD_70301.

The keyboard strap data are normally initialized (sent to the keyboard microprocessor) during the IPL procedure. Strap data are further described below.

The emulation module can order an exchange of the loaded keyboard table, and initiate the loading of a new table. This is further discussed in section on FD Functions and File Handling.

10.2.3 Strap Data for KBU 4140-XXX

The 13 first bytes of the keyboard table header contains strap data. In KBU 4140, these bytes are interpreted as follows:

	Bit 7 6 5 4 3 2 1 0	Meaning
Byte 1	1	Click not generated by keyboard
	X	Not used
	1	Left shift restores shift lock
	0	Left and right shifts restore shift lock
	1	Right shift has separate status bit
	0	Left and right shifts share a common status bit
	1	Shift lamp indicates shift lock
	0	Shift lamp indicates upper case
	1	Key lock used
	0	MID used
	X X	Not used
Byte 2	1 0 0 0 0 0 0 0	Lamp 8 is shift lamp
	.	.
	.	.
	.	.
	0 0 0 0 0 0 1	Lamp 1 is shift lamp
Byte 3	1 0 0 0 0 0 0 0	No shift lock key provided
	0 X X X X X X X	Key number of shift lock key
Byte 4	1 0 0 0 0 0 0 0	No left shift key (shift 1)
	0 X X X X X X X	Key number of left shift key
Byte 5	1 0 0 0 0 0 0 0	No right shift key (shift 2)
	0 X X X X X X X	Key number of right shift key
Byte 6	1 0 0 0 0 0 0 0	No shift 3 key
	0 X X X X X X X	Key number of shift 3 key
Byte 7	1 0 0 0 0 0 0 0	No shift 4 key
	0 X X X X X X X	Key number of shift 4 key
Byte 8	1 0 0 0 0 0 0 0	No special repeat key, all keys are repeating
	0 X X X X X X X	Key number of repeat key, other keys not repeating
Byte 9		Not used
Byte 10	1 1 1 1 1 1 1 1	End of strap data
Byte 11	0 0 0 0 0 0 0 0	Next poll to KB — Always same bit configuration
Byte 12	0 0 0 0 0 0 0 0	Next poll to KB — Always same bit configuration
Byte 13	1 0 1 0 1 0 1 0	End of string

10.2.4 Normal Strap Data for KBU 4143

In KBU 4143, the first 13 bytes of the keyboard table header are interpreted as follows:

	Bit 7 6 5 4 3 2 1 0	Meaning
Byte 1	1	DU alone handles click sound
	0	KB handles click sound
	1	MSR track 3 selected
	0	MSR track 2 selected
	X X	Not used
	1	Shift lamp indicates shift lock
	0	Shift lamp indicates upper case
	1	Key lock used
	0	MSR used
	1	Extended strap data (see next section)
	0	Normal strap data
	X	Not used
Byte 2	Shift lamp No.	As for KBU 4140
Byte 3	SHIFT LOCK 1	Key number assigned by KBU
Byte 4	SHIFT 1a	Key number assigned by KBU
Byte 5	SHIFT 2	Key number assigned by KBU
Byte 6	SHIFT 3	As for KBU 4140
Byte 7	SHIFT 4	As for KBU 4140
Byte 8	REPEAT KEY	As for KBU 4140
Byte 9	1 0 0 0 0 0 0 0	No programmable PF keys
	0 X X X X X X X	Key number of SEMAC key, normally 66 (hexadecimal)
Byte 10-13		As for KBU 4140

Notes.

MSR - Magnetic Strip Reader (built-in).

SEMAC = Select Macro. Key for defining and selecting a sequence of key entries by means of programmable PF-keys.

10.2.5 Extended Strap Data for KBU 4143

If bit No. 6 in the first strap data byte is set, extended strap data are used. This implies that all of the 13 first bytes, and 4 additional bytes are defined by the emulation

This implies that if extended strap data are to be used, the buffer for the keyboard table header must be extended so as to contain 17 bytes of strap data.

Bytes 1-2 are interpreted as for normal strap data above
Bytes 3-17 contain the key numbers for the following functions (all key numbers are defined by the emulation):

Strap-byte:	Key number for function:
Byte 3	SHIFT LOCK 1
Byte 4	SHIFT 1a
Byte 5	SHIFT 2
Byte 6	SHIFT 3
Byte 7	SHIFT 4
Byte 8	REPEAT KEY
Byte 9	SEMAC
Byte 10	SHIFT 1b
Byte 11	SHIFT 1c
Byte 12	SHIFT LOCK 2
Byte 13	SHIFT LOCK 3
Byte 14	SHIFT LOCK 4
Byte 15	ALARM VOLUME
Byte 16	ALARM TONE
Byte 17	CLICK VOLUME
Byte 18	$(FF) END OF STRAP DATA
Byte 19-22	As byte 10-13 in strap data for KB 4140

Notes:
1. SHIFT LOCK 1 is reset by SHIFT 1a-1d
2. SHIFT LOCK 2-4 are reset by SHIFT 2-4
3. Byte value $(80) means "Not used".
4. A string of strap bytes should be ended with $(FF)

10.3 OPENING AND CLOSING FOR INPUT

10.3.1 Functional Description

Before any input is accepted, the DU must be opened for input. Opening is performed by the OS procedure BED_02000. One parameter is required to specify the type of input accepted.

When all input is performed, the DU can be closed for all input by means of a call to the OS procedure BED_02100.

10.3.2 Interface

Declarations:

```
DECLARE BED_02000 ENTRY (BYTE VALUE);
DECLARE BED_02100 ENTRY (BYTE VALUE);
DECLARE INTYPES BYTE;
```

The byte INTYPES have the following meaning:

Bit	7	6	5	4	3	2	1	0	Meaning
	1								Keyboard key input
		1							Selector pen input
			1						Magnetic ID input
				X	X	X	X		Not used
								1	Always 1 in a call from a system module

Call formats:

```
CALL BED_02000(INTYPES);      /* FOR OPENING            */
CALL BED_02100($(01));        /* FOR CLOSING            */
```

Note that the parameter of the closing procedure must always be $(01).

10.4 KEYBOARD INPUT

10.4.1 Functional Description

When opening for keyboard input has been performed, input character codes can be read from the area BED_$1000. This area contains one character at a time. Keyboard input synchronization is described above.

The area BED_$1000 consists of five bytes having the following meaning for keyboard input:

```
Byte 1                              Shift status (16 combinations)
Bit       7 6 5 4 3 2 1 0
          1 0 0                     Always
                1                   KXU 4146 indication (in KB 4143 only)
                  1                 Shift key 4 pressed
                    1               Shift key 3 pressed
                      1             Shift key 2 pressed
                        1           Shift key 1 pressed

Byte 2                              Input type/Repetition status
Bit       7 6 5 4 3 2 1 0
          1                         Keyboard information follows
            1                       MID information follows
              1                     Selector pen information follows
                X                   Not used
                  0 0 0 0           Normal, individual character
                  1 0 0 0           Repetition of previous character
                  0 0 0 1           End of repetition

Byte 3                              Key type (from KBTABLE)
Bit       7 6 5 4 3 2 1 0
          1                         System key. Internal use only
            1                       Repetition ignored
              X X X X               Emulation dependent codes
                          0 0       Alphanumeric key
                          0 1       Short-message key, PA-functions
                          1 0       Function key
                          1 1       Short-message key, PF functions

Byte 4                              Key code (depending on key type):
                                    o Character code if alphanumeric key
                                    o Function code if function key
                                    o Short message code if short-message
                                      key (PA or PF)

Byte 5                              Physical key No.
                                    (Position code)
```

The following function keys always have the same position code:

Key	Code
Cursor left	01
Cursor right	02
Cursor up	03
Cursor down	04
New line	05

Key	Code
Cursor home	06
Back tab	07
Tab	08
Enter	1F

10.4.2 Interface

Declarations:

```
DECLARE BED_$1000 CHAR(5) EXTERNAL;      /* INPUT AREA             */
DECLARE BED_$1900 EVENT EXTERNAL;        /* READY TO RECEIVE       */
DECLARE BED_1910 EVENT EXTERNAL;         /* NEW INPUT DELIVERED    */
```

Call format:

```
ASSIGN BED_$1910;
WAIT BED_$1910;                /*WAIT FOR INPUT                */
/*PROCESS LATEST INPUT IN BED_$1000 AREA */
.
.
POST BED_$1900;                /* READY TO RECEIVE NEW INPUT   */
```

10.5 KEYBOARD REPETITION FREQUENCY

10.5.1 Functional Description

The key repetition frequency is either 12.5 Hz or 25 Hz. The appropriate frequency is normally selected during customizing. However, the frequency can also be defined by means of the externally declared variable BED_$1015.

10.5.2 Interface

Declaration:

```
DECLARE BED_$1015 BYTE EXTERNAL;

BED_$1015 = 0;              /* SET FREQUENCY 25 Hz        */
BED_$1015 = 1;              /* SET FREQUENCY 12,5 Hz      */
```

10.6 CLICK SOUND ACKNOWLEDGEMENT

10.6.1 Functional Description

The click sound acknowledgement of keyboard entries is normally selected during customizing and handled by the keyboard. However, it can also be enabled by means of a call to the OS procedure BED_02400.

The procedure must be called once for each desired acknowledgement.

10.6.2 Interface

Declaration:

```
DECLARE BED_02400 ENTRY;
```

CALL format:

```
CALL BED_2400;
```

10.7 ALARM

10.7.1 Functional Description

The alarm in the keyboard unit can be activated by means of a procedure call. The duration of the signal is defined by the procedure parameter.

10.7.2 Interface

Declarations:

```
DECLARE BED_02600 ENTRY (BYTE VALUE);
DECLARE AL_DUR BYTE;          /* ALARAM DURATION EXPRESSED IN     */
                              /* UNITS OF 20 MS                   */
```

Call format:

```
AL_DUR = 5;                   /* ALARM DURATION 0.1 SEC           */
CALL BED_02600(AL_DUR);
```

10.8 KEYBOARD LAMPS

10.8.1 Functional Description

The eight lamps on the keyboard can be lit up or extinguished by means of a call to the procedure BED_02300. The procedure parameter LAMP_NO is used to define the number of the desired lamp.

10.8.2 Interface

Declarations:

```
DECLARE BED_02300 ENTRY (BYTE VALUE);
DECLARE LAMP_NO BYTE;
```

The parameter LAMP_NO is initialized as follows

Bit	7	6	5	4	3	2	1	0	Meaning
	1								Light lamp No. 8
	0								Extinguish lamp No. 8
				.					.
				.					.
				.					.
								1	Light lamp No. 1
								0	Extinguish lamp No. 1

Call format:

```
LAMP_NO = "suitable value";
CALL BED_02300(LAMP_NO);
```

10.9 MAGNETIC ID INPUT

10.9.1 Functional Description

Input from the magnetic identification device (MID) is handled analogously to keyboard input.

The same memory area (BED_$1000) is used by the system module to receive input. The same event variable (BED_$1910) is used by the Keyboard Handler to indicate that input is transferred to BED_$1000. The event is posted when an ID- card is inserted or removed from the MID.

The information in the area BED_$1000 is interpreted for MID input as follows:

	Bit 7	6	5	4	3	2	1	0	Meaning
Byte 1									Not used
Byte 2	0	1	0	0	0	0	0	0	MID information follows
Byte 3									Not used
Byte 4	1								ID-card reading successful
		1							Parity error in ID-card reading
			1						ID-card removed
				1					ID-card inserted
					X	X			Not used
							1		Faulty ID-card
								X	Not used
Byte 5									Not used

When the event BED_1910 is posted, and byte 2 in BED-$1000 indicates MID information, ID-card reading can be initiated by means of a call to the procedure BED_02500. After reading from the card has been performed, ID data is stored in the BED_$1010 area. The number of characters read into this area is stored in BED_$1012.

If an ID card is inserted into the MID while no input opening has been performed for MID, the alarm is sounded.

If the procedure BED_02500 is invoked with no ID-card inserted into the MID, the "ID-card removed" indication is raised in BED_$1000.

10.9.2 Interface

Declarations:

```
DECLARE BED_$1000 CHAR(5) EXTERNAL;     /* INPUT AREA              */
DECLARE BED_$1010 CHAR(40) EXTERNAL;    /* ID DATA AREA            */
DECLARE BED_$1012 BYTE EXTERNAL;        /* NUMBER OF ID DATA       */
                                        /* CHARACTERS              */
DECLARE BED_$1910 EVENT EXTERNAL;       /* NEW INPUT ARRIVED       */
DECLARE BED_02500 ENTRY;                /*PROC FOR ID-DATA READING*/
```

Call format:

```
ASSIGN BED_$1910
WAIT BED_$1910                          /* WAIT FOR NEW INPUT      */
/*CHECK INPUT TYPE IN BED_$1000*/
.
.
.
CALL BED_02500                          /* READ ID-DATA FROM CARD */
/*PROCESS ID-DATA in BED_$1010*/
.
.
```

10.10 SELECTOR PEN INPUT (DU 4110 only)

10.10.1 Functional Description

Input from the selector pen is treated analogously to keyboard input.

The same memory area (BED_$1000) is used by the system module to receive input. The same event variable (BED_$1910) is used by the keyboard Handler to indicate that input is transferred to BED_$1000. The event is posted when the selector pen hits the display screen.

The information in the BED_$1000 area is interpreted for selector pen input as follows:

	Bit 7 6 5 4 3 2 1 0	Meaning
Byte 1		Not used
Byte 2	0 0 1 0 0 0 0 0	Selector pen information follows
Byte 3	X X X X X X	Hit address, most significant part
Byte 4	X X X X X X	Hit address, least significant part
Byte 5		Not used

If a selector pen hit occurs while no input opening has been carried out for selector pen, the alarm is sounded.

10.10.2 Interface

Declarations:

```
DECLARE BED_$1000                       /* INPUT AREA           */
DECLARE BED_$1910                       /* NEW INPUT ARRIVED    */
```

Call format:

```
ASSIGN BED_1910
WAIT BED_1910                           /* WAIT FOR NEW INPUT   */
/* PROCESS INFO IN BED_$1000*/
.
.
```

10.11 APPLICATION EXAMPLE

The following example illustrates how a keyboard entry editing session can be started and stopped by inserting and removing an ID-card.

```
.
.
DECLARE BED_$1000 CHAR(5) EXTERNAL;     /* INPUT AREA             */
DECLARE BED_$1010 CHAR(40) EXTERNAL;    /* ID DATA AREA           */
DECLARE BED_$1012 BYTE EXTERNAL;        /* NO. OF ID CHARACTERS   */
DECLARE BED_$1900 EVENT EXTERNAL;       /* READY TO RECEIVE INPUT */
DECLARE BED_$1910 EVENT EXTERNAL;       /* NEW INPUT DELIVERED    */
.
.
.
DECLARE BED_02000 ENTRY(BYTE VALUE);    /* OPENING FOR INPUT      */
DECLARE BED_02100 ENTRY(BYTE VALUE);    /* CLOSING FOR INPUT      */
DECLARE BED_02500 ENTRY;                /* READ ID-CARD DATA      */
.
/* LOAD KEYBOARD TABLE INTO MEMORY */
/* AND INITIALIZE KBTAB            */
/* SEE SECTION ON FILE HANDLING    */
.
.
.
CALL BED_02000($(21));        /* OPEN FOR MID INPUT ONLY          */
START:
ASSIGN BED_$1910;             /* READY FOR NEW INPUT              */
POST BED_$1900;
WAIT BED_$1910;
/* CHECK BED_$1000 */
IF "MID input OK"
   THEN
      CALL BED_02500;         /* READ ID-CARD DATA                */
      /* CHECK ID-CARD DATA IN BED_$1010                          */
   ELSE
      /* HANDLE MID INPUT ERROR */
      GOTO START;

/* MID INPUT AND ID-CARD DATA OK */
CALL BED_02100($(01));        /* CLOSE FOR INPUT                  */
CALL BED_02000($(A1));        /* OPEN FOR KB AND MID INPUT        */

EDIT:
ASSIGN BED_$1910;             /* READY FOR NEW INPUT              */
POST BED_$1900;
WAIT BED_$1910;               /*WAIT FOR NEW INPUT                */
/* PROCESS LATEST INPUT */
.
.
.
GOTO EDIT;
```

11 FD UNIT FUNCTIONS AND FILE HANDLING

List of Contents

11.1 GENERAL 3

11.2 FD CONFIGURATIONS 3
11.2.1 System FD 3
11.2.2 Data FD 3
11.2.3 Personal Computer 4

11.3 FILE HANDLING CONCEPTS 5
11.3.1 Physical File Handling 5
11.3.2 Logical File Handling 5
11.3.3 File Handling Software 5

11.4 DATA STRUCTURES 6
11.4.1 Data Sets 6
11.4.2 File Types 7
11.4.3 Authority Levels 7

11.5 FILE HANDLING CONTROL 8
11.5.1 Volume Control Block 8
11.5.2 File Control Block 9
11.5.3 Volume Table 10

11.6 VOLUME ORIENTED COMMANDS 11
11.6.1 Declare Volume 11
11.6.2 CREATE Volume 11
11.6.3 CHANGE Volume 12
11.6.4 DELETE Volume 12
11.6.5 Built-in Functions 13

11.7 FILE ORIENTED COMMANDS 14
11.7.1 Buffer Requirements 14
11.7.2 DECLARE File 15
11.7.3 CREATE File 17
11.7.4 OPEN/CLOSE File 18
11.7.5 READ Record 19
11.7.6 WRITE/REWRITE Record 20
11.7.7 DELETE File 21
11.7.8 LOAD File 21
11.7.9 Built-in Functions 22

11.8 SYNCHRONIZATION 23

11.9 OVERLAY LOADER PROCEDURE 24
11.9.1 Functional Description 24
11.9.2 Interface 24

11.10 LOADING OF CHARACTER GENERATOR 25
11.10.1 Functional Description 25

11.11 EXCHANGEING OF KEYBOARD TABLE 26
11.11.1 Functional Description 26
11.11.2 Interface 26-26

11.1 GENERAL

As described in the section on Hardware Environment, different system units can be used for FD Input/Output.

- FD 4120 is used for 8" diskettes.
- CP 4103 contains one FD drive of the same type as in FD 4120
- FD 4122 is used for 5"1/4 diskettes.
- CP 4104 contains two FD drives of the same type as in FD 4122 (CPR)
- CP 4105 contains two FD drives of the same type as in FD 4122 (CPL)

The diskette software format is presented in Section 4.

All SPL statements concerning file handling are presented in detail in the SPL Reference Manual.

11.2 FD CONFIGURATIONS

At least one FD unit must be included in a cluster. If several FD units are used, one of them is defined to be system-FD and the rest to be data-FDs. FD 4122 may not be used as system FD. FD 4120 consists of the same hardware and is loaded with the same OS whether it is data or system FD. The FD type is not defined until the beginning of the initial program load (IPL) phase.

The same FD OS is loaded into all FD units.

11.2.1 System FD

A system-FD and a data-FD are identical before program loading is performed. Two conditions must be fulfilled to have an FD unit loaded as a system-FD.

- A system diskette must be inserted in the unit.
- The FD unit must receive a system poll from CP.

These conditions must not be fulfilled for more than one FD unit during CP-IPL.

11.2.2 Data FD

The data-FD is used to load and store data and program products, which can be used by all work stations in the cluster.

A 4120 data FD is always loaded via SS3-bus from the system FD. A 4122 data FD is loaded from a special OS-diskette in the data FD itself.

If a reset is performed on the data-FD after OS is loaded, an automatic dump will be carried out on the system volume in the system-FD.

11.2.3 Personal Computer

FD Unit 4122 can be used together with a DU as a stand-alone personal computer workstation. If a PC system volume is loaded into the FD, it is no longer accessible from other DUs than the DU directly connected to this FD via a V.24 interface.

The PC system is not part of Alfaskop System 41 operating system.

The PC system diskettes are briefly presented in section on Diskette Format.

11.3 FILE HANDLING CONCEPTS

Files can be regarded as logical collections of data as well as physical entities on a diskette. Thus, file handling can be divided into logical file handling and physical file handling.

11.3.1 Physical File Handling

All physical file handling is carried out by OS modules in the FD unit. Physical file handling comprises input and output handling of volumes, libraries and simple data sets.

The smallest unit that can be handled by the physical file handling system is a block. A block contains one or more logical records.

11.3.2 Logical File Handling

The logical file handling is performed by software modules in DU and CP.

Logical file handling comprises the handling of volumes, libraries and ordinary files. All record handling is performed by logical file handling modules.

11.3.3 File Handling Software

Physical file handling is performed by the Drive Handler module in FD. The File Manager is also part of the FD OS.

Logical File Handling is performed by the OS module FDIOS (Flexible Disk Input/Output Supervisor) which is loaded into DU and CP. FDIOS serve as an interface between the system modules and the physical file handling system.

FDIOS can be divided into two parts: Basic FDIOS and Extended FDIOS. The extension can be excluded during system generation. All functions presented in this section are part of FDIOS.

The file handling modules all utilize the Communication Handler which is described in section on Internal Communication.

11.4 DATA STRUCTURES

As described in the section on Diskette Format, the information on one diskette is called a volume.

A volume contains:

- A volume label (VOLLAB) containing the volume identification and information about the physical organization of the diskette.
- A volume table of contents (VTOC) containing information on the data sets stored on the diskette
- A number of data sets

11.4.1 Data Sets

There are two classes of data sets stored on a diskette.

- Simple data sets
- Libraries, containing a library table of contents (LTOC) and a collection of library members.

Members are regarded as simple data sets.

A simple data set consists of a number of blocks. The block is the smallest unity which can be handled by the diskette I/O handler in FD.

A block is always of fixed length, and it can contain one or more records.

When a block contains more than one logical record, the data set is said to be blocked, otherwise unblocked.

Records may be stored in packed form (i.e. sequences of identical characters are compressed). A data set containing such records is said to be packed. A packed data set is always blocked.

The data set must be declared as a file in order to be controlled by an SPL-program.

11.4.2 File Types

The file handling systems of Alfaskop System 41 supports the following types of files.

- o Library, type D (i.e. directory).
- o Fixed length record file, type F.
  Contains unpacked data in a number of records.
- o Variable length record file, type V.
  Contains packed data.
- o Absolute file, type A.
  Contains unpacked absolute data (program code).
  Generated by linkage editor.

11.4.3 Authority Levels

When a file is created, it is assigned one of five security levels. The security levels are provided in order to protect the data from unauthorized changing or deleting.

From the operators point of view, the security levels 1-3 are associated with password No. 1, and authority levels 4-5 with password No. 2.

Security level 0 does not require any password authorization.

Members of a library can be assigned higher as well as lower authority than the library itself.

See also the section on Logon and Initialization.

11.5 FILE HANDLING CONTROL

File handling is defined by the user by means of SPL statements and built-in functions operating on files and volumes.

Before any file handling can be performed, the volumes and files must be declared and created. See below.

11.5.1 Volume Control Block

The declaration of a volume results in allocation of a Volume Control Block (VCB) in DU. The VCB is used by the file handling modules of OS, and part of the VCB information can also be obtained by the system modules, by means of built-in functions.

The STATUS byte in VCB has the following meaning:

Bit	7	6	5	4	3	2	1	0	
	1								Error occured. Error type stated in the ERRORTYPE byte
		1							End of file reached.
			X	X	X	X			Not used
							1		Volume is write-protected
								1	Recoverable error occurred

The file control block (FCB) contains an identical STATUS byte. See below.

Analogously to the Task Control Block, the VCB can be regarded as comprising an event control block. The VCB event is posted each time a volume oriented command is executed. The VCB in DU corresponds to a Unit Control Block (UCB) in FD.

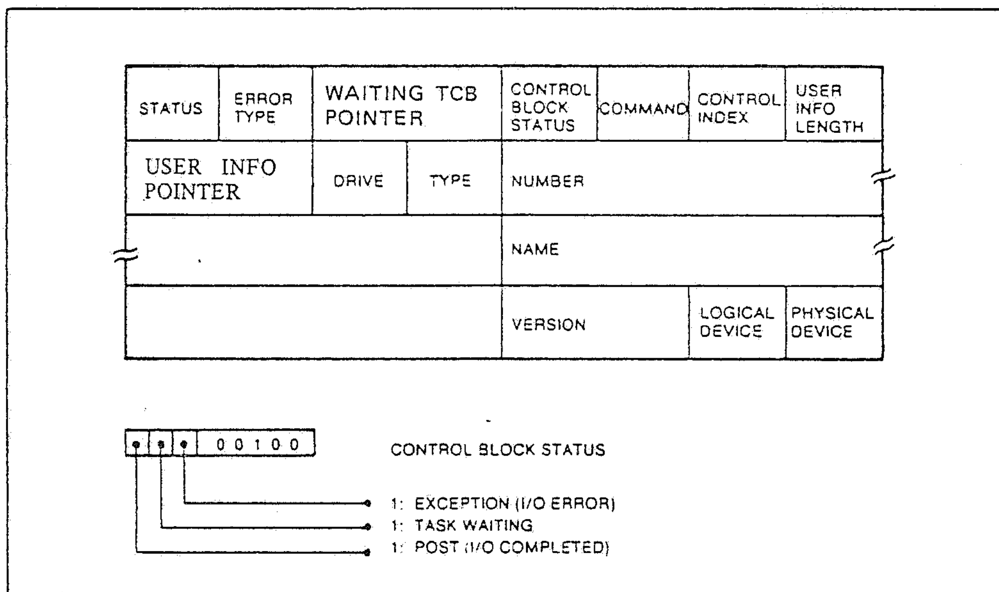


Figure 11.1 Volume Control Block

11.5.2 File Control Block

The declaration of a file results in allocation of a File Control Block (FCB) in DU and FD.

The FCB is used by the file handling modules of OS, and part of the FCB information can also be obtained by the system modules, by means of built-in functions.

The STATUS byte of FCB is the same as in VCB. (See above).

The FCB can be regarded as comprising an event control block. The FCB event is posted each time a file oriented command is executed.

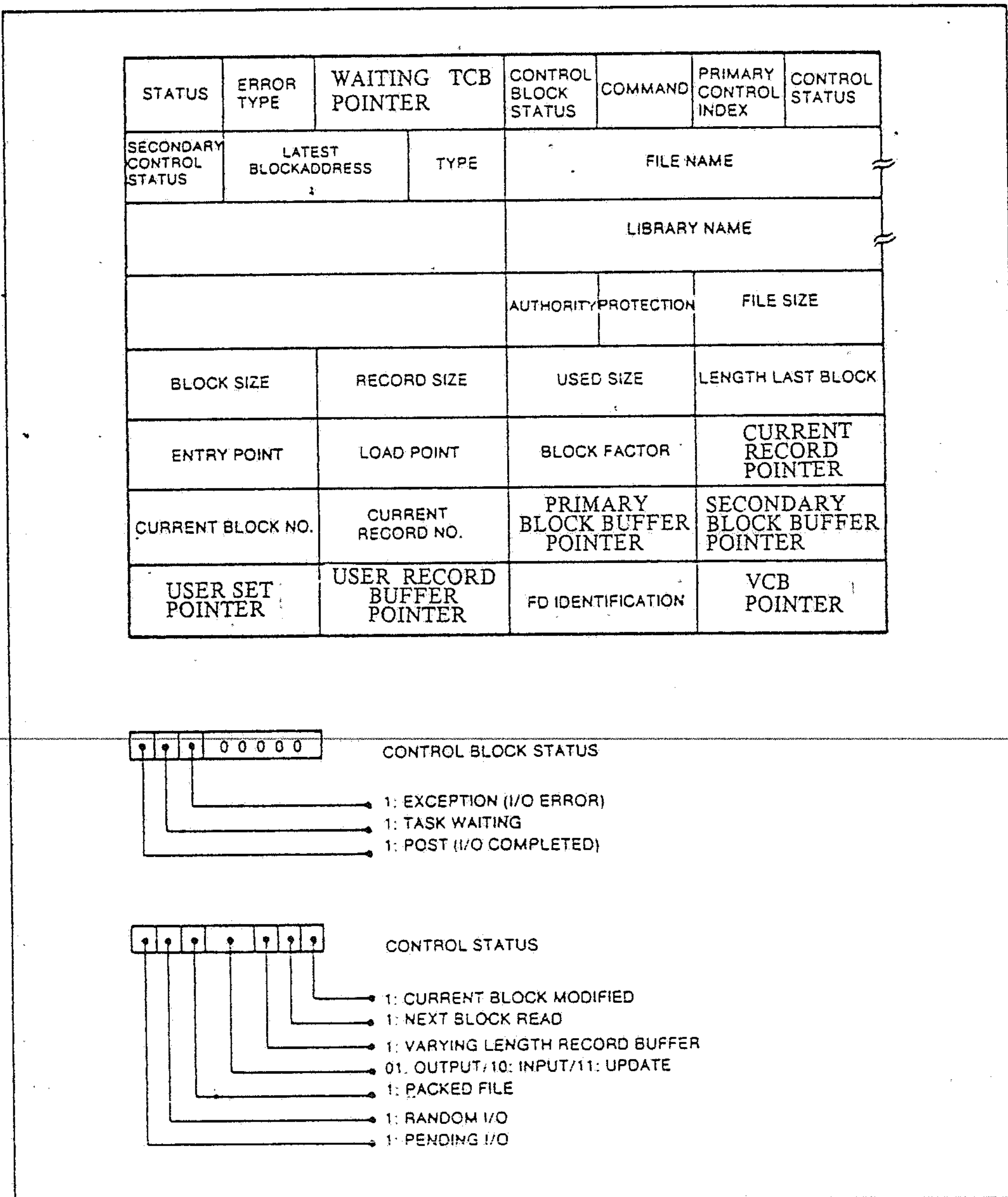


Figure 11.2 File Control Block

11.5.3 Volume Table

The communication processor maintains a table of all mounted volumes in the cluster. A maximum of 64 volumes can be contained in the table. If more than 20 volumes are to be accessed, the CP must be provided with the MRW option.

The entire volume table require about for up to 64 volumes 1 kbyte of main storage. It contains drive numbers and information from the Volume Label of the inserted volumes.

A volume is identified either by its number or by name and version.

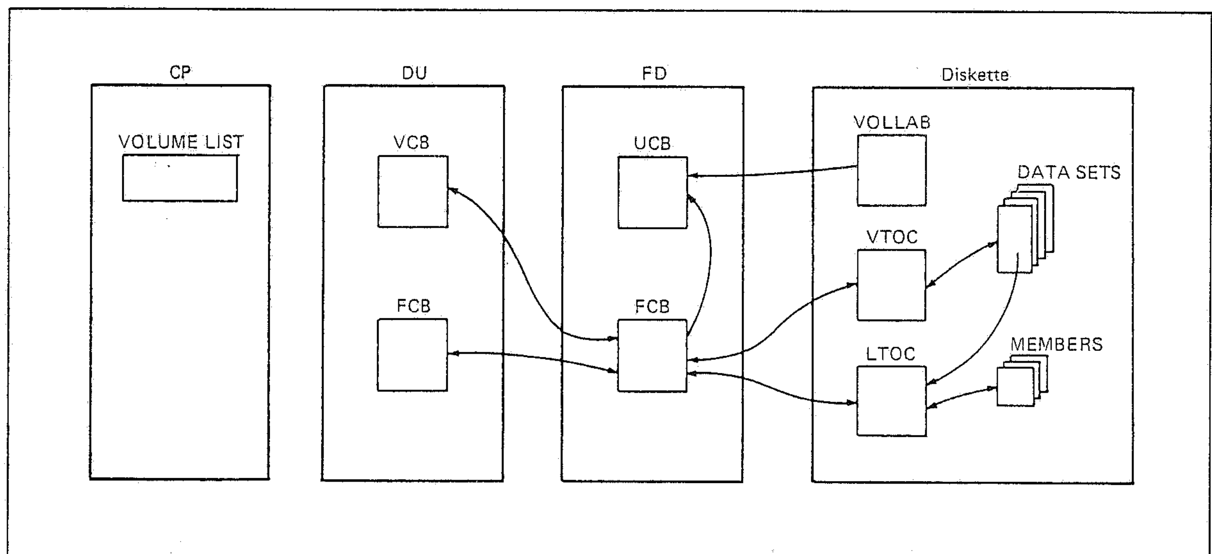


Figure 11.3 Control blocks

11.6 VOLUME ORIENTED COMMANDS

11.6.1 Declare Volume

Before a physical volume (diskette) can be accessed from an SPL program, the volume must be logically defined by allocating and initializing a Volume Control Block (VCB).

Assignment of data to the VCB can be carried out either in the declaration or by a separate ASSIGN statement.

The following parameters can be used in the declaration of a volume: DEVICE, DRIVE, NUMBER, NAME and VERSION.

At least one of the following combinations must be specified:
1) DEVICE and DRIVE
2) NUMBER
3) NAME and VERSION

The following declarations are assumed in the examples below:

```
DECLARE DE   CHAR(4);    /*  LOGICAL ADDRESS OF THE FD UNIT   */
DECLARE DR   BYTE;       /*  DRIVE NO. 1 OR 2                 */
DECLARE NU   CHAR(8);    /*  VOLUME NUMBER                    */
DECLARE NA   CHAR(8);    /*  VOLUME NAME                      */
DECLARE VE   CHAR(2);    /*  VOLUME VERSION                   */
```

Example 1:

```
DECLARE VID VOLUME STATIC ENVIRONMENT (DEVICE(DE) DRIVE(DR)
NUMBER (NU) NAME(NA) VERSION(VE));
```

Example 2:

```
DECLARE VID VOLUME;
ASSIGN VID NAME(NA) DEVICE(DE) DRIVE(DR);
```

In example 2 the declaration only allocates space for the VCB, while the ASSIGN statement causes the VCB to be initialized.

11.6.2 CREATE Volume

An empty volume must be created to initialize VOLLAB and VTOC. This is done either in console mode or by using the CREATE statement as follows:

Example:

```
CREATE VID FILES(FI) TYPE(TY) STAMP(ST) COMMENTS(CO)
BUFFER(BU);
```

Where

FILES defines the maximum (up to 31) number of data sets to be allocated on the volume. This indicates the size of VTOC. FILES must be specified.

TYPE is the volume type. 'S' for system disk or 'D' for data disk.

STAMP specifies a revision date.

COMMENTS is an optional descriptive field containing up to 20 characters.

BUFFER is a user defined area utilized by the Operating System during execution of the CREATE command.

The arguments must have been declared as follows:

```
DECLARE FI   FIXED BIN(15);
DECLARE TY   CHAR(1);
DECLARE ST   CHAR(10);
DECLARE CO   CHAR(20);
DECLARE BU   CHAR(128);
```

11.6.3 CHANGE Volume

The CHANGE statement can be used to modify the following volume I/O options:
NAME
VERSION
STAMP
COMMENTS

Example:
CHANGE VID NAME (VOL_A)

Note that volume NUMBER cannot be changed by the CHANGE command.

11.6.4 DELETE Volume

The statement DELETE VID marks the volume as empty and sets the TYPE byte in the VCB to 'E'.

11.6.5 Built-in Functions

VCB information can be obtained by using the following built-in functions:

```
DEVICE(VID)
DRIVE(VID)
NUMBER(VID)
NAME(VID)
VERSION(VID)
```

Example:

```
DNR = DRIVE(VID)
```

After performing a volume oriented command, the status of the volume control block can be investigated by means of the build-in functions STATUS(VID) and ERRORTYPE(VID).

11.7 FILE ORIENTED COMMANDS

11.7.1 Buffer Requirements

The file I/O commands READ/WRITE/REWRITE is normally performed via a block buffer, since the block is the smallest unit which can be treated by the physical I/O handling procedures.

The block buffer is provided by the user in the file declaration. A pointer into the block buffer is also provided. The buffer pointer is manipulated by means of read and write commands.

If the data set is packed, a block buffer is not sufficient for reading and writing. A record area must also be declared by the user. The record area contains only one record.

If the data set is neither blocked nor packed, and a record area is declared, the block buffer can be omitted. The pointer then refers directly to blocks on the diskette.

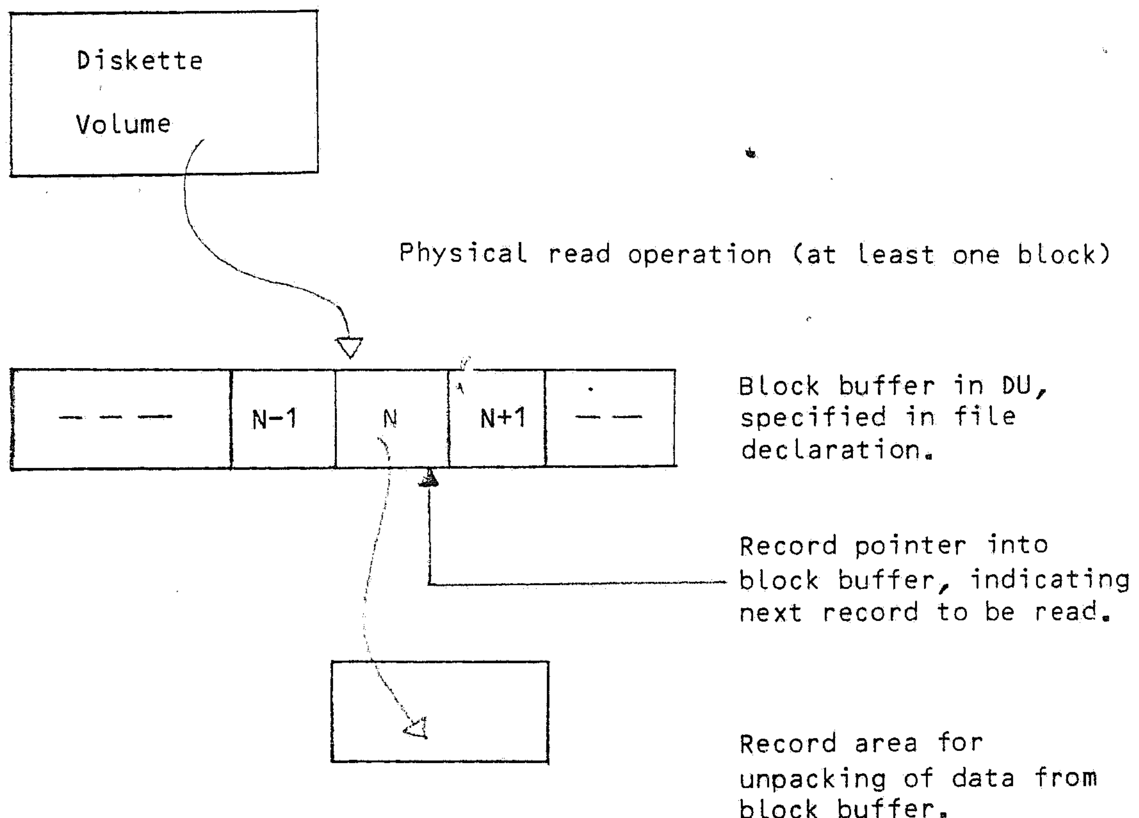


Figure 11.4 Block buffer and record area

11.7.2 DECLARE File

The File Control Block (FCB) is allocated analogously to the VCB. The declaration and initialization of the FCB can be carried out in one single statement where the ENVIRONMENT options depend on the type of data set declared.

The following parameters can be used in the file declarations: VOLUME, BUFFER, SET, LIBRARY, NAME and TYPE.

The following declarations are assumed in the examples below:

```
DECLARE VID  VOLUME;       /* VOLUME ON WHICH THE FILE IS STORED  */
DECLARE BU   CHARACTER(n); /* BLOCK BUFFER FOR INPUT/OUTPUT       */
DECLARE RP   POINTER;      /* RECORD POINTER IN THE BLOCK BUFFER  */
DECLARE LI   CHAR(8);      /* NAME OF LIBRARY                     */
DECLARE NA   CHAR(8);      /* NAME OF MEMBER OR FILE              */
DECLARE TY   CHAR(1);      /* TYPE OF DATA SET                    */
```

Example 1: A volume declared as a single file.

```
DECLARE FID FILE STATIC
ENVIRONMENT (VOLUME (VID) BUFFER (BU) SET (RP));
```

VOLUME — Required. Specifies a VCB.

BUFFER — Optional. Specifies the block buffer area. (See section on Buffer Requirements above).
For sequential access the length of the buffer area, ('n') must be at least twice the blocksize. For random access 'n' must equal the block size.

SET — Optional. Specifies a record pointer, which points to the current record in the block buffer. The pointer must be explicity declared if no record area is provided.

Example 2: VTOC declaration (blocked records)

```
DECLARE FID FILE STATIC
ENVIRONMENT (TYPE ('D') VOLUME (VID) BUFFER (BU) SET (RP));
```

TYPE('D') — Required. Specifies the data set type to be a directory (VTOC).

VOLUME, BUFFER and SET as in example 1. BUFFER, however, is required in this case.

Example 3: LTOC declaration (blocked records)

```
DECLARE FID FILE STATIC
ENVIRONMENT (TYPE ('D') VOLUME (VID) LIBRARY (LI)
BUFFER (BU) SET (RP));
```

LIBRARY(LI) Required. Defines the name of the library.

TYPE, VOLUME, BUFFER and SET as in above examples.

Example 4: Member declaration

```
DECLARE FID FILE STATIC
ENVIRONMENT (NAME (NA) TYPE (TY) VOLUME (VID)
LIBRARY (LI) BUFFER (BU) SET (RP));
```

NAME Required. Defines the member name

TYPE Required. One of the following data set types:

- F Fixed length records file.
- V Variable length records file. The data set will be in packed format, i.e. any string of identical characters will be replaced by a counter and the actual character when stored on disk. When reading or writing, a record area has to be used. Files of type R or V are recognized and treated as files of type V.
- A Absolute file. Used for load modules.

VOLUME, LIBRARY, BUFFER and SET as in example 2. BUFFER, however, is optional in this case.

Example 5: Declaration of other sequential data sets

```
DECLARE FID FILE STATIC
ENVIRONMENT (NAME(NA) TYPE(TY) VOLUME(VID)
BUFFER(BU) SET(RP));
```

NAME Required. Specifies a unique data set name other than a library.

TYPE, VOLUME, BUFFER and SET as in example 4.

Instead of giving the I/O options in the ENVIRONMENT attribute, they can be specified in a separate ASSIGN statement.

Example:

```
ASSIGN FID NAME(NA) TYPE(TY) VOLUME(VID) BUFFER(BU) SET(RP);
```

11.7.3 CREATE File

A declared data set must be created before it can be used.

In the following examples the parameters are declared as follows:

```
DECLARE FI   BIN;
DECLARE BS   BIN;
DECLARE RS   BIN;
DECLARE FS   BIN;
DECLARE SE   BYTE;
DECLARE LP   BIN;
```

Example 1: Create a simple data set

```
CREATE FID BLOCKSIZE(BS) RECORDSIZE(RS)
FILESIZE(FS) SECURITY(SE) LOADPOINT(LP)
```

BLOCKSIZE Required. Specifies the blocksize in number of bytes.

RECORDSIZE Required. Specifies the logical record length in number of bytes. For fixed length records, the blocksize should be a multiple of the record size. For variable records the maximum record length is specified.

FILESIZE Required. Specifies the number of blocks the data set is to contain.

SECURITY Optional. Specifies the level of security. See section on Authority Levels above.

LOADPOINT Optional. Specifies the memory address into which an absolute file is to be loaded.

Example 2: Create a library

```
CREATE FID BLOCKSIZE(BS) FILESIZE(FS)
SECURITY(SE) FILES (FI);
```

Same as in example 1 except for:

FILES Required. Specifies the maximum number of members (up to 104) to be registered in the library.

11.7.4 OPEN/CLOSE File

Before any input or output can be performed on a file, it must be opened. This is done as follows:

Example:
```
OPEN FID     INPUT;
             OUTPUT;
             UPDATE;
```

INPUT allows for read operations only
OUTPUT allows for write operations only
UPDATE only allows changes of existing records

After an OPEN statement is executed, the record pointer points to the first record in the data set.

When a file is opened for output or update, it is in exclusive use, i.e. it can not be accessed by another user until it is closed.

An open file on a volume will cause locking of the FD drive in which the diskette is inserted.

After file input or output is completed, the file must be closed.

Example:
```
CLOSE FID;
```

11.7.5 READ Record

The READ command is a logical command which is associated with a specified record in an open file.

In the declaration of a file, a block buffer is normally defined. A record pointer is associated with the block buffer. (See section on Buffer Requirements above.)

After a READ command is executed, the pointer is updated to refer to the next record in the buffer. If the next record is not present in the buffer, a new block is physically copied into the buffer from the diskette (unless the end of the file is reached).

The READ command can be issued with or without a key, indicating the relative record No. Record No. 1 is the first record of the file.

If the READ command is issued with an INTO (record area) option, the record referred to by the pointer is moved to the specified record area.

Example 1: Read next record

```
READ FID;
```

The record pointer is updated to point to the next record in the buffer. If the pointer was already pointing to the last record in the buffer, a new block is read into the buffer, and the pointer is set to indicate the first record in the new block.

Example 2: Read specified record

```
READ FID KEY(KE);
```

The record pointer will point to the relative record number 'KE' in the file, where number 1 specifies the first record in the file. If the requested record is not available in the buffer area, a new block is fetched from the diskette.

The READ command does not cause a record to be read if used before a block buffer has been defined. It affects KEY, however, and may be used to set the pointer before a WRITE or REWRITE command.

Note. The statement
READ FID KEY(0);
can be used to set the pointer to the end of the file.

Example 3: Read next record into the record area

```
READ FID INTO(RA);
```

Next record will be read into the record area 'RA' from the block buffer or directly from the device.

The record area must be previously declared:
DECLARE RA CHAR(MAX_RL);
or
DECLARE RA CHAR(MAX_RL) VARYING;
where MAX_RL is the maximum record length.

A record area must be used when reading records which are stored in packed format. As soon as the buffer is emptied by continual READ operations, a new block will automatically be read into the buffer.

Example 4: READ specified record into the record area

```
READ FID KEY(KE) INTO(RA);
```

This is a combination of examples 2 and 3. Note that packed files (type V) cannot be read with a key.

11.7.6 WRITE/REWRITE Record

Output of records to the diskette can be performed by either the WRITE or the REWRITE command. Both commands can be used on files which are opened for OUTPUT or UPDATE.

Output to diskette is performed via the same buffer as used for the READ operations (see section on Buffer Requirements above).

Note that keys cannot be used for output operations.

Example 1: Write next record

```
WRITE FID;
```

The record pointer will be updated to point to the next available record area in the current block buffer.

Example 2: Write next record from the record area

```
WRITE FID FROM(RA);
```

The current record in the record area 'RA' will be moved to the next available space in the output buffer or directly to the device. The record pointer is updated to point to the next record in the block buffer.

Example 3: Rewrite last read record

REWRITE FID;

This can be used for a file opened for UPDATE. The record to be written must first have been read into main storage with a READ statement. The corresponding record in the block buffer is marked as changed. The buffer will be rewritten to the file when another block is to be read into the buffer.

Example 4: Rewrite from the record area
REWRITE FID FROM(RA);

The current content in the record area is moved to the block buffer, if there is any. Otherwise a direct rewrite takes place.

Note: If the preceding READ with KEY was performed with neither a block buffer nor an INTO option, this REWRITE statement will cause new data to be written to the relative record position referred to by the current record pointer.

11.7.7 DELETE File

The DELETE statement can be used to delete a file.

Example: DELETE FID;

The file 'FID' will be deleted and the corresponding space on the diskette will be released.

The DELETE command can be performed on closed files as well as on open files.

11.7.8 LOAD File

The LOAD statement loads programs and program segments.

Example: LOAD FID;

The file 'FID' will be loaded into main storage. FID must be declared as an absolute file, type 'A'.

11.7.9 Built-In Functions

The built-in functions STATUS and ERRORTYPE are used to obtain information from the file control block after performing a file oriented command.

STATUS returns the value of the STATUS byte and ERRORTYPE returns the value of the ERROR TYPE byte in FCB. The FCB is presented in section 11.5.2 above.

Example:

```
IF STATUS(FID)  = $(40)       /*  END OF FILE   */
   THEN ...  ;
IF STATUS(FID) >= $(80)       /*  ERROR         */
   THEN
   /* CHECK ERRORTYPE */
```

The built-in function COMPLETION can be used to test if a previously initiated I/O operation has been completed.

When a file is opened, information about its attributes can be obtained by means of special built-in functions.

Examples:

```
BS = BLOCKSIZE(FID);
RS = RECORDSIZE(FID);
FS = FILESIZE(FID);
SE = SECURITY(FID);
LP = LOADPOINT(FID);
SI = SIZE(FID);
```

Refer to the SPL Reference Manual for further details.

11.8 SYNCHRONIZATION

I/O processing is performed by OS tasks parallel to system module execution. Only one I/O operation at a time can be applied to an FCB or a VCB. Thus synchronization of I/O operations is required.

Synchronization is obtained by implicit posting of FCBs and VCBs and by using the WAIT (File) or WAIT (Volume) statements.

Example 1: Wait for event in Volume Control Block

```
WAIT VID EXCEPTION(EX);
```

The calling program is waiting for the OPEN, CLOSE or CREATE operation on the volume operation 'VID' to be finished.

If the operation was successful, the statement following the WAIT statement will be executed, otherwise the statements after the label 'EX' is executed.

The keywords ALL or ANY can be used to wait for all or any of the defined events or tasks to be completed. Refer to the SPL manual.

Example 2: Wait for event in File Control Block

```
WAIT FID EXCEPTION(EX);
```

Analogously to the previous example but for an FCB. Also the operations READ, WRITE, REWRITE, LOAD, CHANGE, CREATE and DELETE results in a posting of the FCB.

11.9 OVERLAY LOADER PROCEDURE

11.9.1 Functional Description

The operating system contains a globally declared procedure (BAD_00220) for loading of overlay segments from the system diskette.

The desired segment is defined in a table which is declared by the user.

The overlay procedure loads the desired segment into the publicly declared file BAD_$0040.

The execution of the loaded segment can be initiated by all call to this file. In the example below, the execution is performed in a task.

11.9.2 Interface

Declarations:

```
DECLARE    BAD_00220 ENTRY (PTR VALUE);  /* OVERLAY LOADER         */
DECLARE    BAD_$0040 FILE EXT;           /* PUBLIC FCB             */
DECLARE  1 FILEDEF,                      /* FILE DEFINITION TABLE  */
           2 LIB CHAR(8),                /* LIBRARY                */
           2 NAME CHAR(8),               /* MEMBER/FILE NAME       */
           2 MAP BYTE,                   /* LOAD MAP NO.           */
           2 LSTAT BYTE,                 /* LOAD STATUS            */
           2 LERRT BYTE;                 /* LOAD ERROR TYPE        */
DECLARE    DEFPTR POINTER STATIC INIT (ADDR(FILEDEF));
DECLARE    NEWPROC TASK;
```

Call format, overlay:

```
LIB  =     "Library name";
NAME =     "file/member name";
MAP  =     "load map No.";          /* =0 IF LOAD MAP NOT USED   */
CALL BAD_00220(DEFPTR);             /* PERFORM OVERLAY           */
IF LSTAT]= 0
  THEN
     /* DO ERROR PROCESSING */
```

Call format, execution:

```
CALL BAD_$0040 TASK(NEWPROC) PRIO(0);      /* EXECUTE NEW PROC     */
,
,
,
WAIT NEWPROC;                    /* WAIT FOR TERMINATION OF NEWPROC  */
```

Note. If the MAP byte is set to $(81), the file is regarded as a character generation table. See below.

11.10 LOADING OF CHARACTER GENERATOR

In DU 4110, the character generator is resident in PROM. In later DU models this table must be loaded from the system diskette.

11.10.1 Functional Description

During IPL of OS in DU, a default character generator is loaded. The default character generator is IBM 24 lines, group A.

The loading of the desired character generator is performed by means of overlay loader BAD_00220, as described above.

If the MAP byte in the file definition table is set to $(81), the specified file is loaded as a character generator.

See section 10.9 above.

11.11 EXCHANGING OF KEYBOARD TABLE

11.11.1 Functional Description

Up to 5 keyboard tables can be defined in a cluster. Each DU is assigned a default keyboard table during customizing. The default keyboard table is loaded by the Logon Handler.

The name of the assigned keyboard table is stored in the globally declared character string BAD_70301.

All keyboard tables are stored in the library KBLIB (or KBLIBA for KBU 4143) on the system diskette. They are named KBTABLE0 to KBTABLE4.

11.11.2 Interface

Declarations:

```
DECLARE KBTABIN FILE
ENVIRONMENT (NAME ('KBTABLEx')            /* DESIRED KBTAB          */
             TYPE ('A')
             LIBRARY ('KBLIB   ')
             VOLUME ("system volume"));

DECLARE 1 KBTABBUF,                       /* KBTABLE WITH HEADER    */
          5 KBSTRAP CHAR(13),             /* KB STRAP DATA          */
          5 KBTABPTR(16) POINTER,         /* TO SUBTABLES           */
          5 KBTAB(n*128) CHAR(2);         /* n = NUMBER OF          */
                                          /* SUBTABLES              */

DECLARE BED_$1100 POINTER EXT;            /* POINTER TO KB STRAP    */
                                          /* DATA                   */

DECLARE BED_02200 ENTRY;                  /* INIT PROC FOR KB       */
                                          /* STRAP DATA             */
```

Call format:

```
CREATE KBTABIN ...;                       /* SEE SECTION ON CREATE  */
OPEN KBTABIN INPUT;                       /* OPEN FOR READING       */
WAIT KBTABIN EXCEPTION (OPEN_FAIL);
READ KBTABIN KEY(1) INTO KBTAB;           /* RECORD SIZE = KBTAB SIZE */
WAIT KBTABIN EXCEPTION (READ_FAIL);
CLOSE KBTABIN EXCEPTION (CLOS_FAIL);

BED_$1100 = ADDR(KBSTRAP);
CALL BED_02200;                           /* INITIALIZE STRAP DATA  */
```

12 PRINTER FUNCTIONS

List of Contents

12.1	GENERAL DESCRIPTION	3
12.2	PRIOS	4
12.2.1	PRIOS Functional Description	4
12.2.2	PRIOS Parameter List	5
12.2.3	PRIOS Event	8
12.2.4	PRIOS TCB	9
12.3	PRINTER STATUS INQUIRY	10
12.3.1	Functional description	10
12.3.2	Interface	10
12.4	PRINT QUEUE REQUEST	12
12.4.1	Functional Description	12
12.4.2	Interface	12
12.5	CANCELLATION OF PRINT QUEUE REQUEST	15
12.5.1	Functional Description	15
12.5.2	Interface	15
12.6	PRINTOUT REQUEST AND RETRY OF PRINTOUT REQUEST	17
12.6.1	Functional Description	17
12.6.2	Interface	18
12.6.3	Hold Printout	20
12.7	PRIOS OPTIONAL FUNCTIONS	21
12.7.1	Functional Description	21
12.7.2	Interface	21
12.8	LOCAL PRINTER STATUS CONTROL	22
12.8.1	Functional Description	22
12.8.2	Interfaces	22
12.9	PRINTER EDITING	23
12.9.1	Functional Description	23
12.9.2	Printer Edit Parameter List	25
12.9.3	Interface	27
12.9.4	Logoff Byte	28
12.10	PRINTER HARDWARE	29
12.10.1	Printer Definition File	29
12.11	APPLICATION EXAMPLES	30
12.11.1	Declarations	30
12.11.2	Initialization at Logon	32
12.11.3	PRIOS Requests	32
12.11.4	Printer Editing Loop	35-35

12.1 GENERAL DESCRIPTION

Output to printer units is handled by two modules in the operating system: the Printer Handler which carries out physical printer handling, and PRIOS which serves as the interface to the system and application modules.

Four main types of commands can be issued to PRIOS:
- o Printer status inquiry
- o Print queue request
- o Cancellation of print queue request
- o Printout request and retry of printout request

The various calls to PRIOS are defined by means of the PRIOS Parameter List, which is initialized by the user.

Synchronization of printer operation is performed by means of a PRIOS Event. The event is always posted by PRIOS to indicate that a requested operation has been completed.

Printer editing is performed by means of an editor which must be linked with the system module. Text to be printed is transferred by PRIOS from the user buffer to a print buffer which is also declared in DU by the user.

From the print buffer, the editor fetches the text character by character, inserts control characters and moves the text continually to the editor buffer. The editor buffer is provided by OS.

From the editor buffer, the Printer Handler transmits the text to the printer unit. See also Fig. 12.2 in section 12.9.

12.2 PRIOS

12.2.1 PRIOS Functional Description

PRIOS itself is declared as the external procedure BID_00100. A call to PRIOS comprise one parameter. The parameter is a pointer to PRIOS Parameter List, which is further described below. PRIOS Parameter List must be properly initialized by the user before any call to PRIOS is issued.

At least one printer editor must be linked with the system module. The present editors are marked in two global variables. The selected editor is specified in PRIOS Parameter List. The function of the editor is controlled by means of a Print Edit Parameter List, which is also initialized by the user. See section on Printer Editing below.

Synchronization of print operations is performed via the PIROS event, declared by the user. This event is posted by PRIOS each time a requested operation (or part of operation) is performed. Thus, the PRIOS event must be assigned by the user before each new call to PRIOS. PRIOS Parameter List contains the pointer to the PRIOS event.

The two parameter lists must not be changed during PRIOS or editor processing.

The figure below illustrates a normal command sequence issued to PRIOS.

System Module	PRIOS
(Initializing performed)	
ASSIGN PRIOSEV; Printer Status Inquiry	
	POST PRIOSEV; Printer OK
ASSIGN PRIOSEV; Print Queue Request	
	POST PRIOSEV; Print queue position indicated
ASSIGN PRIOSEV; (Wait until first in queue)	
	POST PRIOSEV; First in print queue
ASSIGN PRIOSEV; Printout Request	
	POST PRIOSEV; Printout completed

Fig 12.1 Normal sequence in printout operation

12.2.2 PRIOS Parameter List

The PRIOS Parameter List (PRIOSPL) is used to control communication between the system module and PRIOS. Several PRIOS calls may be performed simultaneously by defining one parameter list for each call.

The calls to PRIOS are issued with a pointer to the parameter list as an argument. Note that the content of the parameter list must not be changed before the operation has been performed. Each PRIOS operation is terminated by the posting of a user defined PRIOS event, which contains status and error indications.

PRIOS Parameter List declaration:

```
DECLARE 1 PRIOSPL,                         /* PRIOS PARAMETER LIST     */
         2* CHAR(2)     PTR,   /* INITIATED AS *$(FFFF) OR $(0000)   */
         2 CALLPARM,    PTR,               /* CALL PARAMETERS          */
           3 PRIOSEVP   PTR,               /* PRIOS EVENT POINTER      */
           3 PRIOSOP    BYTE,  /* REQUESTED PRIOS OPERATION            */
           3 TOIDTYPE   BYTE,              /* DESTINATION ID TYPE      */
           3 TOID       BYTE,              /* DESTINATION ID           */
           3 ORDER      BYTE,              /* OPERATION INSTRUCTIONS   */
           3 EDITORID   BYTE,              /* REQUESTED EDITOR         */
         2 MISCPARM,                       /* RETURN PARAMETERS OR     */
           3* CHAR(10);                    /* PRINTER PARAMETERS       */

DECLARE PRIOSPLP PTR STATIC INIT(ADDR(PRIOSPL)); /*POINTER TO PROSPL*/
```

The various identifiers are described below.

Data description for PRIOS Parameter List:

CALLPARM The call parameters must always be specified by the user. They are never changed by PRIOS.

PRIOSEVP Is a pointer to the PRIOS event which is posted one or several times during the PRIOS operation.

PRIOSOP Defines the requested PRIOS operation.

PRIOSOP	Meaning
$(01)	Status inquiry to printer
$(02)	Print queue request
$(03)	Cancellation of print queue request
$(04)	Printout request
$(05)	Retry of printout request

See also section on Hold Printout below.

TOIDTYPE Defines TOID as being a physical or a logical address

TOIDTYPE	Meaning
$(00)	TOID contains physical unit address
$(80)	TOID contains logical unit address

TOID Defines a physical or logical address of the printer to be used. For layout see section on Logical Addresses in Chapter 7.

ORDER Contains supplementary information to PRIOS. Interpreted as follows:

Bit 7	6	5	4	3	2	1	0	Meaning
X	X							Reservation state. See below.
		1						Conditional print queue request. See below.
			0					ETB sequence sent to printer after last character. Printing is marked as completed when the last character has been printed.
			1					No ETB sequence sent to printer. Printing is marked as completed when the last character has been sent to the printer.
				1				Host print. See below.
					X	X	X	Job number. See below.

Reservation State	Meaning
00	Printer is not reserved. The queue element is automatically removed from the print queue when printing is terminated.
01	Queue element is appended on queue. When it has reached the first queue position it can be kept there as long as the user wishes.
10	Queue position is obtained ahead of all queue elements with reservation states 00 or 01, but after the first queue element if the queue is not empty.
11	Absolute reservation. Queue position is obtained ahead of all queue elements except the first one, regardless of reservation state. All queue elements behind are removed. When the queue element has reached the first queue position it is kept there as long as the user wishes.

Conditional Print Queue Request	If the requested printer is not immediately accessible, the request is inhibited. The PRIOS event is posted with STATUS = error, ERRORTYPE = $(0A).
Host print	Defines that the PRIOS request is coming from the host computer. It must have the same value for all PRIOS operations concerning the same printing. Indicates that the user buffer is also print buffer.
Job number	Is a three bit number used to distinguish various PRIOS calls from the same user. It must have the same value for all PRIOS operations concerning the same printing.
EDITORID	Indicates which editor(s) is to be used. Bit mask combinations are possible.
$(80)	Standard editor
$(40)	-
$(20)	Editor for Console Mode
$(10)	Editor for IBM emulation
$(08)	Editor for UNIVAC emulation
$(04)	-
MISCPARM	This field is used in various ways, depending on the desired PRIOS operation. In status inquiries and print queue requests, it contains return parameters from PRIOS. In printout requests, it contains printer parameters to PRIOS.

12.2.3 PRIOS Event

The PRIOS Event is declared by the user. The PRIOS Parameter List contains a pointer to the event.

The PRIOS Event is posted one or several times during the execution of a PRIOS operation. The reason for the latest posting is indicated in the STATUS and ERRORTYPE fields of the event control block (ECB).

The STATUS byte is interpreted as follows:

Bit 7	6	5	4	3	2	1	0	
1								Printer inoperable
	1							Printer inoperable warning
		1						Printer recovered
			1					Error
				1				Warning
					1			Queue number (printout request queued)
						1		Queue ready
							1	Data fetched

Printer inoperable — Indicates that a hardware printer error occurred. The printing has been interrupted.

Printer inoperable warning — Indicates that a less serious hardware printer error occurred. However, the last printing was successful if not the Error bit (see below) indicates another error.

Printer recovered — Indicates that the printer is ready to operate again.

Error — Indicates that a system error occurred, which is specified in ERRORTYPE. Note that the error indication is independent of the printer inoperable indication.

Warning — Indicates a system warning, which is specified in ERRORTYPE. The warning may be ignored by the user. Note that the warning indication is independent of the printer inoperable indication or the printer inoperable warning indication.

Queue number — Indicates that a queue request was successfully queued, and the queue position number entered into the parameter list.

Queue ready — Indicates that the queue element is first in queue.

Data fetched	Indicates that the user text buffer has been emptied by PRIOS and is available to the user. However, the PRIOS operation is not yet terminated. Not used in host print, since print data is already in print buffer.

The ERRORTYPE field of the ECB specifies the occurred error type.

Bit	7	6	5	4	3	2	1	0	Meaning
	X	X	X	X					Not used
					X	X	X	X	Error type number. See below.

Error type number	Meaning
$(00)	-
$(01)	Frequent parity error
$(02)	Communication error
$(03)	Communication connect error
$(04)	Print queue full
$(05)	Print request cancelled
$(06)	Requested editor not available
$(07)	Error in PRIOS call
$(08)	Print queue reserved
$(09)	Print buffer overflow
$(0A)	-
$(0B)	-
$(0C)	Already in print queue
$(0D)	Reserved print ahead
$(0E)	Reserved print put ahead
$(0F)	-

12.2.4 PRIOS TCB

PRIOS is a task which is defined in the task control block BAD_$0250. This TCB is declared globally in OS, but it is not accessible as a normal external entity from the system modules. However, the address of the TCB is fixed to $(8046).

12.3 PRINTER STATUS INQUIRY

12.3.1 Functional description

The current status of any printer in the cluster can be obtained by means of a call to PRIOS.

PRIOS is called with the operation code for status inquiry set in the parameter list. When the operation is completed, the PRIOS event is posted by PRIOS.

After the posting of the event, printer status information is available in the return parameters in PRIOS Parameter List.

12.3.2 Interface

Declarations:

```
DECLARE 1 PRIOSPL,                           /* PRIOS PARAMETER LIST    */
          2* CHAR(2),
             3 PRIOSEVP   PTR,
             3 PRIOSOP    BYTE,
             3 TOIDTYPE   BYTE,
             3 TOID       BYTE,
             3 ORDER      BYTE,
             3 EDITORID   BYTE,
           2 RETPARM,                        /* RETURN PARAMETERS       */
             3  QNO       BYTE,              /* QUEUE POS NO.           */
             3*           CHAR(2),
             3 PRBUFLEN   BIN,               /* LENGTH OF PRINT BUFFER  */
             3 CURREDIT   BYTE,              /* EDITOR USED             */
             3 PRINTDEF,
               4 LINELEN  BYTE,              /* LINE LENGTH             */
               4 PROPT    BYTE,              /* PRINTER OPTIONS         */
               4 PRTYPE   BYTE,              /* PRINTER TYPE            */
               4 PRTABNO  CHAR,              /* PRINTER TABLE           */

DECLARE      PRIOSPLP PTR STATIC INIT(ADDR(PRIOSPL));
DECLARE      PRIOSEV EVENT;
DECLARE      BID_00100 ENTRY (PTR VALUE) /* PRIOS                       */
             OPTIONS ((LOCK(INTERRUPT));
```

PRIOSPLP is the pointer to the parameter list.

Call format:

```
PRIOSEVP = ADDR(PRIOSEV);
PRIOSOP = $(01);                        /* STATUS INQUIRY          */
TOIDTYPE = <unit number type>;
TOID = <unit number>;
ORDER = <only host print bit is relevant>;
ASSIGN PRIOSEV;
CALL BID_00100(PRIOSPLP);
WAIT PRIOSEV;
```

Return parameters:

After PRIOSEV has been posted, the following information is available in the parameter list:

PRBUFLEN	Contains the length of the print buffer.
CURREDIT	Contains the editor(s) currently used. Combinations are possible.
$(80)	Standard editor
$(40)	-
$(20)	Editor for Console Mode
$(10)	Editor for IBM emulation
$(08)	Editor for UNIVAC emulation
$(04)	-

LINELEN	Contains the maximum line lenght for the printer. This value is not checked by PRIOS.
PROPT	Specifies options for the connected printer i.e. sheet feeder.
PRTYPE	Specifies the connected printer type. See section on Printer Hardware below.
PRTABNO	Specifies the current printer table in the DU which has the PU connected. See section on Keyboard Functions.

The user can also obtain information about the operating status of the printer by analyzing the STATUS entry of the PRIOS event.

12.4 PRINT QUEUE REQUEST

12.4.1 Functional Description

Before a request for printout can be issued, a print queue request must be issued and response received.

PRIOS is called with the operation code for print queue request set in the parameter list. PRIOS responds by posting the PRIOS event.

If the STATUS of the PRIOS event is $(04) after a print queue request, the print job is queued properly and the return parameters are available in the parameter list.

After that, the PRIOS event has to be assigned again by the user. When the event is posted a second time, the print job has reached the top of the queue. The STATUS of the PRIOS event is now $(02) if no errors have occurred.

A printout request can then be issued.

If the event posted in response to the print queue request contains STATUS = $(02), the queue request reached the first queue position at once and a printout request can be issued immediately.

If the event status is neither $(04) nor $(02) the operation was not successful. Error type is indicated in the ERRORTYPE byte of the PRIOS Event.

The print queue request can also be issued as a conditional request. this implies that the user wishes to enter the print queue only if it is empty. See section on PRIOS Parameter List above.

12.4.2 Interface

Declarations:

```
DECLARE  1 PRIOSPL,                    /* PRIOS PARAMETER LIST    */
           2* CHAR(2),
           2 CALLPARM,                 /* CALL PARAMETERS         */
             3 PRIOSEVP   PTR,
             3 PRIOSOP    BYTE,
             3 TOIDTYPE   BYTE,
             3 TOID       BYTE,
             3 ORDER      BYTE,
             3 EDITORID   BYTE;
```

```
            2 RETPARM,                         /* RETURN PARAMETERS        */
              3  QNO        BYTE,              /* QUEUE POS NO.            */
              3*            CHAR(2),
              3 PRBUFLEN    BIN,               /* LENGTH OF BUFFER         */
              3 CURREDIT    BYTE,              /* EDITOR USED              */
              3 PRINTDEF,
                4 LINELEN   BYTE,              /* LINE LENGTH              */
                4 PROPT     BYTE,              /* PRINTER OPTIONS          */
                4 PRTYPE    BYTE,              /* PRINTER TYPE             */
                4 PRTABNO   CHAR,              /* PRINTER TABLE            */

DECLARE     PRIOSPLP PTR STATIC INIT(ADDR(PRIOSPL));
DECLARE     PRIOSEV EVENT;
DECLARE     BID_00100 ENTRY(PTR VALUE)  /* PRIOS                           */
            OPTIONS(LOCK(INTERRUPT));
```

Call format:

```
PRIOSEVP = ADDR(PRIOSEV);
PRIOSOP = $(02)                         /* PRINT QUEUE REQUEST             */
TOIDTYPE = <unit number type>;
TOID = <unit number>;
ORDER = <suitable order>;
ASSIGN PRIOSEV;
CALL BID_00100(PRIOSPLP);

WAITQUE:
WAIT PRIOSEV;
IF ((STATUS(PRIOSEV) & $(80)) ]=0)      /* PRINTER INOPERABLE              */
   THEN DO;
   /* DO ERROR PROCESSING */
   /* WAIT FOR PRINTER TO RECOVER */
   ASSIGN PRIOSEV;
   GOTO WAITQUE;
   END;
IF ((STATUS(PRIOSEV) & $(10)) ]=0)
   THEN DO;
   /*COMMUNICATION OR PRINT QUEUE ERROR*/
   IF (ERRORTYPE(PRIOSEV) = $(04))
      THEN /* PRINT QUEUE FULL */
      ELSE
         IF (ERRORTYPE(PRIOSEV) = $(05))
            THEN /* PRINT REQUEST CANCELLED */
            ELSE /* INT. COMMUNICATION ERROR */
                 /* TRY OTHER PRINTER */
            END;
```

```
IF ((STATUS(PRIOSEV) & $(04) ]=0)       /* PRINT REQUEST QUEUED     */
   THEN DO;
   /* PROCESS PARAMETERS IN PARAMETER LIST */
   ASSIGN PRIOSEV;
   GOTO WAITQUE;
   END;
IF ((STATUS(PRIOSEV) & $(02) ]=0        /* FIRST IN QUEUE           */
   THEN DO;
   /* PRINTOUT REQUEST CAN NOW BE ISSUED */
```

Return parameters:

After the PRIOS event has been posted the following parameters are available in the parameter list:

QNO — Contains the position number in the print queue, if the queue was full, the value is $(FF). If the queue request was rejected for some other reason it contains the value 0.

PRBUFLEN — Contains the length of the print buffer. This value must not be exceeded.

CURREDIT — Contains which editor(s) may be used. Combinations are possible. The same bit disposition as in Section 12.4.2.

LINELEN — Contains the maximum line lenght for the printer. This value is not checked by PRIOS.

PRTYPE — Specifies the connected printer type. See section on Printer Hardware below.

PRTABNO — Specifies the current printer table in the DU or PCU which has the PU connected.

PROPT — Specifies options for the connected printer.

12.5 CANCELLATION OF PRINT QUEUE REQUEST

12.5.1 Functional Description

Any element in the print queue can be removed by means of a command to PRIOS.

PRIOS is called with the operation code for cancellation of print queue request set in the parameter list. PRIOS responds by posting the PRIOS event.

If the STATUS of the PRIOS event is 0, the operation was successful and the queue element was removed or not found. There is no other indication to show if a queue element was not found. If the event status is not 0 the operation might have gone wrong.

A queue element is removed from the queue regardless of its queue position or reservation class. If the queue element was first in queue and printing was going on, the printing is aborted.

12.5.2 Interface

Declarations:

```
DECLARE     1 PRIOSPL,                   /* PRIOS PARAMETER LIST    */
              2* CHAR(2),
              2 CALLPARM,                /* CALL PARAMETERS         */
                3 PRIOSEVP PTR,
                3 PRIOSOP BYTE,
                3 TOIDTYPE BYTE,
                3 TOID BYTE,
                3 ORDER BYTE;

DECLARE     PRIOSPLP PTR STATIC INIT(ADDR(PRIOSPL));
DECLARE     PIROSEV EVENT;
DECLARE     BID_00100 ENTRY (PTR VALUE) /* PRIOS                    */
            OPTIONS(LOCK(INTERRUPT));
```

Call format:

```
PRIOSEVP = ADDR(PRIOSEV));
PRIOSOP = $(03);                        /* CANCELLATION OF PQ REQUEST  */
TOIDTYPE = <unit number type>;
TOID = <unit number >;
ORDER = <suitable order>;
ASSIGN PRIOSEV;
CALL BID_00100(PRIOSPLP);
WAIT PRIOSEV;
IF ((STATUS(PRIOSEV) & $(10) ] = 0        /* COMMUNICATION ERROR      */
   THEN DO;
   /* DECIDE IF TO DO A*/
   /* RETRY OR ISSUE AN */
   /* OPERATOR MESSAGE */
   ELSE
   /* QUEUE REQUEST CANCELLED */          /* AS DESIRED               */
   END;
```

12.6 PRINTOUT REQUEST AND RETRY OF PRINTOUT REQUEST

12.6.1 Functional Description

When the print queue request has reached the top of the print queue, a printout request can be issued.

PRIOS is called with the operation code for printout request or retry of printout request set in the parameter list. PRIOS responds by posting the PRIOS event.

If the PRIOS event STATUS = $(01), the content of the user buffer is transferred to the print buffer, but printing is not yet completed. The user must then assign the PRIOS event again and wait for a new posting of the event.

When host print is performed, the posted event normally has the STATUS 0 since the user buffer is also used as print buffer, and thus no transfer takes place.

After the PRIOS event has been posted a second time, the event STATUS is 0 if printing has been completed without errors. The queue element is removed from the print queue if the element had reservation class 0.

If the event status is neither $(01) nor $(00), the operation might have been unsuccessful.

In the case of a retry of printout request, the print buffer already contains the desired text since it was transferred from the user buffer at the previous request. Consequently the event STATUS is 0 if the operation was successful.

12.6.2 Interface

Declarations:

```
DECLARE     1 PRIOSPL,                    /* PRIOS PARAMETER LIST     */
              2* CHAR(2),
              2 CALLPARM,                 /* CALL PARAMETERS          */
                3 PRIOSEVP PTR,
                3 PRIOSOP BYTE,
                3 TOIDTYPE BYTE,
                3 TOID BYTE,
                3 ORDER BYTE,
                3 EDITORID BYTE,
              2* CHAR,
              2 PRPARM,                   /* PRINTER PARAMETERS       */
                3 UBUFSTRT PTR,           /* POINTER TO USER BUFFER   */
                3 UTEXTLEN BIN,           /* LENGTH OF TEXT TO BE     */
                                          /* PRINTED                  */
                3 EDITPARM,               /* EDITOR PARAMETERS        */
                  4 EDPARM CHAR(5)

DECLARE     PRIOSPLP PTR STATIC INIT(ADDR(PRIOSPEL));
DECLARE     PRIOSEV EVENT;
DECLARE     BID_00100 ENTRY(PTR VALUE)    /* PRIOS                    */
            OPTIONS(LOCK(INTERRUPT));
```

Call format:

```
PRIOSEVP = ADDR(PRIOSEV);
PRIOSOP = $(04);                          /* PRINTOUT REQUEST         */
TOIDTYPE = <unit number type>;
TOID = <unit number>;
ORDER = <suitable order>;
EDITORID = <suitable editor>;
UBUFSTRT = <user buffer start address>;
UTEXTLEN = <text length>;
EDITPARM = <user defined parameters>;
ASSIGN PRIOSEV;
```

```
PRIOCALL:
CALL BID_00100(PRIOSPLP);
WAITBUFF:
WAIT PRIOSEV;
IF ((STATUS(PRIOSEV) & $(80) ]= 0)       /* PRINTER INOPERABLE       */
   THEN DO;
   /*DO ERROR PROCESSING, FOR EXAMPLE*/
   /*ISSUE MESSAGE ON MESSAGE LINE AND*/
   /*LIGHT I/O ERROR LAMP*/
   /*WAIT FOR PRINTER TO RECOVER*/
   ASSIGN PRIOSEV;
   GOTO WAITBUFF;
   END;
IF ((STATUS(PRIOSEV & $(20)) ]= 0)       /* PRINTER RECOVERED        */
   THEN DO;
   /*DO RECOVERY PROCESSING,*/
   /*ERASE TEXT ON MESSAGE LINE AND*/
   /*TURN OFF I/O ERROR LAMP*/
   ASSIGN PRIOSEV;
   GOTO PRIOCALL;
   END;
IF ((STATUS(PRIOSEV) & $(10)) ]= 0)      /* PRINTER ERROR            */
   THEN DO;
   /*DO ERROR PROCESSING, FOR EXAMPLE*/
   /*TRY OTHER PRINTER*/
   END;
/*BUFFER MOVED (BIT 0 IN STATUS(PRIOSEV) SET)*/
/*START PRINTOUT*/
ASSIGN PRIOSEV;
WAITPRNT:
WAIT PRIOSEV;
IF ((STATUS(PRIOSEV) & $(80) ]= 0)       /* PRINTER INOPERABLE       */
   THEN DO;
   /*WAIT FOR PRINTER TO RECOVER*/
   ASSIGN PRIOSEV;
   GOTO WAITPRNT;
   END;
IF ((STATUS(PRIOSEV & $(20)) ]= 0)       /* PRINTER RECOVERED        */
   THEN DO;
   /*DO RECOVERY PROCESSING*/
   /*DO A RETRY OF PRINTOUT REQUEST*/
   PRIOSOP = $(05);                      /*RETRY OF PRINTOUT REQUEST*/
   ASSIGN PRIOSEV;
   CALL BID_00100(PRIOSPLP);
   GOTO WAITPRNT;
   END;
IF ((STATUS(PRIOSEV) & $(10)) ]= 0)      /* PRINTER ERROR            */
   THEN DO;
   /*DO ERROR PROCESSING*/
   /*TRY OTHER PRINTER*/
   END;
/*PRINTING COMPLETED*/
.
.
```

12.6.3 Hold Printout

The hold printout function defines the actions to be carried out if a printer becomes inoperable.

The actions are specified in the PRIOSOP byte in PRIOS Parameter List for a printout request. (Note that only bits 3-0 are used to define the desired operation type).

PRIOSOP

Bit 7	6	5	4	3	2	1	0	
				X	X	X	X	PRIOS operation request. See section 12.2.2.
			1					If the printer becomes inoperable, no New Line or Form Feed is added
		1						Hold function enabled. If the printer becomes inoperable, the printout is not aborted. Instead PRIOS waits until the printer is operable again, and the printout is then continued.
X	X							Not used.

If the hold function is invoked, PRIOS event is posted with STATUS = "printer inoperable" at the interrupt and with STATUS = "printer recovered" when the printer is operable again.

The hold function can be used to temporarily suspend a progressing printout by setting the printer unit off line. When the printer is set on line again, the printout is continued as if it had never been interrupted.

Of course this is not possible if the printer unit erases its own buffer when it is set off line.

12.7 PRIOS OPTIONAL FUNCTIONS

12.7.1 Functional Description

A number of special functions in PRIOS have been gathered into an optional PRIOS module. This module must be linked to the system if any of those functions are required.

The presence of the optional module must be indicated in the globally declared variable BID_$0310 and the entry point for the module must be stated in the globally declared variable BID_$0320. When the optional module is no longer used, for example at logoff, BID_$0310 must be set to zero.

The optional module contains the following functions:

- Absolute reservation of printer (reservation state 3). See the Section on PRIOS parameter list.
- Removal of all other queue elements when an element with reservation state 3 is put into the print queue.
- Message to all users of removed queue elements that the element has been removed.
- Message to all users of queue elements which have been bypassed by a queue element with reservation state 2.
- Message to all print queue users that the printer is inoperable.
- Message to all print queue users that the printer is ready to operate again after it has been inoperable.

12.7.2 Interface

Declarations:

```
DECLARE BID_$0310 BYTE EXT; /* PRESENCE OF PRIOS OPTIONAL MODULE  */
DECLARE BID_$0320 PTR EXT;  /* ENTRY POINT OF THE MODULE          */
```

BID_$0310 can have the following contents:

```
$(00)    PRIOS optional module not present
$(80)    PRIOS optional module present
```

Initialization:

```
BID_$0310 = $(80);
BID_$0320 = <PRIOS optional module entry point>;
```

12.8 LOCAL PRINTER STATUS CONTROL

12.8.1 Functional Description

The user can obtain local printer status information via a call from PRIOS to a user written routine. This function can only be used within the DU which has the V24-printer connected.

PRIOS issues a call to the special user written routine each time the printer status is changed. The entry point of the user routine is stored in the globally declared variable BID_$0270.

If this facility is to be used, the entry point must be set at initialization time. If it is not used, the variable must be set to zero.

The current status of the connected printer unit is indicated in the globally declared variable BID_$0280.

12.8.2 Interface

Declarations:

```
DECLARE BID_$0270 PTR EXT;          /* USER ROUTINE ENTRY POINT    */
DECLARE BID_$0280 BYTE EXT;         /* CURRENT PRINTER STATUS      */
```

BID_$0280 can have the following contents:

```
$(80)   Printer inoperable
$(40)   Printer inoperable warning
$(00)   Printer operable
```

Call format initialization:

```
BID_$0270 = <entry point of user status routine>;
```

12.9 PRINTER EDITING

12.9.1 Functional Description

Data to be printed is edited by a system module, the editor, which is executed as a task. The editor adds printer control characters to the text before it is transmitted to the printer unit by the Printer Handler.

An internal buffer, the print buffer, must be declared by the user in DU. The start address and length of this buffer must be entered into the global variables BID_$0250 and BID_$0260.

PRIOS moves the print data from the user buffer to the print buffer before editing commences.

The purpose of the print editor is
- o To read out the print buffer content, character by character
- o To intersperse the appropriate printer control characters
- o To transfer the edited text to the editor buffer, which is a small memory area provided by the operating system.

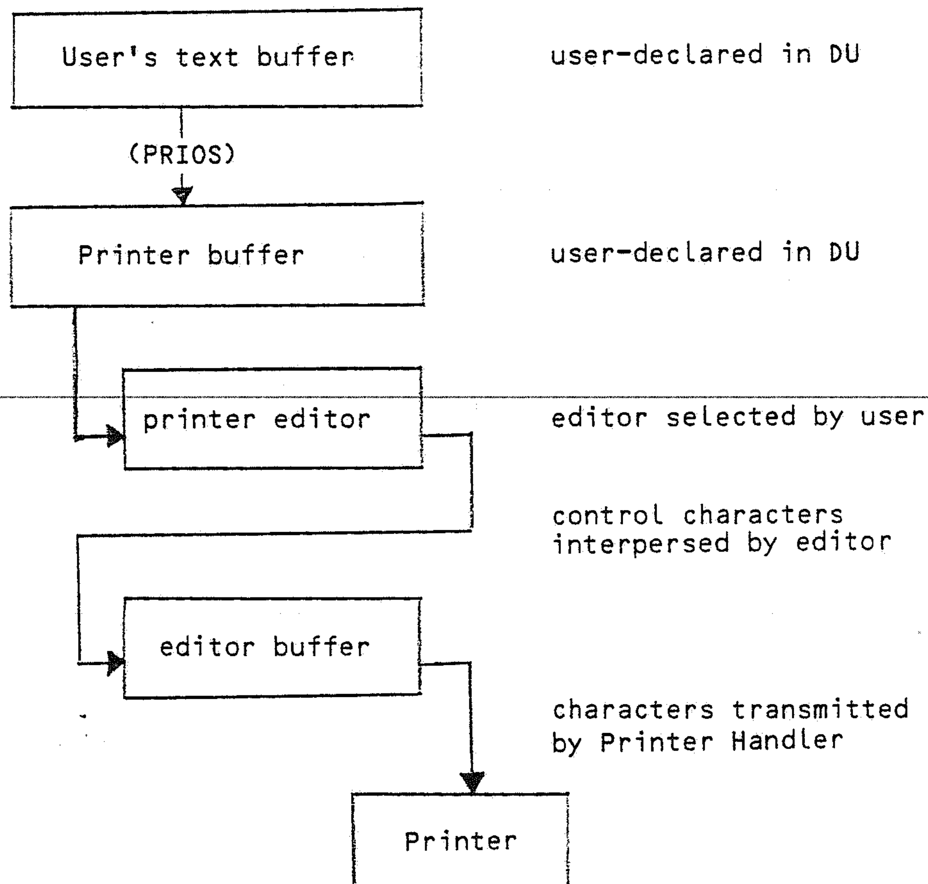


Fig 12.2 Data transmission in printout operations

The system permits several editors, but only one at a time. The different editors have an identical interface to PRIOS except for the contents of the EDITPARAM entry of the Print Edit Parameter List, which are transparent to PRIOS. The Print Edit Parameter List is described below.

When a system module containing a printer editor is loaded, the presence of the editor must be specified in a certain globally declared byte. BID_$0240 is used for the emulation editors and BID $0241 for the standard editor. When the editor no longer is used, the byte must be set to zero.

When there is text in the print buffer, PRIOS posts an event variable, BID_$0220, to activate the editor. The editor fills the editor buffer cyclically, and investigates after each character whether the editor buffer is full. In such case, the event variable BID_$0230 is posted and the editor waits for a new activation signal from PRIOS.

The Printer Handler continually fetches characters from the editor buffer. If the editor has filled the buffer and entered the wait state, PRIOS activates the editor again when half the buffer is released.

Each time the editor is activated, PRIOS provides information in the EDITSTAT entry in the Print Edit Parameter List on whether editing is to start or continue. The editor uses the same entry to inform PRIOS on whether editing shall continue or is terminated.

Some simple editing is performed by PRIOS when transferring text from the user text buffer to the print buffer:

The character New Line, $(15), is always converted to Carriage Return plus Line Feed (CR LF).

The codes $(C0) - $(DF) are converted to $(00) - $(1F) respectively (i.e. $(C0) to $(00), $(C1) to $(01) etc).

12.9.2 Printer Edit Parameter List

The printer Edit Parameter List is declared in the global structure BID_$0210. It is used to control communication between the editor module and PRIOS.

Declarations:

```
DECLARE  1 BID_$0210 EXT,
           2 PRPARM,                    /* PRINTER AND EDITOR PARAMETER */
             3 PRBUSTRT    PTR,
             3 TEXTLEN     BIN,
             3 EDITPARM(5) BYTE,
           2 EBUFPARM,                  /* EDITOR BUFFER PARAMETERS     */
             3 EBUFFSTRT   PTR,
             3 EBUFEND     PTR,
             3 EBUFLEN     BYTE,
             3 EBUFINP     PTR,
             3 *  CHAR (2),
             3 EBUFBCNT    BYTE,
           2 PREDITOR,                  /* EDITOR PARAMETERS            */
             3 EDISTAT     BYTE,
             3*            PTR,
             3*            BYTE,
           2 PRPHYS,                    /* PRINTER UNIT PARAMETERS      */
             3 PRTYPE      BYTE,
             3*            BYTE,
             3 LINELEN     BYTE,
             3 PAGESIZE    BYTE,
```

The following parameters are set by PRIOS:

PRBUSTRT A pointer containing the first address of the print buffer, as declared by the user.

TEXTLEN Length of text to be edited in the print buffer.

EDITPARM Parameters to the editor, which have been transferred unchanged from the user PRIOS call. The meaning of the parameters depends on the editor used.

EBUFSTRT A pointer containing the start address of the editor buffer.

EBUFEND A pointer containing the end address of the editor buffer. This pointer must be used to indicate when the input pointer should be set to EBUFSTRT.

EBUFLEN Maximum number of characters in the editor buffer.

EBUFCNT A byte counter indicating the number of characters left in the editor buffer. After each character is entered into the buffer the editor must increment the counter. PRIOS decrements the counter for each character it gets from the buffer. Note that in some cases this counter can exceed EBUFLEN.

EDITSTAT This byte is set by PRIOS at each editor call and indicates the editing phase to the editor:

EDITSTAT	Meaning
$(FF)	Beginning of the editing job
$(00)	Continuation of the same editing job

The following parameters are set by the editor:

EBUFINP A pointer in the editor buffer to the next character to be edited. Cyclic updating must be performed by the editor after each character entered. The pointer is initialized by PRIOS.

EDITSTAT This byte is set by the editor after each editing phase and indicates to PRIOS when all text in the print buffer has been edited.

EDITSTAT	Meaning
$(00)	Continue editing
$(FF)	Editing terminated

12.9.3 Interface

Declarations for editing:

```
DECLARE  BID_$0210 EXT;      /* PRINT EDIT PARAMETER LIST          */
(As set forth above)
          .
          .
          .
DECLARE BID_$0250 PTR EXT;              /* ADDRESS OF PRINT BUFFER */
DECLARE BID_$0260 BIN EXT;              /* LENGTH OF PRINT BUFFER  */
DECLARE PRBUFPTR;                       /* POINTER IN PRINT BUFFER */
DECLARE BID_$0220 EVENT EXT;            /* ACTIVATE EDITOR         */
DECLARE BID_$0230 EVENT EXT;            /* EDITOR BUFFER FULL      */
DECLARE BID_$0240 BYTE EXT;             /* EDITOR IDENTITY         */
```

Call format for editing:

```
/*INITIALIZATION*/
BID_$0240 = <suitable emulation editor>;
BID_$0250 = ADDR(PRBUFF);
BID_$0260 = <length of print buffer>;

EDIT:
ASSIGN BID_$0220;            /* MAJOR LOOP START                    */
WAIT BIT_$0220;              /* WAIT FOR PRIOS TO ACTIVATE EDITOR   */
IF EDITSTAT = $(FF)          /* EDITING JOB START                   */
   THEN
      PRBUFPTR = PRBUSTRT;

PROCEED:                      /* MINOR LOOP START                   */
EDCHAR = PRBUFPTR → PRCHAR;   /*GET CHARACTER FROM PRINT BUFFER     */
PRBUFPTR = PRBUFPT + 1;
IF PRBUFPTR - PRBUSTRT + 1 > TEXTLEN    /* ENTIRE PR BUF EDITED     */
THEN
      GOTO READY;
/* PUT NEW CHARACTER IN EDITOR BUFFER*/
EBUFINP = EBUFINP + 1;
IF EBUFINP > EBUFEND         /* END OF EDITOR BUFFER                */
   THEN
      EBUFINP = EBUFSTRT;

EBUFCNT = EBUFCNT + 1;       /* UNDER LOCK(KERNEL)                  */
IF EBUFCNT = EBUFLEN         /* EDITOR BUFFER FULL                  */
   THEN
      GOTO BUFFULL;

GOTO PROCEED;                /* MINOR LOOP END                      */
```

```
BUFFULL:                          /* EDITOR BUFFER FULL           */
POST BID_$0230;
GOTO EDIT;

READY:
/*EDITING COMPLETED*/
EDITSTAT = $(FF);
POST BID_$0230;
GOTO EDIT;                        /* MAJOR LOOP END               */
```

12.9.4 Logoff Byte

The system module must not be logged off when the printer editor is processing. The globally declared byte BID_$0245 indicates whether logoff is allowed or not.

BID_0245 content.

```
Bit   7 6 5 4 3 2 1 0       Meaning
      1                     One or more print jobs in print queue.
        1                   Printing in progress. Logoff not allowed.
          X X X X X X       Not used.
```

Declaration:

```
DECLARE BID_$0245 BYTE EXTERNAL;
```

12.10 PRINTER HARDWARE

One printer unit can be connected to each DU or PCU in the cluster. The connection is established via a V.24 interface. The entire cluster configuration definition, including printer configuration is carried out during system customizing.

12.10.1 Printer Definition File

The printer definition file (PRDEF) is a file in library SYSLIB on the system diskette. In PRDEF the printer parameters for each printer in the cluster are defined.

The byte PRTYPE in PRDEF defines the printer type, i.e. the printer hardware.

The PRTYPE byte value is interpreted as follows:

PRTYPE	PROD.NR	PRINTER	COMMENT
01 (hex)	4154	OKI DP100	
	(4153	Facit 4540)	Normally PRTYPE 07
03	4155	OKI DP 125	
04	4152	OKI ML82A	
	(4151	OKI ML83A)	Normally PRTYPE 05
05	4151	OKI ML83A	
06	4156	Facit 4565	Standard; traktor-feed
07	4153	Facit 4540	
08	4156	Facit 4565	BDT sheet feeder, 1 mag.
0A	4157	Facit 4544	4-colours
0B	4160	Facit 4570	Standard
0D	3576	GE 340	
0E	4160	Facit 4570	Standard; sheet feeder, 2 mag.

12.11 APPLICATION EXAMPLES

12.11.1 Declarations

The following is an example of how the PRIOS and print editor parameters can be declared in a system module. Note the DEFINE declaration of RETPARM.

```
/* DECLARATIONS*/
DECLARE 1 PRIOSPL,           /* PRIOS PARAMETER LIST                  */
          2* CHAR(2),
             3 PRIOSEVP   PTR,
             3 PRIOSOP    BYTE,
             3 TOIDTYPE   BYTE,
             3 TOID       BYTE,
             3 ORDER      BYTE,
             3 EDITORID   BYTE,
          2  RETPARM,                  /* RETURN PARAMETERS         */
             3  QNO       BYTE,        /* QUEUE POS NO.             */
             3*           CHAR(2),
             3 PRBUFLEN   BIN,         /* LENGTH OF PRINTBUFFER     */
             3 CURREDIT   BYTE,        /* EDITOR USED               */
             3 PRINTDEF,
               4 LINELEN  BYTE,        /* LINE LENGTH               */
               4*         BYTE,
               4 PRTYPE   BYTE,        /* PRINTER TYPE              */
               4 PRTABNO  CHAR,        /* PRINTER TABLE             */

DECLARE 1 PRPARM DEF RETPARM,          /* FOR PRINTOUT REQUEST      */
          2* CHAR,
          2  UBUFSTRT     PTR,
          2  UTEXTLEN     BIN,
          2  EDITPARM,
             3 EDPARM     CHAR(5);

DECLARE     PRIOSPLP PTR STATIC INIT(ADDR(PRIOSPL));
DECLARE     PRIOSEV EVENT;
DECLARE     BID_0100 ENTRY (PTR VALUE)  /*PRIOS                      */
            OPTIONS ((LOCK(INTERRUPT));
```

```
DECLARE 1 BID_$0210 EXTERNAL,          /* PRINT EDIT PARAM LIST            */
          2 PRPARM,                    /* PRINTER AND EDITOR PARAMETERS */
            3 PRBUSTRT     PTR,
            3 TEXTLEN      BIN,
            3 EDITPARM(5)  BYTE,
          2 EBUFPARM,                  /* EDITOR BUFFER PARAMETERS        */
            3 EBUFSTRT     PTR,
            3 EBUFEND      PTR,
            3 EBUFLEN      BYTE,
            3 EBUFINP      PTR,
            3 * CHAR (2),
            3 EBUFBCNT     BYTE,
          2 PREDITOR,                  /* EDITOR PARAMETERS               */
            3 EDISTAT      BYTE,
            3*             PTR,
            3*             BYTE,
          2 PRPHYS,                    /* PRINTER UNIT PARAMETERS         */
            3 PRTYPE       BYTE,
            3*             BYTE,
            3 LINELEN      BYTE,
            3 PAGESIZE     BYTE,

DECLARE BID_$0220 EVENT EXTERNAL;           /* ACTIVATE EDITOR          */
DECLARE BID_$0230 EVENT EXTERNAL;           /* ED BUFFER FULL           */
DECLARE BID_$0240 BYTE EXTERNAL;            /* EDITOR IDENTIFICATION    */
DECLARE BID_$0250 POINTER EXTERNAL;         /* POINTER TO PRBUFFER      */
DECLARE BID_$0260 BIN EXTERNAL;             /* LENGTH OF PRBUFFER       */
DECLARE PRBUFFER CHAR(2000);                /* PR BUFFER                */
DECLARE BID_$0245 BYTE EXTERNAL;            /* LOGOFF BYTE              */
DECLARE PRBUFPTR POINTER;                   /* POINTER IN PRBUFFER      */
DECLARE PRCHAR CHAR BASE(PRBUFPTR);         /* TO TAKE CHARACTER OUT    */
DECLARE EDCHAR BYTE;                        /* TO PUT CHARACTER IN      */
```

12.11.2 Initialization at Logon

```
/*INITIALIZATION TO BE DONE AT LOGON TIME*/

BID_$0240 = $(10);                /* IBM EDITOR PRESENT            */
BID_$0250 = ADDR(PRBUFFER);
BID_$0260 = 2000;
```

12.11.3 PRIOS Requests

```
/*OPERATOR DEPRESSES PRINT KEY*/
/*KEYBOARD IS LOCKED*/
/*PROCEDURE START*/
/*QUEUE REQUEST FOR PRINTER PR01*/

PRIOSEVP = ADDR (PRIOSEV);
PRIOSOP = $(02);                  /* PRINT QUEUE REQUEST           */
TOIDTYPE = $(80);                 /* LOGICAL                       */
TOID = PR01;                      /* PRINTER UNIT NO               */
ORDER = 0;                        /*RESERVATION STATE 0, JOB NO 0  */
EDITORID = $(10);                 /* SELECT IBM EDITOR             */
ASSIGN PRIOSEV;
CALL BID_00100 (PRIOSPLP);              /* CALL PRIOS              */
QUEUWAIT:
WAIT PRIOSEV;
IF ((STATUS(PRIOSEV) & $(80)) ]= 0)     /* PRINTER INOPERABLE      */
   THEN DO:
      /*PUT "PRINTER INOPERABLE" ON MESSAGE LINE*/
      /*LIGHT I/O ERROR LAMP*/
      /*WAIT FOR PRINTER TO RECOVER*/
      ASSIGN PRIOSEV;
      GOTO QUEUWAIT;
      END;
IF ((STATUS(PRIOSEV) & $(10)) ]= 0)     /* PRINTER ERROR           */
   THEN DO;
      /*PUT "PRINTER ERROR" ON MESSAGE LINE*/
      /*LIGHT I/O ERROR LAMP*/
      /*TRY OTHER PRINTER*/
      GOTO SECPRINT;
      END;
IF ((STATUS(PRIOSEV) & $(04)) ] = 0     /* PRINT REQUEST QUEUED    */
   THEN DO:
      /*EVALUATE QUEUE POSITION*/
      IF QNO > 3
        THEN
         /*QUEUE TO LONG TRY OTHER PRINTER*/
         GOTO SECPRINT;
      ASSIGN PRIOSEV;
      GOTO QUEUEWAIT; /*WAIT FOR QUEUE ENTRY REACH FIRST POSITION  */
      END;
```

```
/*FIRST IN QUEUE(EVENT STATUS = $(01))*/

PRIOSOP = $(04);                             /* PRINTOUT REQUEST          */
UBUFSTRT = <user buffer start address>;
UTEXTLEN = 1920;
EDPARM =<command string for IBM editor>;
ASSIGN PRIOSEV;

PRIOCALL:
CALL BID_00100(PRIOSPLP);
WAITBUFF:
WAIT PRIOSEV;
IF ((STATUS(PRIOSEV) & $(80)) ] = 0)       /* PRINTER INOPERABLE        */
   THEN DO:
      /*ISSUE "PRINTER INOPERABLE" ON MESSAGE LINE*/
      /*LIGHT I/O ERROR LAMP*/
      /*WAIT FOR PRINTER TO RECOVER*/
      ASSIGN PRIOSEV;
      GOTO WAITBUFF;
      END;
IF ((STATUS(PRTIOSEV) & $(20)) ] = 0       /* PRINTER RECOVERED         */
   THEN DO;
      /*ERASE TEXT ON MESSAGE LINE*/
      /*TURN OFF I/O ERROR LAMP*/
      /*CALL PRIOS AGAIN*/
      ASSIGN PRIOSEV;
      GOTO PRIOCALL;
      END;
IF ((STATUS(PRIOSEV) & $(10)) ] = 0)       /* PRINTER ERROR             */
   THEN DO;
      /*PUT "PRINTER ERROR" ON MESSAGE LINE*/
      /*LIGHT I/O ERROR LAMP*/
      /*TRY OTHER PRINTER*/
      GOTO SECPRINT;
      END;

/*BUFFER MOVED*/
/*START PRINTOUT*/
ASSIGN PRIOSEV;
WAITPRNT:
WAIT PRIOSEV;

IF((STATUS(PRIOSEV) & $(80))  ]= 0)        /* PRINTER INOPERABLE        */
   THEN DO;
      /*PUT "PRINTER INOPERABLE" ON MESSAGE LINE*/
      /*LIGHT I/O ERROR LAMP*/
      /*WAIT FOR PRINTER TO RECOVER*/
      ASSIGN PRIOSEV;
      GOTO WAITPRNT;
      END;
```

```
IF ((STATUS(PRIOSEV) & $(20)) ] = 0)     /* PRINTER RECOVERED         */
   THEN DO;
      /*ERASE TEXT ON MESSAGE LINE*/
      /*TURN OFF I/O ERROR LAMP*/
      /*DO A RETRY OF PRINTOUT REQUEST*/
      PRIOSOP = $(05);                   /* RETRY OF PR OUT REQUEST */
      ASSIGN PRIOSEV;
      CALL BID_00100(PRIOSPLP);
      GOTO WAITPRNT;
      END;
IF ((STATUS(PRIOSEV) & $(10)) ] = 0)     /* PRINTER ERROR             */
   THEN DO;
      /*PUT "PRINTER ERROR" ON MESSAGE LINE*/
      /*LIGHT I/O ERROR LAMP*/
      /*TRY OTHER PRINTER*/
      GOTO SECPRINT;
      END;
/*PRINTING COMPLETED WITHOUT ERRORS*/
.
.
.
.
.
.
/* TRY OTHER PRINTER */
SECPRINT:
PRIOSOP = $(03);              /* CANCEL QUEUE REQUEST ON PR01      */
ASSIGN PRIOSEV;
CALL BID_00100(PRIOSPLP);
WAIT PRIOSEV;                 /* WAIT FOR COMPLETED CANCELLING     */
/*FILL IN PARAMETER LIST FOR PRINTER PR02*/
PRIOSOP = $(02);              /* PRINT QUEUE REQUEST               */
TOID = PR02;                  /* STILL LOGICAL                     */
/*CONTINUES AS FOR PRINTER PR01*/
.
.
.
/* BUFFER MOVED */
/* START PRINTOUT */
ASSIGN PRIOSEV;
WAITPRINT:
WAIT PRIOSEV;
```

12.11.4 Printer Editing Loop

```
EDIT:
ASSIGN BID_$0220;             /* MAJOR LOOP START                      */
WAIT BID_$0220;               /* WAIT FOR PRIOS TO ACTIVATE EDITOR     */
IF EDITSTRT = $(FF)
   THEN                       /* EDITING JOB START                     */
   PRBUFPTR = PRBUSTRT;

PROCEED:                      /* MINOR LOOP START                      */
EDCHAR = PRBUFPTR → PRCHAR;   /* GET CHARACTER FROM PRINT BUFFER       */
PRBUFPTR = PRBUFPTR + 1;
IF PRBUFPTR - PRBUSTRT + 1 > TEXTLEN    /* ALL CHARACTERS EDITED?      */
   THEN
    GOTO READY;
EBUFINP → PRCHAR = EDCHAR;    /* PUT CHARACTER IN EDITOR BUFFER        */
EBUFINP = EBUFINP + 1;
IF EBUFINP > EBUFEND          /* LAST EDITOR BUFFER POSITION           */
   THEN
    EBUFINP = EBUFSTRT;

/*LOCK KERNEL TO PREVENT BUFFER COUNTER*/
/*TO RUN OUT OF SYNCHRONIZATION*/
LOCK (KERNEL);
EBUFBCNT = EBUFBCNT + 1;
UNLOCK (KERNEL);

IF EBUFBCNT = EBUFLEN         /* EDITOR BUFFER FULL                    */
   THEN
    GOTO EBUFFULL;
/*GET NEXT CHARACTER*/
GOTO PROCEED;                 /* MINOR LOOP END                        */

EBUFFULL:
/*EDITOR BUFFER FULL*/
POST BID_$0230;
GOTO EDIT;

READY:
/*MARK EDITING COMPLETED*/
EDITSTRT = $(FF);
POST BID_$0230;
GOTO EDIT;                    /* MAJOR LOOP END                        */
```

13 TIMER FUNCTIONS

List of Contents

13.1 GENERAL TIMER DESCRIPTION 3

13.2 GET TIMER 4
13.2.1 Functional Description 4
13.2.2 Interface 4

13.3 RELEASE TIMER 5
13.3.1 Functional Description 5
13.3.2 Interface 5

13.4 APPLICATION EXAMPLE 6
13.4.1 Error Handling 7
13.4.2 Application Example 7-7

13.1 GENERAL TIMER DESCRIPTION

Timing in software execution is carried out by procedures in the Timer Handler.

The Timer Handler comprise real time clocks which can be utilized as time-out counters. The user reserves a time counter by invoking a global function. The counting is started by assigning the counter a time interval. When the time has expired, an event is posted by the Timer Handler.

An activated counter can be reset by assigning it the value 0. The resetting causes no event to be posted.

During counting, the amount of time left can be read from a specified memory cell. The address of this cell is returned by the timer reservation function.

There are two types of timer counters. One is used for time intervals specified in seconds. The other is for shorter intervals specified in units of 20 milliseconds. For both types, the maximum number of time units is 255.

The following counters are available to the user:

In CP:
6 counters using time units of 20 ms
2 counters using time units of 1 s

In DU:
4 counters using time units of 20 ms
2 counters using time units of 1 s

Reserved counters can be released by the user.

13.2 GET TIMER

13.2.1 Functional Description

A time counter is reserved by a call to the function BFG_01500. Two parameters are passed with the call. The first parameter specifies the desired time unit, the other parameter specifies the address of the time-out event.

If the address of the time-out event is 0, no event will be posted, but the counter can still be used for time measurements.

The function returns a pointer to the cell from which the remaining time interval can be read. The initial value is 0. The counter is activated by assigning this cell the desired time interval.

If all available time counters of the desired type was already reserved, the function returns the address FFFA hexadecimal, and the time-out event is cancelled.

Note that the parameters in the function call are not checked by the operating system.

13.2.2 Interface

Declarations:

```
DECLARE BFG_01500 ENTRY (BYTE VALUE, PTR VALUE)
RETURNS (POINTER);
DECLARE TIMEV EVENT;                         /* TIME-OUT EVENT          */
DECLARE TIMEVP POINTER VALUE STATIC INIT (ADDR(TIMEV));
DECLARE TIMCNTP POINTER;                     /* POINTER TO COUNTER CELL */
DECLARE TIMCNT BYTE BASED TIMCNTP            /* TIME COUNTER            */
```

Call format:

```
TIMCNTP = BFG_01500 (<constant>, TIMEVP); /* RESERVE TIMER          */
TIMCNT  = <number of time units>;         /* ACTIVATE TIMER         */
```

<constant> specifies the the desired time unit
$(01): time unit 20 milliseconds
$(02): time unit 1 second

13.3 RELEASE TIMER

13.3.1 Functional Description

The number of time counters is limited, but a reserved timer can be released after being used. (All timers must be released before the logoff event is posted. This releasing is performed by the operating system).

A timer is released by a call to the procedure BFG_01600. Two parameters are passed with the call. The pointers are the same as used in the reservation of the time counter.

The release procedure is optional and must be linked with the system module if desired.

13.3.2 Interface

Declarations:

```
DECLARE BFG_01600 ENTRY (BYTE VALUE, PTR VALUE);
DECLARE TIMEV EVENT;                         /* TIME-OUT EVENT            */
DECLARE TIMEVP POINTER VALUE STATIC INIT (ADDR(TIMEV));
```

TIMEV and TIMEVP are declared in connection with the reservation of the timer. See above.

Call format:

```
CALL BFG_01600 (<constant>, TIMEVP);
```

13.4 APPLICATION EXAMPLE

```
DECLARE MSGIN EVENT;
DELCARE TIMEOUT EVENT;
DECLARE TIMEVP POINTER STATIC INIT(ADDR)TIMEOUT));
DECLARE TIMCNTP POINTER;
DECLARE TIMCNT BYTE BASED TIMCNTP;
DECLARE EVENTNR BYTE;

DECLARE BFG_01500 ENTRY(BYTE VALUE, PTR VALUE) RETURNS(POINTER);
DECLARE BFG_01600 ENTRY BYTE VALUE, PTR VALUE);
.
.
.
TIMCNTP = BFG_01500($(02), TIMEVP);  /* RECERVE TIMER TIME UNIT     */
.                                    /* 1 SEC                       */
TIMCNT  = 3;                         /* COUNT 3 SEC                 */
WAIT MSGIN, TIMEOUT ANY(EVENTNR);    /* WAIT FOR MSGIN BUT NOT      */
                                     /* MORE THAN 3 SEC             */
IF EVENTNR = 0                       /* MSGIN POSTED                */
  THEN DO;
  .
  .
  .
  END;
ELSE
  DO;
  ASSIGN TIMEOUT;
  .
  .
  .
  END;
.
.
.
```

If the event pointer = 0, it means that no event variable will be posted for a time-out.

Example:

```
DECLARE TIMEOUT EVENT;
DECLARE EVPTR PTR STATIC INIT(ADDR(TIMEOUT));
DECLARE TOPTR PTR;
DECLARE TIMER BYTE BASED TOPTR;
DECLARE BFD_01500 ENTRY(BYTE VALUE, PTR VALUE) RETURNS(PTR);
.
.
.
TOPTR = BFD_01500($(02), EVPTR);     /* GET 1-SEC TIMER             */
.
.
.
```

13.4.1 Error Handling

If there is no unoccupied time counter cell, the pointer that points to time counter cell becomes FFFA and the event variable is cancelled if POST has been requested.

The call parameters do not undergo reasonableness checks.

13.4.2 Application Example

```
DECLARE MSGIN EVENT;
DECLARE TIMEOUT EVENT;
DECLARE EVPTR PTR STATIC INIT(ADDR)TIMEOUT));
DECLARE CNTPTR PTR;
DECLARE TIMER BYTE BASED TOPTR;
DECLARE I BYTE;
DECLARE BFD_01500 ENTRY(BYTE VALUE, POINTER VALUE);
RETURNS (POINTER);
.
.
.

TOPTR = BFD_01500($(02), EVPTR);
.
.
.
TIMER = 3;                          /* SET TIMER = 3 SEC           */

WAIT MSGIN, TIMEOUT ANY(1);         /* BUT NOT MORE THAN 3 SEC     */

IF 1 = 0                            /* MESSAGE GOT                 */
  THEN
    DO;
      ASSIGN TIMEOUT;
    .
    .
    .
END;
  ELSE                              /* TIMEOUT                     */
    DO;
    .
    .
    .
.
.
```

14 ERROR HANDLING

List of Contents

14.1	GENERAL CONCEPTS	3
14.2	STATUS	3
14.3	EXCEPTION	4
14.4	ERRORTYPE	5
14.5	ERRORTYPE INDICATIONS FROM OS	6
14.5.1	Error Numbers from CP	6
14.5.2	Error Numbers from Drive Handlers	6
14.5.3	Error Numbers from Input/Output Modules	7
14.5.4	Error Numbers from File Manager	7-9

14 ERROR HANDLING

14.1 GENERAL CONCEPTS

Error and status control is generally achieved by using the built-in functions STATUS, EXCEPTION and ERRORTYPE after each executed operation (or set of operations).

The built-in function must always be declared by the user. These declarations are however omitted in the program examples in this manual.

All control blocks (ECB, TCB, VCB, FCB) contain a STATUS byte and an ERRORTYPE byte. These bytes are changed when operations, such as POST (event) or CLOSE (file), are performed on the control blocks.

The content of these fields can be checked by using the built-in functions STATUS and ERRORTYPE.

Refer to the SPL Reference Manual for detailed description of the functions.

See also sections on Event Control Block, Task Control Block, Volume Control Block and File Control Block.

14.2 STATUS

The built-in function STATUS returns the value of the STATUS byte in the control block specified in the parameter. The parameter must be of type Event, Task, Volume or File.

If STATUS is used whithout an argument, it is regarded as a psedo variable. The pseudovariable assigns a bit string to the executing task control block.

Example 1:

```
DECLARE STATUS BUILTIN;
DECLARE FILE1 FILE;
DECLARE EVENT1 EVENT;
.
.
IF STATUS(FILE1) = '01010000'B
  THEN
    STATUS(EV1) = $(0C);          /* TO ECB                        */
  ELSE STATUS = $(08);            /* TO TCB                        */
```

14.3 EXCEPTION

The built-in function EXCEPTION can be used either in IF - statements or in Volume/File handling statement.

The function returns a value > 0 if an operation on a task, event, volume or file was terminated abnormally.

When used in IF - statements, the control block must be specified (ECB, TCB, FCB or FCB). When used in Volume/File commands, the control block is specified in the statement and the argument of EXCEPTION indicates a label from which execution is conturned of EXCEPTION occurred.

Example 2:

```
DECLARE EXCEPTION BUILTIN;
DECLARE FILE2 FILE;
.
.
.
OPEN FILE2 INPUT;
IF EXCEPTION (FILE2) > 0             /* ALT 1                        */
  THEN
   CALL ERROPEN;
.
.
.
CLOSE FILE2 EXCEPTION (ERRCLOSE);    /* ALT 2                        */
.
.
.
ERRCLOSE:                            /* LABEL                        */
```

14.4 ERRORTYPE

The built-in function ERRORTYPE returns the value of the ERROTYPE byte in the specified control block (ECB, TCB, VCB or FCB). This value is > 0 if the specified control block has an error indicated in its STATUS byte.

If used without a parameter, ERRORTYPE is regarded as a pseudo-variable. The pseudo-variable assigns a bit string to the executing task control block.

Example 3:

```
DECLARE ERRORTYPE BUILTIN;
DECLARE VOL1 VOLUME;

IF ERRORTYPE(VOL1) = $(04)
  THEN
    ERRORTYPE = $(0F)                    /* TO TCB              */
    CALL ERRPROC,:
.
.
.
```

14.5 ERRORTYPE INDICATIONS FROM OS

14.5.1 Error Numbers from CP

Error number	Error type	Cause of error/recommended action
7B	Connection error	The requested unit is not connected. Check the two-wire connection.
7C	Transmission error	Bad two-wire communication. Check the two-wire connection.
7D	Volume not inserted	Check volume number
7E	No S41 Volume	Initiate the volume
7F	Unformatted volume	Format and initiate the volume

If an error occurs, the file is closed.

14.5.2 Error Numbers from Drive Handlers

The following error numbers are available from the FD unit. These error numbers are always sent to the accessing unit together with a status byte with a value >=$(80).

Error number	Error type	Cause of error/recommended action
01	Seek error	The track has been destroyed. The diskette has to be reformatted and copied again.
02	CRC error	The data on this sector(s) are not valid. Rewrite the information.
03	ID/Record not found	The track has been destroyed. The diskette has to be reformatted and copied again.
04	Lost data	Hardware error or program failure.
05	Write fault	Hardware error or program failure.
06	Write protected disk at write	Program failure.
07	Seek error during restore	Hardware error.

Error number	Error type	Cause of error/recommended action
08	Drive not ready	Hardware error or program failure.
09	Time out	Hardware error or program failure.
20	Too high track number	Program failure.
31	Buffer not available	FD temporary overloaded. Try again.

14.5.3 Error Numbers from Input/Output Module

Error number	Error type	Cause of error/recommended action
43	Volume not mounted	Check the volume number.
44	IPL-volume not mounted	The mounted system diskette is not the same as when the FD was loaded.

14.5.4 Error Numbers from File Manager

Error number	Error type	Cause of error/recommended action
A0	Duplicate name and type at change or create	A file/member with the same name and type already exists. Choose another name.
A1	Wrong specification of volume or library at create	Check number of files, file size etc.
A2	The VTOC/LTOC is full	There is no unused or released file descriptor element left. If possible delete a file member big enough for the new one. In other cases create the file on another diskette.
A3	Error in VTOC/LTOC	For example; the information in the Free Space Element is not correct. May be caused by program failure. Create a new diskette and try to do a logical copy of every file/library.

Error number	Error type	Cause of error/recommended action
A4	Unsufficient free space on the diskette	If possible delete a file member big enough for the new one. In other cases create the file on a new diskette.
A5	Wrong type of file	The file type is not an alphabetical one or it is a type D for a member.
A6	No free Space Element in VTOC	May be caused by program failure. Create a new diskette and try to do a logical copy of every file/library.
C0	No file control block available	Temporary overload. Try again.
C1	Volume access conflict	The volume is already accessed by another user. The different types of accesses are in conflict with each other.
C2	Write protected volume	If the diskette is write protected the commands, open, update, delete, change and create are not allowed.
C3	Open type conflict	The volume/file/member is already opened by another user. The different open types are in conflict with each other. For example open input and update at the same time.
C4	Unallowable commands for a "not empty diskette"	For example; create volume for an already created volume.
C5	Library access conflict	The library is already accessed by another user. The different types of accesses are in conflict with each other.
C6	Unallowable commands for an "empty diskette"	For example create file before the volume is created.
C7	Library not found	Check the library name.
C8	Wrong command	Program failure.
C9	Wrong command	Program failure.

Error number	Error type	Cause of error/recommended action
CB	Error at F.SYSBOT read (IPL)	For example: a) No buffer is available for reading. Try again. b) The track or a sector has been destroyed. Rewrite the destroyed data. In worst case the diskette has to be reformatted before re-write.
CC	Internal error	Program failure.
CE	Error at LTOC read	See error number CB.
CF	Member not found	Check number type and name.
D0	Wrong command for a data set of type D	Check data set type and command.
D1	Unsufficient authority	The authority is not high enough for executing this command.
D2	Illegal command with respect to open type	For example: a write command for a file that is opened just for input.
D3	Illegal command for a VTOC/Volume after it has been opened for input or update	Close the volume before executing the command.
D4	Bounding error	Read or write outside the allocated area.
D6	File type error at load and IPL	The file type for a load or IPL command must be A.
DC	Block size error at write	The unit which wants to write data on the diskette does not use the same blocksize as in the file descriptor element.
DD	No memory available at load/IPL	Temporary overload. Try again.
DE	F.SYSBOT is busy	Temporary overload. Try again.
DF	File not found	Check file name and type.

A 2.1 EXTERNAL ENTITIES IN DU

Identifier	Section	Description
BAD_00220	11.9	PROCEDURE for overlay segment loading
BAD_70301	10.2.2	Buffer containing name of the assigned keyboard table
BAD_$0040	11.9	FCB used by BAD_00220 for overlay segment loading
BAD_$0250	12.2.4	PRIOS TCB
BCD_70112	8.9.4	EVENT indicating changed DU status
BCG_08000	8.2	User Interface Module (UIM) in DU
BCG_70104	8.9.4	Structure containing pointer to device status area
BCG_70107	8.3	User Interface Control Block (UICB) in DU
BCG_70113	8.11.3	Buffer for CP/DU/PU status
BCP_70112	8.9.4	EVENT indicating changed PU status
BED_02000	10.3.1	PROCEDURE for opening for DU input
BED_02100	10.3.1	PROCEDURE for closing for DU input
BED_02300	10.8	PROCEDURE for handling lamps on keyboard
BED_02400	10.6	PROCEDURE which enables keyboard clock sound acknowledgement
BED_02500	10.9.1	PROCEDURE for ID data reading
BED_02600	10.7	PROCEDURE which enables keyboard alarm
BED_07000	9.4	PROCEDURE for cursor handling
BED_07700	9.6.4	PROCEDURE for base colour switching in DU 4112
BED_$1000	10.1	Buffer for KB/MID/SP input to the system module. (See also 10.4.1 for KB, 10.9.1 for MID, 10.10.1 for SP.)
BED_$1010	10.9.1	Buffer for ID data input
BED_$1012	10.9.1	BYTE indicating number of characters available in BED_$1010
BED_$1100	10.2.2	POINTER to the keyboard table header

Identifier	Section	Description
BED_$1015	10.5	BYTE indicating KB repetition frequency
BED_$1900	10.1.1	EVENT indicating that the system module is ready to receive a new input character in BED_$1000
BED_$1910	10.1.1	EVENT indicating that a new input character has been transmitted to BED_$1000
BED_$7002	9.4	Buffer for cursor handling parameters
BED_$7100	9.3	Buffer containing initialization parameters to display area adaptation circuitry
BED_$7200	9.2.2	Default display area. Start address 7830
BED_$7210	9.2.2	Last character in default display area. Address 7FAF
BED_$7400	9.3	POINTER to the buffer BED_7100
BED_$7701	9.6.4	BYTE indicating present colour mode in DU 4112
BFG_01500	13.2	PROCEDURE which reserves a timer
BFG_01600	13.2	PROCEDURE which releases a reserved timer
BGD_00110	9.5.3	PROCEDURE for message line handling
BGD_$0020	9.5.3	EVENT indicating completed input on message line
BID_00100	12	PROCEDURE for printer handling (PRIOS)
BID_$0210	12.9	Buffer for printer edit parameter list
BID_$0220	12.9	EVENT indicating that there is text in the print buffer waiting for the print editor (activate editor)
BID_$0230	12.9	EVENT indicating that the editor buffer is full
BID_$0240	12.9	BYTE indicating printer editor identity
BID_$0245	12.9.4	BYTE indicating whether or not logoff is permitted

Identifier	Section	Description
BID_$0250	12.9	POINTER containing address to print buffer
BID_$0260	12.9	BINARY indicating length of print buffer
BID_$0270	12.8	POINTER to user-written procedure for local printer status control
BID_$0280	12.8	BYTE indicating printer status
BID_$0310	12.7	BYTE indicating presence of PRIOS optional module
BID_$0320	12.7	POINTER to the entry point of PRIOS optional module
BUFFEREM	5.2.1	Buffer for messages (See also 9.5)
MAXRAM	9.6.2	Variable indicating size of RWM in DU
OSPNTD	8.11.3	POINTER to user-written DU procedure which will be invoked each time host system status changes
OSPNTP	8.11.3	POINTER to user-written PU procedure which will be invoked each time host system status changes
PROMFUNC	9.6.1	Buffer containing DU type identifier

A 2.2 EXTERNAL ENTITIES IN CP

Identifier	Section	Description
BCC_08000	8.2	User Interface Module in CP
BCC_70105	8.9.3	POINTER to device status area in CP
BCC_70107	8.3	User Interface Control Block in CP
BCC_70109	8.10.2	POINTER to user-written abort procedure
BCC_70110	8.10.2	BYTE indicating simple device status
BCC_$0910	8.11.2	BYTE indicating host system status

INDEX

A

acknowledgement 8.6
active state 1.4.3
adaptation circuitry 9.3
answer to poll 7.4.2
ASSIGN event 1.4.2, 6.3.6
attach task 1.4.1, 6.3.3
authorization levels 5.6, 11.4.3
autologon 5.2.6
automatic connect/disconnect 8.3, 8.5

B

basic OS modules 3.2

C

CANCEL event 6.3.9
CANCEL task 6.3.10
character generation codes 9.2.3, Appendix
click sound acknowledgement 10.6
cluster configurations 2.1.1
common variables 1.5.2
communication channels 7.1.2
communication concept 7.1
communication control messages 7.4.3
communication handler 3.5.1
communication processor 2.1.3
communication software 7.1.1
conditional print queue request 12.4
connect/disconnect 8.5
control blocks 6.2
control info 5.2.1
console mode 3.11
CP, communication processor 2.1.3
CRCC 7.3.7
CRU, cathode ray tube unit 2.1.2
cursor handling 9.4

D

data diskette 4.4.2
data-FD 11.2.2
data message 7.4.4
data organisation 4.3
data set 4.3
data set types 4.8
DELETE file 11.7.7
DIA, display interface adapter 9.3
diskette capacities 4.2
diskette format 4
display area 9.2
display handler 3.6.1
display unit 2.1.2, 9.1
display unit functions 9
DPU, display processor unit 2.7, 9,6,5
DSA, destination session address 7.3.4

E

ECB, event control block 6.2.2
efficiency considerations 1.5
empty diskette 4.4.1
emulation abort 8.10
emulation status 8.9
emulation status updating 8.9.2
error handling 14
error numbers 14.5
ERRORTYPE 14.4, 14.5
event 1.4.2
event control 6.2
event declaration 6.2.2
exchange of character generation table 11.10
exchange of keyboard table 11.11
EXIT 6.3.11

F

FAC, field attribute characters 9.2.4
FCB, file control block 11.5.2
FD configurations 11.2
FD functions 11
FDE, data set directory entry 4.6
FDIOS 3.8.1, 11.3.3
field attribute characters 9.2.4
file control block 11.5.2
file handling 11.3, 11.5
file oriented commands 11.7
file types 11.4.2
FRDEV 7.3.3

G

general message format
get timer 13.2

H

hardware dependent functions 9.6
hardware environment 2
hold printout 12.6.3
host system statys 8.11

I

I/O synchronization 11.8
inactive state 1.4.3
initialization 5.1
input function modules 3.7
input functions 10
internal communication protocol 7
interrupts and priorities 1.4.4
interrupt handling 6.4
interrupt levels 6.4.4
interrupt procedures 6.4.5
interrupt types 6.4
interrupt vectors 6.4.6
IPL, initial program load 3.3.1, 5.1.1
IRQ, interrupt request 6.4.3

J

K

keyboard 2.3, 10.1
keyboard click sound 10.6
keyboard data structures 10.2
keyboard functions 10
keyboard input 10.4
keyboard input synchronization 10.1.2
keyboard lamps 10.8
keyboard repetition frequency 10.5
keyboard strap data 10.2.3 - 10.2.5
keyboard table 10.2.1
keyboard table header 10.2.2
keyboard type indicator 10.1.2

L

leading flag 7.3.1
library organisation 4.3.2
LOAD file 11.7.8
load map 5.2.1, 5.3.2
load modules 3.3
load character generation table 11.10
load overlay segment 11.9
LOCK 6.4.5
logical address 7.2.2
logoff 5.4
logoff byte 12.9.4
logon 5.2
logon file 5.2.2
logon modules 3.4
LTOC, library table of contents 4

M

manual logon 5.2.1
map loader 3.3.3
maskable interrupts 6.4.4
maxram 9.6.2
MDE, member directory entry 4.3.2
member 4.3.2
memory maps 3.13
memory options 2.2
memory requirements 3.12
menufile 5.2.3
message content 7.3.6
message format 7.3
message line 9.5
message line control block 9.5.2
message line handler 3.6.2
message line logon 5.2.5
messages 7.4
MID, magnetic identification device 2.3.1, 10.9
MID input 10.9
MSA 7.4.2
MSGTYP 7.3.5, 7.4.3, 7.4.4
MRW, read/write memory, 2.1.2, 2.2.1, 9.6.2
multitasking functions 6.3
mutual exclusion 1.4.5

N

NIP, nucleus initialization procedure 3.3.2, 5.1.2

O

OPEN/CLOSE file 11.7.4
open/close for input to DU 10.3
OS interrupt handler 3.2.2
OS request handler 3.2.1
OS request interrupt 6.4.2
overlay 1.5.3
overlay loader procedure 11.9

P

page area 3.12.1
password 5.2.1
parallell processing 1.4
parameter passing 1.5.2
PCU, peripheral control unit 2.5
personal computer 11.2.3
physical address 7.2.1
poll message 7.4.1
poll responses 7.4.2
print edit parameter list 12.9.2
print queue request 12.4
printer editing 12.9
printer handler 3.9.1, 12.1
printer status inquiry 12.3, 12.8
printer hardware 12.10
printer types 12.10
printout request 12.6
priority (task) 1.4.4, 6.3.1
priority (message) 7.4.2
PRIOS 3.9.2, 12.2
PRIOS event 12.2.3
PRIOS optional functions 12.7
PRIOS parameter list 12.2.2
PRIOS TCB 12.2.4
process 1.4
program load 5.3, 11.7.8
PROM identification 9.6.1

Q

R

read DU 8.8
READ record 11.7.5
ready state 1.4.3
references 1.1
release timer 13.3
reset 5.1.1
reset interrupt 6.4.1

S

SCB, session control block 7.1.4
SCC, synchronous communication controller 2.6
selector pen 2.3.2
selector pen input 10.10
send command without data 8.6
session 7.1.3
session control block 7.1.4
short write 8.7.4
SMSA 7.4.2
software interrupt 6.4.2
SSA 7.4.2
STATUS built-in function 14.2
STATUS byte in message 7.3.5, 7.4.1, 7.4.2
status byte in control block, see ECB, TCB, VCB, FCB
status list in OS 7.1.5
strap data 10.2.3 - 10.2.5
SWI, software interrupt 6.4.2
SYSBOT 5.1.1, 5.5.1
SYSIPL 5.5.3
SYSLIB 5.5.4
system data sets 5.5
system diskette 4.4.3
system-FD 11.2.1

T

task 1.4.1, 6.3
task communication 1.4.2
task control block 6.3.2
task declaration 6.3.1
task manager 3.2.3, 6.1
task states 1.4.3
task structures 1.5.1
task termination 6.3.8, 6.3.10, 6.3.11
terminal console functions 3.11
timer functions 13
timer handler 3.2.4
TODEV 7.3.2
TP-status list 7.3.8
trailing flag 7.3.8
transmission commands 8.6, 8.7, 8.8

U

user interface 8
user interface control block (UICB) 8.3
user interface module (UIM) 3.5.2, 8.2
utility supervisor 3.10

V

VCB, volume control block 11.5.1
VOLLAB, volume label 4.5
volume oriented commands 11.6
volume table 11.5.3
volume types 4.4
VTOC, volume table of contents 4.6

W

WAIT event 1.4.2, 6.3.4
WAIT task 1.4.2, 6.3.7
wait state 1.4.3
write DU 8.7
write DU with emulation control 8.7.4
WRITE/REWRITE record

X

Y

Z