Multiprogramming System

RC 4000®

DATAMATICS

RCSL No: 55-D140

RC 4000 SOFTWARE
MULTIPROGRAMMING SYSTEM

Edited by
Per Brinch Hansen

2. edition

A/S REGNECENTRALEN
Copenhagen – February 1971

FOREWORD

The RC 4000 multiprogramming system consists of a monitor program that can be extended with a hierarchy of operating systems to suit diverse requirements of program scheduling and resource allocation.
This manual defines the functions of the monitor and the basic operating system, which allows users to initiate and control parallel program execution from typewriter consoles. The manual is divided into four parts:

PART I.	GENERAL DESCRIPTION
PART II.	MONITOR FUNCTIONS
PART III.	CATALOG INITIALIZATION
PART IV.	BASIC OPERATING SYSTEM

The *general description* explains the philosophy and structure of the system. This part will be of interest to anyone wishing an understanding of the system in order to evaluate its possibilities and limitations without going into details about exact conventions. The discussion treats the hardware structure of the RC 4000 only in passing.
The *initialization of the catalog* of files on the backing store and the *basic operating system* should be studied by those who are to operate the system from consoles.
The definition of *monitor functions* is of interest mainly to programmers. Since it contains exact conventions for the use of monitor procedures in assembly language, the reader must be familiar with the manuals of the RC 4000 computer and the slang assembler.
A maintenance manual of the system will be published separately.
The design of the system is based on the ideas of *Jørn Jensen, Søren Lauesen,* and the author; *Leif Svalgaard* participated in its implementation.

CONTENTS

PART I. GENERAL DESCRIPTION

PART II. MONITOR FUNCTIONS

PART III. CATALOG INITIALIZATION

CONTENTS

PART IV. BASIC OPERATING SYSTEM

PART I.
GENERAL DESCRIPTION

Chapter 1

SYSTEM OBJECTIVES

This chapter outlines the philosophy that guided the design of the RC 4000 multiprogramming system. It emphasizes the need for different operating systems to suit different applications.

The primary goal of *multiprogramming* is to share a central processor and its peripheral equipment among a number of programs loaded in the internal store. This is a meaningful objective if single programs only use a fraction of the system resources and if the speed of the machine is so fast, compared to that of peripherals, that idle time within one program can be utilized by other programs.

The present system is implemented on the RC 4000 computer, a 24-bit, binary computer with typical instruction execution times of 4 microseconds. It permits practically unlimited expansion of the internal store and standardized connection of all kinds of peripherals. Multiprogramming is facilitated by concurrency of program execution and input/output, program interruption, and storage protection.

The aim has been to make multiprogramming feasible on a machine with a minimum internal store of 16 k words backed by a fast drum or disk. Programs can be written in any of the available programming languages and contain programming errors. The storage protection system guarantees non-interference among 8 parallel programs, but it is possible to start up to 23 programs provided some of them are error free.

The system uses standard multiprogramming techniques: the central processor is shared between loaded programs. Automatic swapping of programs in and out of the store is possible but not enforced by the system. Backing storage is organized as a *common data bank*, in which users can retain named files in a semi-permanent manner. The system allows a *conversational mode* of access from typewriter consoles.

An essential part of any multiprogramming system is an *operating system*, a program that coordinates all computational activities and input/output. An operating system must be in complete control of the strategy of program execution, and assist the users with such functions as operator communication, interpretation of job control statements, allocation of resources, and application of execution time limits.

For the designer of advanced information systems, a vital requirement of any operating system is that it allows him to change the mode of operation it controls; otherwise his freedom of design can be seriously limited. Unfortunately this is precisely what present operating systems do not allow. Most of them are based exclusively on a single mode of operation, such as batch processing, priority scheduling, real-time scheduling, or time-sharing.

When the need arises, the user often finds it hopeless to modify an operating system that has made rigid assumptions in its basic design about a specific mode of operation. The alternative – to replace the original operating system with a new one – is in most

computers a serious, if not impossible, matter, the reason being that the rest of the software is intimately bound to the conventions required by the original system.

This unfortunate situation indicates that the main problem in the design of a multiprogramming system is not to define functions that satisfy specific operating needs, but rather to supply a system nucleus that can be extended with new operating systems in an orderly manner. This is the primary objective of the RC 4000 system.

The nucleus of the RC 4000 multiprogramming system is a *monitor* program with complete control of storage protection, input/output, and interrupts. Essentially the monitor is a software extension of the hardware structure, which makes the RC 4000 more attractive for multiprogramming. The following elementary functions are implemented in the monitor:

scheduling of time slices among programs executed in parallel by means of a digital clock,

initiation and control of program execution at the request of other running programs,

transfer of messages among running programs,

initiation of data transfers to or from peripherals.

The monitor has no built-in strategy of program execution and resource allocation; it allows any program to initiate other programs in a hierarchal manner and to execute them according to any strategy desired. In this *hierarchy of programs* an operating system is simply a program that controls the execution of other programs. Thus operating systems can be introduced in the system as other programs without modification of the monitor. Furthermore operating systems can be replaced dynamically, enabling each installation to switch among various modes of operation; several operating systems can, in fact, be active simultaneously.

In the following chapters we shall explain this dynamic operating system concept in detail. In accordance with our philosophy all questions about particular strategies of program scheduling will be postponed, and the discussion will concentrate on the fundamental aspects of the control of an environment of parallel processes.

Chapter 2

ELEMENTARY MULTIPROGRAMMING PROBLEMS

This chapter introduces the elementary multiprogramming problems of mutual exclusion and synchronization of parallel processes. The discussion is restricted to the logical problems that arise when independent processes try to access common variables and shared resources. An understanding of these concepts is indispensable to the uninitiated reader, who wants to appreciate the difficulties of switching from uniprogramming to multiprogramming.

2.1. Multiprogramming

In multiprogramming the sharing of computing time among programs is controlled by a *clock*, which interrupts program execution frequently and activates a *monitor program*. The monitor saves the registers of the interrupted program and allocates the next slice of computing time to another program and so on. Switching from one program to another is also performed whenever a program must wait for the completion of input/output.

Thus although the computer is only able to execute one instruction at a time, multiprogramming creates the illusion that programs are being executed simultaneously, mainly because peripherals assigned to different programs indeed operate in parallel.

2.2. Parallel Processes

Most of the elementary problems in multiprogramming arise from the fact that one *process* (e.g. an executed program) cannot make any assumptions about the relative speed and progress of other processes. This is a potential source of conflict whenever two processes try to access a common variable or a shared resource.

It is evident that this problem will exist in a truly parallel system, in which programs are executed simultaneously on several central processors. It should be realized, however, that the problem will also appear in a quasi-parallel system based on the sharing of a single processor by means of interrupts; since a program cannot detect when it has been interrupted, it does not know how far other programs have progressed.

Another way of stating this is that if one considers the system as seen from within a program, it is irrelevant whether multiprogramming is implemented on one or more central processors – the logical problems are the same.

Consequently a multiprogramming system must in general be viewed as an environment with a number of truly *parallel processes*. Having reached this conclusion, a natural generalization is to treat not only program execution but input/output also as independent, parallel processes. This point will be illustrated abundantly in the following chapters.

2.3. Mutual Exclusion

The idea of multiprogramming is to share the computing equipment among a number of parallel programs. At any moment, however, a given resource must belong to one program only. In order to ensure this it is necessary to introduce global variables, which programs can inspect to decide whether a given resource is available or not.

As an example consider a typewriter used by all programs for messages to the operator. To control access to this device we might introduce a global boolean *typewriter available.* When a program p wishes to output a message, it must examine and set this boolean by means of the following instructions:

```
wait:   load      typewriter available
        skip if   true
        jump to   wait
        load      false
        store     typewriter available
```

While this is taking place the program may be interrupted after the loading of the boolean, but before inspection and assignment to it. The register containing the value of the boolean is then stored within the monitor, and program q is started. Q may load the same boolean and find that the typewriter is available. Q accordingly assigns the value false to the boolean and starts using the typewriter. After a while q is interrupted, and at some later time p is restarted with the original contents of the register reestablished by the monitor. Program p continues the inspection of the original value of the boolean and concludes erroneously that the typewriter is available.

This conflict arises because programs have no control over the interrupt system. Thus the only indivisible operations available to programs are single instructions such as load, compare, and store. This example shows that one cannot implement a multiprogramming system without ensuring a *mutual exclusion* of programs during the inspection of global variables. Evidently the entire reservation sequence must be executed as an *indivisible function.* One of the purposes of a monitor program is to execute indivisible functions in the disabled mode.

In the use of reservation primitives one must be aware of the problem of "the *deadly embrace*" between two processes, p and q, which attempt to share the resources r and s as follows:

process p: wait and reserve(r) - - - wait and reserve(s) -
process q: wait and reserve(s) - - - wait and reserve(r) -

This can cause both processes to wait forever, since neither is aware of that it wants what the other one has.

To avoid this problem we need a third process (an *operating system*) that controls the allocation of shared resources between p and q in a manner that guarantees that both will be able to proceed to completion (if necessary by delaying the other until resources become available).

2.4. Mutual Synchronization

In a multiprogramming system parallel processes must be able to *cooperate* in the sense that they can activate one another and exchange information. One example of a process activating another process is the initiation of input/output by a program. Another example is that of an operating system that schedules a number of programs.

The exchange of information between two processes can also be regarded as a problem of mutual exclusion, in which the receiver must be prevented from inspecting the information until the sender has delivered it in a common storage area.

Since the two processes are independent with respect to speed, it is not certain that the receiver is ready to accept the information at the very moment the sender wishes to deliver it, or conversely the receiver can become idle at a time when there is no further information for it to process.

This problem of the *synchronization* of two processes during a transfer of information must be solved by indivisible monitor functions, which allow a process to be *delayed* on its own request and *activated* on request from another process.

For a more extensive analysis of multiprogramming fundamentals, the reader should consult E. W. Dijkstra's monograph: *Cooperating Sequential Processes.* Math. Dep. Technological University, Eindhoven, (Sep. 1965).

Chapter 3

BASIC MONITOR CONCEPTS

This chapter opens a detailled description of the RC 4000 monitor. A multiprogramming system is viewed as an environment in which program execution and input/output are handled uniformly as cooperating, parallel processes. The need for an exact definition of the process concept is stressed. The purpose of the monitor is to bridge the gap between the actual hardware and the abstract concept of multiprogramming.

3.1. Introduction

The aim has been to implement a multiprogramming system that can be extended with new operating systems in a well-defined manner. In order to do this a sharp distinction must be made between the *control* and the *strategy* of program execution.

The mechanisms provided by the monitor solve the logical problems of the control of parallel processes. They also solve the safety problems that arise when erroneous or malicious processes try to interfere with other processes. They do, however, leave the choice of particular strategies of program scheduling to the processes themselves.

With this objective in mind we have implemented the following fundamental mechanisms within the monitor:

- simulation of parallel processes,
- communication among processes,
- creation, control, and removal of processes.

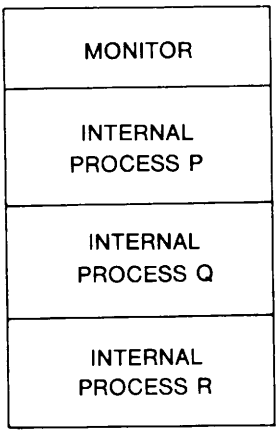
3.2. Programs and Internal Processes

As a first step we shall assign a precise meaning to the process concept, i.e. introduce an unambiguous terminology for what a process is and how it is implemented on the RC 4000.

We distinguish between internal and external processes, roughly corresponding to program execution and input/output.

More precisely: an *internal process* is the execution of one or more interruptable programs in a given storage area. An internal process is identified by a unique *process name*. Thus other processes need not be aware of the actual location of an internal process in the store, but can refer to it by name.

The following figure illustrates a division of the internal store among the monitor and three internal processes, p, q, and r:



Later it will be explained how internal processes are created and how programs are loaded into them. At this point it should only be noted that an internal process occupies a fixed, contiguous storage area during its whole lifetime. The monitor has a *process description* of each internal process; this table defines the name, storage area, and current state of the process.

Computing time is shared cyclically among all active internal processes; as a standard the monitor allocates a maximum time slice of 25 milliseconds to each internal process in turn; after the elapse of this interval the process is interrupted and its registers are stored in the process description; following this the monitor allocates 25 milliseconds to the next internal process, and so on. The cyclic queue of active internal processes is called the *time slice queue.*

A sharp distinction is made between the concepts program and internal process. A *program* is a collection of instructions describing a computational process, whereas an internal process is the execution of these instructions in a given storage area.

An internal process like p can involve the execution of a sequence of programs, for example, editing followed by translation and execution of an object program. It is also possible that copies of the same program (e.g. the Algol compiler) can be executed simultaneously in two processes q and r. These examples illustrate the need for a distinction between programs and processes.

3.3. Documents and External Processes

In connection with input/output the monitor distinguishes between peripheral devices, documents, and external processes.

A *peripheral device* is an item of hardware connected to the data channel and identified by a device number.

A *document* is a collection of data stored on a physical medium. Examples of documents are:

a roll of paper tape,
a deck of punched cards,
a printer form,
a reel of magnetic tape,
a data area on the backing store.

By the expression *external process* we refer to the input/output of a given document identified by a unique process name. This concept implies that once a document has been mounted, internal processes can refer to it by name without knowing the actual device it uses.

For each external process the monitor keeps a process description defining its name, kind, device number, and current state. The *process kind* is an integer defining the kind of peripheral device on which the document is mounted.

For each kind of external process the monitor contains an interrupt procedure that can initiate and terminate input/output on request from internal processes.

3.4. Monitor

The monitor is a program activated by means of interrupts. It can execute privileged instructions in the disabled mode, meaning that (1) it is in complete control of input/output, storage protection, and the interrupt system, and that (2) it can execute a sequence of instructions as an indivisible entity.

After initial system loading the monitor resides permanently in the internal store.

We do not regard the monitor as an independent process, but rather as a software extension of the hardware structure, which makes the computer more attractive for multiprogramming. Its function is to (1) keep descriptions of all processes; (2) share computing time among internal and external processes; and (3) implement procedures that processes can call in order to create and control other processes and communicate with them.

So far we have described the multiprogramming system as a set of independent, parallel processes identified by names. The emphasis has been on a clear understanding of relationships among resources (store and peripherals), data (programs and documents), and processes (internal and external).

Chapter 4

PROCESS COMMUNICATION

This chapter deals with the monitor procedures for the exchange of information between two parallel processes. The mechanism of message buffering is defended on the grounds of safety and efficiency.

4.1. Message Buffers and Queues

Two parallel processes can cooperate by sending messages to each other.

A *message* consists of eight words. Messages are transmitted from one process to another by means of *message buffers* selected from a common *pool* within the monitor.

The monitor administers a *message queue* for each process. Messages are linked to this queue when they arrive from other processes. The message queue is a part of the process description.

Normally a process serves its queue on a first-come, first-served basis. After the processing of a message, the receiving process returns an *answer* of eight words to the sending process in the same buffer.

As described in Section 2.4, communication between two independent processes requires a synchronization of the processes during a transfer of information. A process requests synchronization by executing a wait operation; this causes a delay of the process until another process executes a send operation.

The term *delay* means that the internal process is removed temporarily from the time slice queue; the process is said to be *activated* when it is again linked to the time slice queue.

4.2. Send and Wait Procedures

The following monitor procedures are available for communication among internal processes:

send message (receiver, message, buffer)
wait message (sender, message, buffer)
send answer (result, answer, buffer)
wait answer (result, answer, buffer)

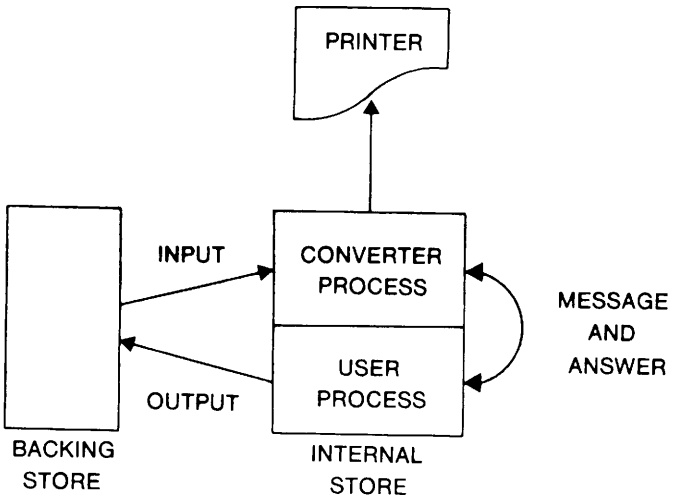
Send message copies a message into the first available buffer within the pool and delivers it in the queue of a named receiver. The receiver is activated if it is waiting for a message. The sender continues after being informed of the address of the message buffer.

Wait message delays the calling process until a message arrives in its queue. When the process is allowed to proceed, it is supplied with the name of the sender, the contents of the message, and the address of the message buffer. The buffer is removed from the queue and is now ready to transmit an answer.

Send answer copies an answer into a buffer in which a message has been received and delivers it in the queue of the original sender. The sender of the message is activated if it is waiting for the answer. The answering process continues immediately.

Wait answer delays the calling process until an answer arrives in a given buffer. On arrival, the answer is copied into the process and the buffer is returned to the pool. The result specifies whether the answer is a response from another process, or a dummy answer generated by the monitor in response to a message addressed to a non-existing process.

The use of these procedures can be illustrated by the following example of a conversational process. The figure below shows one of several user processes, which deliver their output on the backing store. After completion of its output a user process sends a message to a converter process requesting it to print the output. The converter process receives and serves these requests one by one, thus ensuring that the line printer is shared by all user processes with a minimum delay.



The algorithms of the converter and the user are as follows:

```
converter process:
        wait message (sender, message, buffer);
        print from backing store (message);
        send answer (result, answer, buffer);
        goto converter process;
```

```
user process:
        ---
        output on backing store;
        send message(<:converter:>, message, buffer);
        wait answer( result, answer, buffer);
        ---
```

4.3. General Event Procedures

The communication procedures enable a conversational process to receive messages simultaneously from several other processes. To avoid becoming a bottleneck in the system, however, a conversational process must be prepared to be actively engaged in more than one conversation at a time. As an example think of a conversational process that engages itself, on request from another process, in a conversation with one of several human operators in order to perform some manual operation (mounting of a tape etc.). If one restricts a conversational process to only accepting one request (i.e. a message) at a time, and to completing the requested action before receiving the next request, the unacceptable consequence of this is that other processes (including human operators at consoles) can have their requests for response delayed for a long or even undefined time.

As soon as a conversational process has started a lengthy action, by sending a message to some other process, it must receive further messages and initiate other actions. It will then be reminded later of the completion of earlier actions by means of normal answers. In general a conversational process is now engaged in several requests at one time. This introduces a scheduling and resource problem: when the process receives a request, some of its resources (storage or peripheral devices) can be tied up by already initiated actions; thus in some cases the process will not be able to honor new requests before old ones are completed. In this case the process wants to postpone the reception of some requests and leave them pending in the queue, while examining others.

The procedures *wait message* and *wait answer*, which force a process to serve its queue in a strict sequential order and delay itself while its own requests to other processes are completed, do not fulfill the above requirements.

Consequently we have introduced two more general communication procedures, which enable a process to wait for the arrival of the next message or answer and serve its queue in any order:

```
wait event (last buffer, next buffer, result)
get event (buffer)
```

The term *event* denotes a message or an answer. In accordance with this the queue of a process from now on will be called the *event queue.*

Wait event delays the calling process until either a message or an answer arrives in its queue after a given last buffer.The process is supplied with the address of the next buffer and a result indicating whether it contains a message or an answer. If the last

buffer address is zero, the queue is examined from the start. The procedure does not remove the next buffer from the queue or in any other way change its status.
As an example, consider an event queue with two pending buffers A and B:

queue = buffer A, buffer B

The monitor calls: wait event(0,buffer) and wait event(A, buffer) will cause immediate return to the process with *buffer* equal to A and B, respectively; while the call: wait event(B, buffer) will delay the process until another message or answer arrives in the queue after buffer B.
Get event removes a given buffer from the queue of the calling process. If the buffer contains a message, it is made ready for the sending of an answer. If the buffer contains an answer, it is returned to the common pool. The copying of the message or answer from the buffer must be done by the process itself before *get event* is called (see Appendix A.3. Administration of Message Buffers).
The following algorithm illustrates the use of these procedures within a conversational process:

```
first event:     buffer:=0;
next event:      last buffer:=buffer;
                 wait event(last buffer, buffer, result);
                 if result = message then
                 begin
exam request:    if resources not available then go to next event;
init action:     get event (buffer);
                 reserve resources;
                 ----
                 send message to some other process;
                 save state of action;
                 end else
                 begin comment: result = answer;
term action:     restore state of action;
                 get event (buffer);
                 release resources;
                 send answer to original sender;
                 end;
                 go to first event;
```

The process starts by examining its queue; if empty, it awaits the arrival of the next event. If it finds a message, it checks whether it has the necessary resources to perform the requested action; if not, it leaves the message in the queue and examines the next event. Otherwise it accepts the message, reserves resources, and initiates an action. As soon as this involves the sending of a message to some other process, the conver-

sational process saves information about the state of the incomplete action and proceeds to examine its queue from the start in order to engage itself in another action.

Whenever the process finds an answer in its queue, it immediately accepts it and completes the corresponding action. It can now release the resources used and send an answer to the original sender that made the request. After this it examines the entire queue again to see whether the release of resources has made it possible to accept pending messages.

One example of a process operating in accordance with this scheme is the basic operating system s, which creates internal processes on request from typewriter consoles. S can be engaged in conversations with several consoles at the same time. It will only postpone an operator request if its storage is occupied by other requests, or if it is already in the middle of an action requested from the same console (see Section 16.6).

4.4. Advantages of Message Buffering

In the design of the communication scheme we have given full recognition to the fact that the multiprogramming system is a dynamic environment, in which some of the processes may turn out to be black sheep.

The system is dynamic in the sense that processes can appear and disappear at any time. Therefore a process does not in general have a complete knowledge about the existence of other processes. This is reflected in the procedure *wait message*, which makes it possible for a process to be unaware of the existence of other processes until it receives messages from them.

On the other hand once a communication has been established between two processes (e.g. by means of a message), they need a common identification of it in order to agree on when it is terminated (e.g. by means of an answer). Thus we can properly regard the selection of a buffer as the creation of an identification of a conversation. A happy consequence of this is that it enables two processes to exchange more than one message at a time.

We must be prepared for the occurance of erroneous or malicious processes in the system (e.g. undebugged programs). This is tolerable only if the monitor ensures that no process can interfere with a conversation between two other processes. This is done by storing information about the sender and receiver in each buffer, and checking it whenever a process attempts to send or wait for an answer in a given buffer.

Efficiency is obtained by the queuing of buffers, which enables a sending process to continue immediately after delivery of a message or an answer regardless of whether the receiver is ready to process it or not.

In order to make the system dynamic it is vital that a process can be removed at any time, even if it is engaged in one or more conversations. In the previous example of user processes that deliver their output on the backing store and ask a converter process to print it, it would be sensible to remove a user process that has completed its task and is now only waiting for an answer from the converter process. In this case the

monitor leaves all messages from the removed process undisturbed in the queues of other processes. When these processes terminate their actions by sending answers, the monitor simply returns the buffers to the common pool.

The reverse situation is also possible: during the removal of a process, the monitor finds unanswered messages sent to the process. These are returned as dummy answers to the senders. A special instance of this is the generation of a dummy answer to a message addressed to a process that does not exist.

The main drawback of message buffering is that it introduces yet another resource problem, since the common pool contains a finite number of buffers. If a process was allowed to empty the pool by sending messages to ignorant processes, which do not respond with answers, further communication within the system would be blocked. We have consequently set a limit to the number of messages a process can send simultaneously. By doing this, and by allowing a process to transmit an answer in a received buffer, we have placed the entire risk of a conversation on the process that opens it (see Section 7.4).

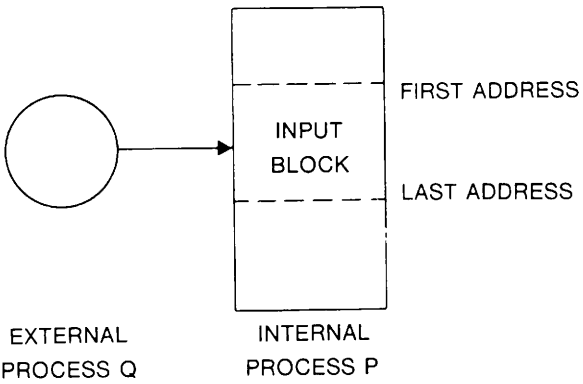
Chapter 5

EXTERNAL PROCESSES

This chapter clarifies the meaning of the external process concept. It explains initiation of input/output by means of messages from internal processes, dynamic creation and removal of external processes, and exclusive access to documents by means of reservation. The similarity of internal and external processes is stressed.

5.1. Initiation of Input/Output

Consider the following situation, in which an internal process, p, inputs a block from an external process, q (say, a magnetic tape):



P initiates input by sending a message to q:

send message (<:q:>, message, buffer)

The message consists of eight words defining an input/output operation and the first and last addresses of a storage area within process p:

message: operation
first storage address
last storage address
(five irrelevant words)

The monitor copies the message into a buffer and delivers it in the queue of process q. Following this it uses the kind parameter in the process description of process q to switch to a piece of code common to all magnetic tapes. If the tape station is busy, the message is merely left in its queue; otherwise input is initiated to the given storage area. On return, program execution continues in process p.

When the tape station completes input by means of an interrupt, the monitor generates an answer and delivers it in the queue of p, which in turn receives it by calling *wait answer:*

wait answer (result, answer, buffer)

The answer contains status bits sensed from the device and the actual block length expressed as the number of bytes and characters input:

```
answer:   status bits
          number of bytes
          number of characters
          (five irrelevant words)
```

After delivery of the answer, the monitor examines the queue of the external process q and initiates its next operation (unless the queue is empty).

Essentially all external processes follow this scheme, which can be defined by the following algorithm:

```
external process:   wait message;
                    analyse and check message;
                    initiate input/output;
                    wait interrupt;
                    generate answer;
                    send answer;
                    goto external process;
```

With low-speed, character-oriented devices, the monitor repeats input/output and the interrupt response for each character until a complete block has been transferred; (while this is taking place, the time between interrupts is of course shared among internal processes). Internal processes can therefore regard all input/output as block oriented.

5.2. Reservation and Release

The use of message buffering provides a direct way of sharing an external process among a number of internal processes: an external process can simply accept messages from any internal process and serve them in their order of arrival. An example of this is the use of a single typewriter for output of messages to a main operator.

This method of sharing a device ensures that a block of data is input or output as an indivisible entity. When sequential media such as paper tape, punched cards, or magnetic tape are used, however, an internal process must have exclusive access to the entire document. This is obtained by calling the following monitor procedure:

reserve process (name, result)

The result indicates whether the reservation has been accepted or not.

An external process that handles sequential documents of this kind rejects messages from all internal processes except the one that has reserved it. Rejection is indicated by the result of the procedure *wait answer.*

During the removal of an internal process, the monitor removes all reservations made by it. Internal processes can, however, also do this explicitly by means of the monitor procedure:

release process (name)

5.3. Creation and Removal

From the operator's point of view an external process is created when he mounts a document on a device and names it. The name must, however, be communicated to the monitor by means of an operating system, i.e. an internal process that controls the execution of programs. Thus it is more correct to say that external processes are created when internal processes assign names to peripheral devices. This is done by means of the monitor procedure:

create peripheral process (name, device number, result)

The monitor has, in fact, no way of ensuring whether a given document is mounted on a device. Furthermore, there are some devices which operate without documents, e.g. the real-time clock.

The name of an external process can be explicitly removed by a call of the monitor procedure:

remove process (name, result)

It is also possible to implement an automatic removal of the process name when the monitor detects operator intervention in a device. At present, this is done only in connection with magnetic tapes (see Section 10.1).

5.4. Replacement of External Processes

The decision to control input/output by means of interrupt procedures within the monitor, instead of using dedicated internal processes for each kind of peripheral device, was made to obtain immediate initiation of input/output after the sending of messages. In contrast the activation of an internal process merely implies that it is linked to the time slice queue; after activation several time slices can elapse before the internal process actually starts to execute instructions.

The price paid for the present implementation of external processes is a prolongation of the time spent in the disabled mode within the monitor. This limits the system's ability to cope with real-time events, i.e. data that are lost unless they are input and processed within a certain time.

An important consequence of the uniform handling of internal and external processes is that it allows us to replace any external process by an internal process of the same name; other processes that communicate with it are quite unaware of this replacement.

Thus it is possible to improve the response time of the system by replacing a time-consuming external process, such as the paper tape reader, by a somewhat slower internal process, which executes privileged instructions in the enabled mode.

This type of replacement also makes it possible to enforce more complex rules of access to a document. In the interests of security, for example, one might want to limit the access of an internal process to one of several files recorded on a particular magnetic tape. This can be ensured by an internal process that traps all messages to the tape and decides whether they should be passed on to it.

As a final example let us consider the problem of debugging a process control system before it is connected to an industrial plant. A convenient way of doing this is to replace analog inputs with an internal process that simulates relevant values of actual measuring instruments.

We conclude that the ability to replace any process in the system with another process is a very useful tool. This can now be seen as a practical result of the general, but somewhat vague idea (expressed in Section 2.2) that internal and external processes are independent processes, which differ only in their processing capability.

Chapter 6

INTERNAL PROCESSES

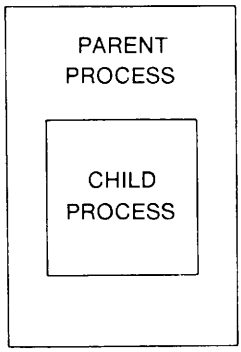
This chapter explains the creation and control of internal processes. The emphasis is on the hierarchal structuring of internal processes, which makes it possible to extend the system with new operating systems. The dynamic behaviour of the system is explained in terms of process states and the transition between these.

6.1. Creation, Control, and Removal

Internal processes are *created* on request from other internal processes by means of the monitor procedure:

create internal process (name, parameters, result)

The monitor initializes the process description of the new internal process with its name and storage area selected by the *parent process*. The storage area must be within the parent's own area. Also specified by the parent is a protection key, which must be set in all storage words of the *child process* before it is started.



After creation the child process is simply a named storage area, which is described within the monitor. It has not yet been linked to the time slice queue.

The parent process can now *load* a program into the child process by means of an input operation. Following this the parent can *initialize* the *registers* of its child using the monitor procedure:

modify internal process (name, registers, result)

The register values are stored in the process description until the child process is started. As a standard convention adopted by parent processes (but not enforced by

the monitor), the registers inform the child about the process descriptions of itself, its parent, and the typewriter console it can use for operator communication.
Finally the parent can *start* program execution within the child by calling:

start internal process (name, result)

which sets the protection keys within the child and links it to the time slice queue. The child now shares time slices with other active processes including the parent.
On request from a parent process, the monitor waits for the completion of all input/output initiated by a child process and *stops* it, i.e. removes it from the time slice queue:

stop internal process (name, buffer, result)

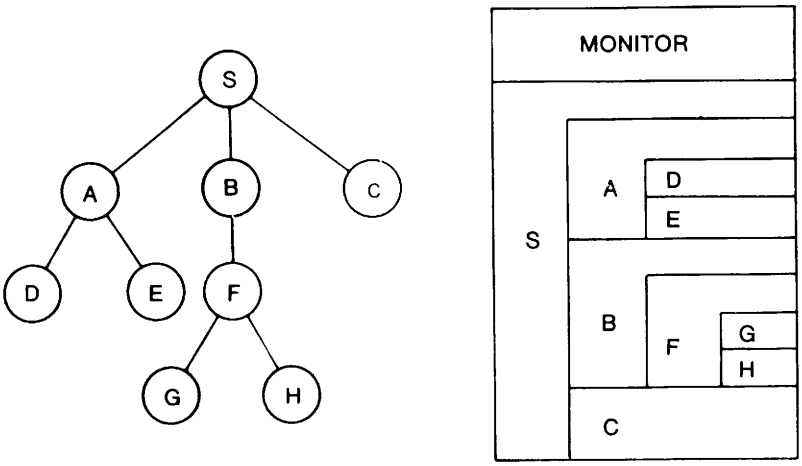
The meaning of the message buffer will be made clear in Section 6.3.
In the stopped state a child process can be modified and started again, or it can be completely *removed* by the parent process:

remove process (name, result)

During removal, the monitor generates dummy answers to all messages sent to the child and releases all external processes used by it. Finally the protection keys are reset to the value used within the parent process. The parent can now use the storage area to create other child processes.

6.2. Process Hierarchy

The idea of the *monitor* has been described as the simulation of an environment in which program execution and input/output are handled uniformly as parallel, cooperating processes. A fundamental set of procedures allows the dynamic creation and control of processes as well as communication among them.
For a given installation we still need, as part of the system, programs that control strategies for operator communication, program scheduling, and resource allocation. But it is essential for the orderly growth of the systems that these *operating systems* be implemented as other programs. Since the difference between operating systems and production programs is one of jurisdiction only, this problem is solved by arranging the internal processes in a *hierarchy* in which parent processes have complete control over child processes.
After initial loading the internal store contains the monitor and an internal process, s, which is the *basic operating system.* S can create parallel processes, a, b, c, etc., on request from consoles. These processes can in turn create other processes, d, e, f, etc. Thus while s acts as a primitive operating system for a, b, and c, these in turn act as operating systems for their children, d, e, f, etc. This is illustrated by the following figure, which shows a *family tree* of processes on the left and the corresponding storage allocation on the right:



This family tree of processes can be extended to any level, subject only to a limitation of the total number or processes. At present the maximum number of internal processes is 23 including the basic operating system s. It must, however, be remembered that the storage protection system only provides mutual protection of 8 independent processes. When this number is exceeded, one must rely on some of the processes being error free.

In this multiprogramming system all privileged functions are implemented in the monitor, which has no built-in strategy. Strategies can be introduced at the various higher levels, where each process has the power to control the scheduling and resource allocation of its own children. The only rules enforced by the monitor are the following: a process can only allocate a subset of its own resources (including storage) to its children; a process can only modify, start, stop, and remove its own children.

The structure of the family tree is defined in the process descriptions within the monitor. We emphasize that the only function of the tree is to define the basic rules of process control and resource allocation. Time slices are shared evenly among active processes regardless of their position in the hierarchy, and each process can communicate with all other processes.

As regards the future development of operating systems, the most important characteristics can now be seen as the following:

1. *New operating systems can be implemented as other programs* without modification of the monitor. In this connection we should mention that the Algol and Fortran languages for the RC 4000 contain facilities for calling the monitor and initiating parallel processes. Thus it is possible to write operating systems in high-level languages.
2. *Operating systems can be replaced dynamically,* thus enabling an installation to switch among various modes of operation; several operating systems can, in fact, be active simultaneously,

3. *Standard programs and user programs can be executed under different operating systems* without modification; this is ensured by a standardization of communication between parents and children.

6.3. Process States

We are now in a position to define the possible states of an internal process as described within the monitor. An understanding of the transition from one state to the other is vital as a key to the dynamic behaviour of the system.

An internal process is either *running* (executing instructions or ready to do so) or *waiting* (for an event outside the process). In the running state the process is linked to the time slice queue; in the waiting state it is temporarily removed from this queue.

A process can either be *waiting* for a *message,* an *answer,* or an *event,* as explained in Chapter 4.

Of a more complex nature are the situations in which a process is *waiting* to be *stopped* or *started* by another process. In order to explain this we shall once more refer to the family tree shown in the previous section.

Let us say that process b wants to stop its child f. The purpose of doing this is to ensure that all program execution and input/output within the storage area of process f is stopped. Since a part of the storage area has been allocated to children of f, it is obviously necessary to stop not only the *child* f but also all *descendants* of f. This is complicated by the fact that some of these descendants may already have been stopped by their own parents. In the present example process g may still be running, while process h may have been stopped by its parent f. Consequently the monitor should only stop processes f and g.

Consider now the reverse situation, in which process b starts its child f again. Now the purpose is to reestablish the situation exactly as it was before process f was stopped. Thus the monitor must be very careful only to start those descendants of f that were stopped along with f. In our example the monitor must start processes f and g but not h. Otherwise we confuse f, which still relies on its child h being stopped.

Obviously, then, the monitor must distinguish between processes that are stopped by their *parents* and by their *ancestors.*

The possible *states* of an internal process are the following:

running
running after error
waiting for message
waiting for answer
waiting for event
waiting for start by parent
waiting for stop by parent
waiting for start by ancestor
waiting for stop by ancestor
waiting for process function

A process is created in the state *waiting for start by parent.* When it is started, its state becomes *running.* The meaning of the state *running after error* is explained in Section 8.1.

When a parent wants to stop a child, the state of the child is changed to *waiting for stop by parent,* and all running descendants of the child are described as *waiting for stop by ancestor.* At the same time these processes are removed from the time slice queue.

What remains to be done is to ensure that all input/output initiated by these processes is terminated. In order to control this each internal process description contains an integer called the *stop count.* The stop count is increased by one each time the internal process initiates input/output from an external process. On arrival of an answer from an external process, the monitor decreases the stop count by one and examines the state of the internal process. If the stop count becomes zero and the process is *waiting for stop by parent* (or *ancestor),* its state is changed to *waiting for start by parent* (or *ancestor).*

Only when all involved processes are waiting for start is the stop operation finished. This can last some time, and it may not be acceptable to the parent (being an operating system with many other duties) to be inactive for so long. For this reason the stop operation is split into two parts. The stop procedure:

stop internal process (name, buffer, result)

only initializes the stopping of a child and selects a message buffer for the parent. When the child and its running descendants are completely stopped, the monitor delivers an answer to the parent in this buffer. Thus the parent can use the procedures *wait answer* or *wait event* to wait for the completion of the stop.

A process can be in any state when a stop is initiated. If it is waiting for a message, answer, or an event, its state will be changed to waiting for stop, as explained above, but at the same time its instruction counter is decreased by two in order that it can repeat the call of *wait message, wait answer,* or *wait event* when it is started again.

It should be noted that a process can receive messages and answers in its queue in any state. This ensures that a process does not loose contact with its surroundings while stopped.

The meaning of the state *waiting for process function* is explained in Section 9.1.

Chapter 7

RESOURCE CONTROL

This chapter describes a set of monitor rules that enables a parent process to control the allocation of resources to its children.

7.1. Introduction

In the multiprogramming system the internal processes compete for the following limited resources:

computing time
storage and protection keys
message buffers
process descriptions
peripheral devices
backing storage

Initially all resources are owned by the basic operating system s. As a basic principle enforced by the monitor a process can only allocate a subset of its own resources to a child process. These are returned to the parent process when the child is removed.

7.2. Time Slice Scheduling

All running processes are allocated *time slices* in a cyclical manner. Depending on the interrupt frequency of the hardware interval timer, the length of a time slice can vary between 1.6 and 1638.4 milliseconds. A reasonable time slice is 25.6 milliseconds; with shorter intervals the percentage of computing time consumed by timer interrupts grows drastically; with longer intervals the delay between activation and execution of an internal process increases.

In practice internal processes often initiate input/output and wait for it in the middle of a time slice. This creates a scheduling problem when internal processes are activated by answers: Should the monitor link processes to the beginning or to the end of the time slice queue? The first possibility ensures that processes can use peripherals with maximum speed, but there is the danger that a process can monopolize computing time by communicating frequently with fast devices. The second choice prevents this, but introduces a delay in the time slice queue, which slows down peripherals.

We have introduced a modified form of round-robin scheduling to solve this dilemma. As soon as a process is removed from the time slice queue, the monitor stores the actual value of the *time quantum* used by it. When the process is activated again, the monitor compares this quantum with the maximum time slice. As long as this limit is not exceeded, the process is linked to the beginning of the queue; otherwise it is linked to the end of the queue and its time quantum is reset to zero. The same test is applied when the interval timer interrupts an internal process.

This scheduling attempts to share computing time evenly among active internal processes regardless of their position in the hierarchy. It permits a process to be

activated immediately until it threatens to monopolize the central processor; only then is it pushed into the background to give other processes a chance. This is admittedly a built-in strategy at the microlevel. Parent processes can in fact only control the allocation of computing time to their children in larger portions (on the order of seconds) by means of the procedures *start* and *stop internal process*.

For accounting purposes the monitor retains the following information for each internal process: the time at which the process was created and the sum of time quantums used by it; these quantities are denoted *start time* and *run time*.

7.3. Storage Allocation and Protection

An internal process can only create child processes within its own storage area. The monitor does not check whether storage areas of child processes overlap each other. This freedom can be used to implement time-sharing of a common storage area among several processes as described in Sections 10.2 and 10.4.

During creation of an internal process the parent must specify the values of the *protection register* and the *protection key* used by the child. In the protection register each bit corresponds to one of the eight possible protection keys; if a bit is zero the process can change or execute storage words with the corresponding key.

The protection key is the key that is set in all storage words of the child process itself. A parent process can only allocate a subset of its own protection keys to a child. It has complete freedom to allocate identical or different keys to its children. The keys remain accessible to the parent after creation of a child.

7.4. Message Buffers and Process Descriptions

The monitor only has room for a finite number of message buffers and tables describing internal processes and the so-called area processes (files on the backing store used as external processes). A message buffer is selected when a message is sent to another process; it is released when the sending process receives an answer. A process description is selected when an internal process creates another internal process or an area process, and released when the process is removed.

Thus it is clear that message buffers and process descriptions only assume an identity when they are actually used. As long as they are unused, they can be regarded as anonymous pools of resources. Consequently it is sufficient to specify the maximum number of each resource an internal process can use. These so-called *buffer claim*, *internal claim*, and *area claim* are defined by the parent when a child process is created. The claims must be a subset of the parent's own claims, which are diminished accordingly; they are returned to the parent when the child is removed.

The buffer claim defines the maximum number of messages an internal process can exchange simultaneously with other internal and external processes. The internal claim limits the number of children an internal process can have at the same time. The area claim defines how many backing store areas an internal process can access simultaneously.

The monitor decreases a claim by one each time a process actually uses one of its

resources, and increases it by one when the resource is released again. Thus at any moment the claims define the number of resources that can still be used by the process.

7.5. Peripheral Devices

A distinction has been made between peripheral devices and external processes. An external process is created when a name is assigned to a device.

Thus it is also true of peripheral devices that they only assume an identity when they are actually used for input/output. Indeed the whole idea of identification by names is to give the operator complete freedom in allocation of devices. It would therefore seem natural to control the allocation of devices to internal processes by a complete set of claims - one for each kind of device.

In a system with remote peripherals, however, it is unrealistic to treat all devices of a given kind as a single, anonymous pool. An operating system must be able to force its children and their human operators to remain within a certain geographical *configuration* of devices. It should be noted that the concept of configuration must be defined in terms of physical devices and not in terms of external processes, since a parent generally speaking does not know in advance which documents its children are going to use.

Configuration control is exercised as follows. From the point of view of other processes an internal process is identified by a name. Within the monitor, however, an internal process can also be identified by a single bit in a machine word. The process descriptions of peripheral devices include a word in which each bit indicates whether the corresponding internal process is a *potential user* of the device. Another word indicates the *current user* that has reserved the device in order to obtain exclusive access to a document.

Initially the basic operating system s is a potential user of all peripherals. A parent process can *include* or *exclude* a child as a user of any device, provided the parent is also a user of it:

include user (child, device number, result)
exclude user (child, device number, result)

During removal of a child, the monitor excludes it as a user of all devices.

All in all three conditions must be fulfilled before an internal process can initiate input/output:

The device must be an external process with a unique name.

The internal process must be a user of the device.

The internal process must reserve the external process if it controls a sequential document.

7.6. Privileged Functions

Files on the backing store are described in a catalog, which is also kept on the backing store. Clearly there is a need to be able to prevent an internal process from reserving an excessive amount of space in the catalog or on the backing store as such. It seems difficult, however, to specify a reasonable rule in the form of a claim that is defined once and for all when a child process is created. The main difficulty is that catalog entries and data areas can survive the removal of the process that created them; in other words backing storage is a resource a parent process can loose permanently by allocating it to its children.

As a half-hearted solution we have introduced the concept of *privileged monitor procedures.* A parent process must supply each of its children with a *function mask,* in which each bit specifies whether the child is allowed to perform a certain monitor function. The mask must be a subset of the parent's own mask.

At present the privileged functions include all monitor procedures that:

change the catalog on the backing store,
create and remove names of peripheral devices,
change the real-time clock.

Chapter 8

MONITOR FEATURES

This chapter is a survey of specific monitor features such as internal interruption, the real-time clock, conversational access from consoles, and permanent storage of files on the backing store. Although these are not essential primitive concepts, they are indispensable features of practical multiprogramming systems.

8.1. Internal Interruption

The monitor can assist internal processes with the detection of infrequent events such as violation of storage protection or arithmetic overflow. This causes an interruption of the internal process followed by a jump to an *interrupt procedure* within the process.

The interrupt procedure is defined by calling the monitor procedure:

set interrupt (interrupt address, interrupt mask)

When an internal interrupt occurs, the monitor stores the values of registers at the head of the interrupt procedure and continues execution of the internal process in the body of the procedure

interrupt address: working registers
instruction counter
interrupt cause
(execution continues here)

The system distinguishes between the following *causes* of internal interruption:

protection violation
integer overflow
floating-point overflow or underflow
parameter error in monitor call
breakpoint forced by parent

The *interrupt mask* specifies whether arithmetic overflow should cause internal interruption. Other kinds of internal interrupts cannot be masked off.

If an internal process provokes an interrupt without having defined an interrupt procedure after its creation, the monitor removes the process from the time slice queue and changes its state to *running after error.* The process does not receive any more computing time in this state, but from the point of view of other processes it is still an existing process. The parent of the erroneous process can, however, reactivate it by means of stop and start.

A parent can force a *breakpoint* in a child process as follows: first, stop the child; second, fetch the registers and interrupt address from the process description of the child and store the registers in the interrupt area together with the cause; third, modify the registers of the child to ensure that program execution continues in the interrupt procedure; fourth, start the child again.

8.2. Real-Time Clock

Real time is measured by means of a hardware interval timer, which counts modulo 16384 in units of 0.1 msec and interrupts the computer regularly (normally every 25.6 msec).

The monitor uses this timer to update a programmed *real-time clock* of 48 bits. This clock can be initialized and sensed by means of the procedures:

```
set clock (clock)
get clock (clock)
```

The setting of the clock is a privileged function. A standard convention adopted by operating systems (but not enforced by the monitor) is to let the clock express the time interval elapsed since midnight 31 December 1967 in units of 0.1 msec.

The interval timer is also used to implement an external process that permits the synchronization of internal processes with real time. All internal processes can send messages to this *clock process*. After the elapse of a time interval specified in the message, the clock process returns an answer to the sender. In order to avoid a heavy overhead time of clock administration, the clock process only examines its queue every second.

8.3. Console Communication

A multiprogramming system encourages a conversational mode of operation, in which users interact directly with internal processes from typewriter consoles. The external processes for consoles clearly reflect this objective.

Initially all program execution is ordered by human operators who communicate with the basic operating system. It would be very wasteful if the operating system had to examine all consoles regularly for possible operator requests. Therefore our first requirement is that consoles be able to activate internal processes by sending messages to them. Note that other external processes are only able to receive messages.

Second, it must of course be possible for an internal process to open a conversation with any console.

Third, a console should accept messages simultaneously from several internal processes. This will enable us to control more than one internal process from the same console, which is valuable in a small installation.

In short, consoles should be independent processes that can open conversations with any internal process and vice versa. The console should assist the operator with the identification of the internal processes using it.

An operator opens a conversation by depressing an interrupt key on the console. This causes the monitor to select a line buffer and connect it to the console. The operator must now identify the internal process to which his message is addressed. Following this he can input a message of one line, which is delivered in the queue of the receiving process.

A message to the basic operating system s can, for example, look like this (the word in italics is output by the console process in response to the key interrupt):

to s
new pbh run

An internal process opens a conversation with a console by sending a message to it. Before the input/output operation is initiated, the console identifies the internal process to the operator. This identification is suppressed after the first of a series of messages from the same process.

In the following example internal processes a and b share the same console for input/output. Process identifications are in italics:

to a
first input line to a
second input line to a
from b
first output line from b
second output line from b
from a
first output line from a
etc.

Note that these processes are unaware of their sharing the same console. From the point of view of internal processes the identification of user processes makes it irrelevant whether the system contains one or more consoles. (Of course one cannot expect operators to feel the same way about it).

8.4. Files on Backing Store

8.4.1. Introduction

The monitor permits semi-permanent storage of files on a backing store consisting of one or more drums and disks. The monitor makes these appear as a single backing store with a number of segments of 256 words each. This *logical backing store* is organized as a collection of named *data areas.* Each area occupies a consecutive number of segments on a single backing store device. A fixed part of the backing store is reserved for a *catalog* describing the names and locations of data areas.

Data areas are treated as external processes by the internal processes; input/output is initiated by sending messages to the areas specifying input/output operations, storage areas, and relative segment numbers within the areas. The identification of a data area requires a catalog search. In order to reduce the number of searches, input/output must be preceded by an explicit creation of an *area process* description within the monitor.

8.4.2. Catalog Entries

The catalog is a fixed area on the backing store divided into a number of *entries* identified by unique *names.* Each entry is of fixed length and consists of a *head,* which identifies the entry, and a *tail,* which contains the rest of the information. The

monitor distinguishes between entries describing data areas on the backing store and entries describing other things.

An entry is *created* by calling the monitor procedure:

create entry (name, tail, result)

The first word of the tail defines the *size* of an area to be reserved and described in the entry; if the size is negative or zero, no area is reserved. The rest of the tail contains nine *optional parameters,* which can be selected freely by the internal process.

Internal processes can *look up, change, rename,* or *remove* existing entries by means of the procedures:

look up entry (name, tail, result)
change entry (name, tail, result)
rename entry (name, new name, result)
remove entry (name, result)

The catalog describes itself in an entry named <:catalog:>.

The search for catalog entries is minimized by using a hashed value of names to define the first segment to be examined. Each segment contains 15 entries; thus most catalog searches only require the input of a single segment unless the catalog is filled to the brim. The allocation of data areas is speeded up by keeping a bit table of available segments within the monitor. In practice the creation or modification of an entry therefore requires only the input and output of a single catalog segment.

8.4.3. Catalog Protection

Since many users share the backing store as a common data base, it is vital that they have a means of protectiong their files gainst unintentional modification or complete removal. The protection system used is similar to the storage protection system: each catalog entry is supplied with a *catalog key* in its head; the rules of access within an internal process are defined by a *catalog mask* set by the parent of the internal process. Each bit in this mask corresponds to one of 24 possible catalog keys; if a bit is one, the internal process can modify or remove entries with the corresponding key; otherwise it can only look up these entries. A parent can only allocate a subset of its own catalog keys to a child process. Initially the basic operating system owns all keys.

In order to prevent the catalog and the rest of the backing store from being filled with irrelevant data, the concept of *temporary entry* is introduced. This is an entry that can be removed by another internal process as soon as the internal process that created the entry has been removed. Typical examples are working areas used during program compilation and data areas created, but not removed, by faulty programs.

This concept is implemented as follows. After creation of an internal process, the monitor increases an integer *creation number* by one and stores it within the new process description. Each time an internal process creates a catalog entry, the monitor includes its creation number in the entry head indicating that it is temporary. Internal processes can at any time scan the catalog and remove all temporary entries provided the corresponding creators no longer exist within the monitor. Thus in accordance

with our basic philosophy the monitor only provides the necessary mechanism for the handling of temporary entries, but leaves the actual strategy of removal to the hierarchy of processes.

In order to ensure the survival of a catalog entry, an internal process must call the privileged monitor function:

permanent entry (name, catalog key, result)

to replace the creation number with a catalog key. A process can of course only set one of its own keys in the catalog; otherwise it might fill the catalog with highly protected entries, which could be difficult to detect and remove.

8.4.3. Area Processes

In order to be used for input/output a data area must be looked up in the catalog and described as an external process within the monitor:

create area process (name, result)

The area process is created with the same name as the catalog entry.

Following this internal processes can send messages with the following format to the area process:

message: input/output operation
first storage address
last storage address
first relative segment

The reader is reminded that the tables used to describe area processes within the monitor are a limited resource, which is controlled by means of area claims defined by parent processes (Section 7.4).

The backing store is a random access medium that serves as a common data base. In order to utilize this property fully internal processes should be able to input simultaneously from the same area (e.g. when several copies of the Algol compiler are executed in parallel). On the other hand access to an area should be exclusive during output, because its content is undefined from the point of view of other processes.

Consequently we distinguish between internal processes that are *potential users* of an area process and the single process that may have *reserved* the area exclusively. This distinction was also made for peripheral devices (Section 5.2), but the rules of access are different here: An internal process is a user of an area after the creation of it. This enables the internal process to perform input as long as no other process reserves it. An internal process can reserve an area process if its catalog mask permits modification of the corresponding catalog entry. After reservation the internal process can perform both input and output.

Finally we should mention that the catalog is described permanently as an area process within the monitor. This enables internal processes to input and scan the catalog sequentially, for instance, during the detection and removal of temporary entries. Only the monitor itself, however, can perform output to the catalog.

Chapter 9

SYSTEM IMPLEMENTATION

This chapter gives important details about the implementation as well as figures about the size and performance of the system.

9.1. Interruptable Monitor Functions

Some of the monitor functions are too long to be executed entirely in the disabled mode, e.g. updating of the catalog on the backing store and creation, start, stop, and removal of processes. These so-called *process functions* are called as other monitor procedures, but behind the scenes they are executed by an anonymous internal process, which only operates in disabled mode for short intervals while updating monitor tables; otherwise the anonymous process shares computing time with other internal processes.

When an internal process calls a process function, the following takes place: the calling process is removed from the time slice queue and its state is changes to *waiting for process function.* At the same time the process description is linked to the event queue of the anonymous process that is activated. The anonymous process serves the calling processes one by one and returns them to the time slice queue after completion of each function.

Process functions are interruptable like other internal processes. From the point of view of calling processes, however, process functions are indivisible, since (1) they are executed only by the anonymous process one at a time in their order of request, and (2) calling processes are delayed until the functions are completed.

The following monitor procedures are implemented as interruptable functions:

create entry
look up entry
change entry
rename entry
remove entry
permanent entry
create area process
create peripheral process
create internal process
start internal process
stop internal process
modify internal process
remove process

9.2. Stopping Processes

According to theory an internal process cannot be stopped while input/output is in

progress within its storage area (Section 6.3). This requirement is inevitable in the case of high-speed devices such as a drum or a magnetic tape station, which are beyond program control during input/output. On the other hand it is not strictly necessary to enforce this for low-speed devices controlled by the monitor on a character-by-character basis.

In practice the monitor handles the stop situation as follows:

Before an external process initiates *high-speed input/output,* it examines the state of the sending process. If the sender is stopped (or waiting to be stopped), input/output is not initiated, but the external process immediately returns an answer with block length zero; the sender must then repeat input/output after restart. If the sender is not stopped, its stop count is increased and input/output is initiated. Note that if the stop count was increased immediately after the sending of a message, the sending process could only be stopped after completion of all previous operations pending in the external queue. By increasing the stop count as late as possible, we ensure that high-speed peripherals at most prevent the stopping of internal processes during a single block transfer.

Low-speed devices never increase the stop count. During output an external process fetches one word at a time from the sending process and outputs it character by character regardless of whether the sender is stopped meanwhile. Before fetching a word the external process examines the state of the sender. If it is stopped (or waiting to be stopped), output is terminated by an answer defining the actual number of characters output; otherwise output continues. During input an external process examines the state of the sender after each character. If the sender is stopped (or waiting to be stopped), input is terminated by an answer; otherwise the character is stored and input continues. Some devices, such as the typewriter, lose the last input character when stopped; others, such as the paper tape reader, do not. It can be seen that low-speed devices never delay the stopping of a process.

9.3. System Size

After initial system loading the monitor and the basic operating system s occupy a fixed part of the internal store. The size of a typical system is as follows:

		words:
monitor procedures:		2400
code for external processes:		1150
clock	50	
backing store	100	
typewriters	300	
paper tape readers	250	
paper tape punches	150	
line printers	100	
magnetic tape stations	200	
process descriptions and buffers:		1250
15 peripheral devices	350	
20 area processes	200	
6 internal processes	200	
25 message buffers	300	
6 console buffers	200	
basic operating system s		1400
total system		6200

It should be noted that the 6 internal processes include the anonymous process and the basic operating system, thus leaving room for 4 user processes.

As a minimum the standard programs (editor, assembler, and compilers) require an internal process of 5 – 6000 words for their execution. This means that a 16 k store can only hold the system plus 1 - 2 standard programs, while a 32 k store enables parallel execution of 4 such programs. A small store can of course hold more programs, if these are written in machine code and executed without the assistance of standard programs.

9.4. System Performance

The following execution times of monitor procedures are conservative estimates based on a manual count of instructions. The reader should keep in mind that the basic instruction execution time of the RC 4000 computer is 4 usec.

A complete conversation between two internal processes takes about 2 milliseconds distributed as follows:

	msec
send message	0.6
wait answer	0.4
wait message	0.4
send answer	0.6

It can be seen that one internal process can activate another internal process in 0.6 msec; this is also approximately the time required to activate an external process.

An analysis shows that the 2 msec required by an internal communication are used as follows:

	percent
validity checking	25
process activation	45
message buffering	30

This distribution is so even that one cannot hope to speed up the system by introducing additional, ad hoc machine instructions. The only realistic solution is to make the hardware faster.

The maximum time spent in the disabled mode within the monitor limits the system's response to real-time events. The monitor procedures themselves are only disabled for 0.2 – 1 msec. The situation is worse in the case of interrupt procedures that handle low-speed devices with hardware buffers, because the monitor empties or fills such buffers in the disabled mode after each interrupt. For the paper tape reader (flexowriter input) and the line printer, the worst-case figures are:

empty reader buffer (256 characters) 20 msec
fill printer buffer (170 characters) 7 msec

It should be noted, however, that these buffers normally only contain 64 - 70 characters corresponding to 4 - 5 msec. The worst-case situations can be remedied either by using smaller input/output areas within internal processes, or by replacing these external processes with dedicated internal processes (Section 5.4).

Finally we shall look at the interruptable monitor functions. An internal process of 5000 words can be created and controlled by a parent process with the following speed:

	msec
create internal process	3
modify internal process	2
start internal process	26
stop internal process	4
remove internal process	30

Most of the time required to start and remove an internal process is used to set storage protections keys.

Assuming that the backing store is a drum with a transfer time of 15 msec per segment, the catalog can be accessed with the following speed:

	msec
create entry	38
look up entry	20
change entry	38
rename entry	85
remove entry	38
permanent entry	38

The execution time of process functions should be taken with some reservations. First it must be remembered that process functions, like other internal processes, can be delayed for some time before they receive a time slice. In practice process functions will be activated immediately as long as they have not used a complete time slice (Section 7.2). Second one must take into consideration the fact that process function calls are queued within the monitor. Thus when a process wants to stop another process, the worst thing that can happen is that the anonymous process is engaged in updating the catalog. In this situation the stop is not initiated before the catalog has been updated. One also has to keep in mind that process functions share the drum or disk with other processes, and must wait for the completion of all input/output operations that preceed their own in the drum or disk queue. The execution times given here assume that process functions and catalog input/output are initiated instantly.

9.5. System Tape

The first version of the multiprogramming system consists of the monitor, the basic operating system s, and a program for initializing the catalog. It is programmed in the Slang 3 language. Before assembly the system is edited to include process descriptions of the peripheral devices connected to a particular installation and to define the following *options:*

number of storage bytes
number of internal processes
number of area processes
number of message buffers
number of console buffers
maximum time slice
inclusion of code for external processes
backing store configuration
size of catalog

The system is delivered in the form of a binary paper tape, which can autoload and initialize itself. After loading the system starts the basic operating system. Initially the operating system executes a program that can initialize the backing store with catalog entries and binary Slang programs input from paper tape. When this has been done, the operating system is ready to accept operator commands from consoles.

Chapter 10

SYSTEM POSSIBILITIES

The strength of the monitor is the generality of its basic concepts, its weakness that it must be supported by operating systems to obtain realistic multiprogramming. We beleive that the ultimate limits to the use of the system will depend on the imagination of designers of future operating systems. The purpose of this chapter is to stimulate creative thinking by pointing out a few of the possibilities inherent in the system.

10.1. Identification of Documents

In tape-oriented installations, operating systems should assist the operator with automatic identification of magnetic tapes. At present the external process concept gives the operator complete freedom to mount a magnetic tape on any station and identify it by name. When a tape station is set in the *local* mode, the monitor immediately removes its name to indicate that the operator has interfered with it. The station gives an interrupt when the operator returns it to the *remote* mode. Thus the monitor distinguishes between three states of a tape station:

document removed(after intervention)
unidentified document mounted (after remote interruption)
identified document mounted (after process creation)

It is a simple matter to introduce a *watch-dog process* in the monitor, to which internal processes can send messages in order to receive answers each time an unidentified tape is mounted somewhere. After reception of an answer, an internal process can give the actual station a temporary name, identify the tape by reading its label, and rename it accordingly.

Automatic identification requires general aggreement on the format of tape labels, at least to the extent of assigning a standard position to the names of tapes.

10.2. Temporary Removal of Programs

We have not imposed any restrictions on individual programs with respect to their demand for storage, run time, and peripherals. It is taken for granted that some programs will need most of the system resources for several hours. Such large programs must not, however, prevent other users from obtaining immediate access to the machine in order to execute more urgent programs of short duration. Thus the system must permit temporary removal of a program in order to make its storage area and peripherals available for other programs. One example, where this is absolutely necessary, is the periodic supervision of a real-time process combined with the execution of large background programs in idle intervals.

A program can be removed temporarily by stopping the corresponding internal process and dumping its storage area on the backing store by an output operation. Note that this dump automatically includes all children and descendants created

within the area. The monitor is only aware of the process being stopped; it is still described within the monitor and can receive messages from other processes.

It is now possible to create and start other processes in the same storage area, since the monitor does not check whether internal processes overlap each other as long as they remain within their parent processes. Peripherals can also be taken from the dumped process and assigned to others simply by mounting new documents and renaming the peripherals.

Temporary removal makes sense only if it is possible to restart a program at a later stage. This requires reloading the program into its original storage area as well as mounting and repositioning of its documents. After restart the internal process can detect interference with its documents in one of two ways: either it finds that a document does not exist any more, whereupon it must ask the operator to mount and name it; or it discovers that an existing document no longer is reserved by it, meaning that the operator has mounted it, but that it needs to be repositioned. These cases are indicated by the result parameter after a call of *wait answer*.

The need for repositioning can also arise during normal program execution, if the operator interferes with a peripheral device (by mistake or in order to move a document to a more reliable device). Consequently all major programs should consider each input/output operation as a potential restart situation.

10.3. Batch Processing

In the design of a batch processing system the distinction between parent and child processes prevents the batch of programs from destroying the operating system. Note that in general an operating system must remove a child process (and not merely stop it) to ensure that all its resources are released again (Section 7.4). Even then, it must be remembered that messages sent by a child to other processes remain in their queues until these processes either answer them or are removed (Section 4.4).

The multiprogramming capabilities can be utilized to accept job requests in a conversational mode during execution of the batch. Thus a *batch processing* system can include facilities for *remote job entry* combined with *priority scheduling* of programs.

10.4. Time-Sharing

The basic requirement of a *time-sharing* system, in which a large number of users have conversational access to the system from consoles, is the ability to swap programs between the internal store and the backing store. A time-sharing operating system must create an internal process for each user, and make these processes share the same storage area by frequent removal and restart of programs (say, every few seconds). The problem is that stopping a process temporarily also means stopping its communication with peripherals. Thus in order to keep typewriter input/output alive while a user process is dumped, the system must include an internal process that buffers all data between programs and consoles.

10.5. Real-Time Scheduling

We conclude these hints with an example of a *real-time* system. The application we have in mind is a process control system, in which a number of programs must perform data logging, alarm scanning, trend logging, and so forth periodically under the real-time control of an operating system.

This can be organized as follows: initially all task programs send messages to the operating system and wait for answers. The operating system communicates with the clock process and is activated every second in order to scan a time table of programs. If the real time exceeds the start time of a task program, the operating system activates the program by an answer. After completion of its task, the program again sends a message to the operating system and waits for the answer. In response the operating system increases the start time of the program by the period between two successive executions of the task.

PART II.
MONITOR FUNCTIONS

Chapter 11

GENERAL MONITOR CONVENTIONS

This chapter explains the general rules of monitor calls, internal interruption, privileged functions, names, and catalog protection.

11.1. Monitor Call

The monitor is called within an internal process by the execution of an instruction with the format:

jd 1<11 + <unsigned integer>

after loading parameter values in the working registers.

The monitor checks that the <unsigned integer> is the number of a monitor procedure.

Internal interrupts not recognized as monitor calls cause immediate switching to the interrupt procedure of the internal process (Section 11.2).

The monitor also checks the validity of procedure parameters. The basic rule is that the return address and the addresses of stored parameters must be within the calling process; otherwise the monitor provokes an internal interruption of the process.

The following example shows the notation used to define the call of a monitor procedure:

```
look up entry (name address, tail address, result)
w0    result (return)
w1    tail address (call)
w2
w3    name address (call)
      jd 1<11+42
name address:     entry name (call)
tail address:     number of segments (return)
      + 2         9 optional words (return)
```

The procedure *look up entry* is called by executing the instruction jd 1<11+42 with registers w1 and w3 containing the adresses of two storage areas, called *tail* and *name*, within the calling process. The name area contains an *entry name* when the procedure is called.

Upon return from the monitor procedure, register w0 contains a *result* parameter, while the other registers are unchanged. The monitor has stored a parameter, called *number of segments*, and 9 *optional words* in the tail area.

11.2. Interrupt Handling

An internal process can call the monitor and define the address of an interrupt procedure to be called when the process provokes an internal interrupt not recognized as a monitor call.

When the interrupt occurs, current register values are stored in the head of the interrupt procedure and execution of the internal process continues within the interrupt procedure:

interrupt address:	working register 0
+ 2	working register 1
+ 4	working register 2
+ 6	working register 3
+ 8	exception register
+ 10	instruction counter
+ 12	interrupt cause
+ 14	first instruction executed after interruption

The *interrupt cause* has one of the following values:

0	protection violation
2	integer overflow
4	floating-point overflow/underflow
6	parameter error in monitor call
8	breakpoint forced by parent

An *interrupt address* equal to zero indicates that no interrupt procedure is defined by the process. If such a process provokes an internal interrupt, the process is removed from the timer queue and set in the state *running after error.* The process does not receive any more computing time in this state, but from the point of view of other processes it is still an existing process. The parent of the process can, however, reactivate it by means of stop and start.

The monitor defines a standard value of the *interrupt mask* used during execution of an internal process to ensure that all external interrupts are served. An internal process can, however, call the monitor and define the interrupt mask bits corresponding to integer and floating-point overflow. If these interrupts are masked off, they will not cause internal interruption of the process. Other kinds of internal interrupts cannot be masked off.

An internal process is created with the interrupt address zero and with arithmetic interrupts masked off.

11.3. Function Mask

During creation of an internal process the parent specifies a *function mask* in which each bit defines whether the child process is allowed to call one of the privileged monitor procedures. At present the function mask bits have the following meaning (1 = function call allowed, 0 = function call forbidden):

bit:	function:
0	create entry
1	change entry, remove entry
2	rename entry
3	permanent entry
4	create peripheral process
5	remove peripheral process
6	generate name
7	set clock
8	can receive last console buffer

11.4. Names

A *name* is a textstring of 12 ISO characters (equal to 4 words) beginning with a small letter followed by a maximum of 10 small letters or digits terminated by NULL characters.

Names are used to identify processes and catalog entries.

A *process name* must be unique among all internal and external processes.

An *entry name* must be unique among all catalog entries.

An *area process name* must be unique among all process names and entry names.

During creation of a process or a catalog entry the monitor can either use a name specified by the calling process or it can generate a name. A *generated name* has the format:

wrk<6 digits>

followed by three NULL characters.

11.5. Catalog Protection

All catalog entries are created as temporary entries with the catalog key zero and the creation number of the calling internal process.

An internal process can only modify or remove a catalog entry and its associated data area if the following conditions are fulfilled:

1. The catalog register of the internal process must have a one in the bit corresponding to the catalog key of the entry.
2. The entry must not be a temporary entry created by another existing internal process.
3. A data area described by the entry must not exist as a process reserved by another internal process.

If an internal process changes the size or catalog key of an entry, a corresponding area process is updated accordingly.

If an internal process renames or removes an entry, a corresponding area process is removed.

Chapter 12

DEFINITION OF MONITOR PROCEDURES

This chapter defines the functions of monitor procedures and the conventions for calling them within Slang programs.

12.1. Procedure Set Interrupt

```
set interrupt (interrupt address, interrupt mask)
w0    interrupt mask (call)
w1
w2
w3    interrupt address (call)
      jd 1<11+0
interrupt address: (see Section 11.2)
```

Defines the internal interrupt address of the calling process and the *interrupt mask* bits 1 and 2 for integer and floating-point overflow, the remaining mask bits are irrelevant in the call; they are always set to a standard value by the monitor.
The *interrupt address* must either be zero or point to an area within the calling process.

Parameter error: interrupt area or return address outside calling process.

12.2. Procedure Process Description

```
process description (name address, result)
w0   result (return)
w1
w2
w3   name address (call)
     jd 1<11+4
name address: process name (call)
```

Checks the existence of a process with a given *name*. The *result* is the address of the process description within the monitor, if the process exists; otherwise the result is zero.

result	=0	process does not exist
	>0	process description address

Parameter error: process name or return address outside calling process.

12.3. Procedure Initialize Process

```
initialize process (name address, result)
w0   result (return)
w1
w2
w3   name address (call)
     jd 1<11+6
name address: process name (call)
```

Checks the existence of a process with a given *name* and prepares it for communication with the calling process. The initializing depends on the *kind* of the process as follows:

Internal Process, Clock, or *Typewriter:* Only the existence of the process is checked.

Other Peripheral Process: The existence of the peripheral process with the calling process as a user is checked. If the peripheral process controls a sequential document, it is reserved for the calling process, unless another process has reserved it.

Area Process: The existence of the area process is checked. If the calling process is not a user of the area process, it is defined as a user and its area claim decreased by one. After initialization, only input from the area is allowed.

Messages received by the process before initialization will still be processed and answered.

result =	0	process initialized
	1	reserved by another process
	2	calling process is not a user; area claim exceeded
	3	process does not exist

Parameter error: process name or return address outside calling process.

12.4. Procedure Reserve Process

```
reserve process (name address, result)
w0   result (return)
w1
w2
w3   name address (call)
     jd 1<11+8
name address: process name (call)
```

Reserves a process with a given *name* for exclusive communication with the calling process. Reservation depends on the *kind* of the process as follows:

Internal Process, Clock, or *Typewriter:* Reservation is not allowed.

Other Peripheral Process: The existence of the peripheral process with the calling process as a user is checked. It is reserved for the calling process unless another process has reserved it.

Area Process: The existence of the area process is checked. If the calling process is not a user of the area process, it is defined as a user and its area claim decreased by one. The area process is reserved for the calling process provided no other process has reserved it, and provided the area is not protected against the calling process, i.e. the catalog mask of the calling process must have a one in the bit corresponding to the catalog key of the area. After reservation both input and output to the area are allowed.

Messages received by the process before reservation will still be processed and answered.

```
result =   0  process reserved
           1  reserved by another process
           2  calling process is not a user; area claim exceeded;
              process cannot be reserved
           3  process does not exist
```

Parameter error: process name or return address outside calling process.

12.5. Procedure Release Process

```
release process (name address)
w0
w1
w2
w3    name address (call)
      jd 1<11+10
name address: process name (call)
```

Releases a possible reservation of a process with a given *name*. Release depends on the *kind* of the process as follows:

Internal Process: Nothing is done.

External Process: If the external process exists and is reserved by the calling process, the reservation is cancelled.

Messages received by the process before release will still be processed and answered.

Parameter error: process name or return address outside calling process.

12.6. Procedure Include User

include user (name address, device number, result)
w0 result (return)
w1 device number (call)
w2
w3 name address (call)
jd 1<11+12
name address: process name (call)

Includes an internal process with a given *name* as a user of a peripheral device with a given *number*. The internal process must be a child of the calling process and the latter must be a user of the device.

result = 0 child included as a user
2 calling process is not a user
3 described process is not a child
4 device number does not exist

Parameter error: process name or return address outside calling process.

12.7. Procedure Exclude User

```
exclude user (name address, device number, result)
w0   result (return)
w1   device number (call)
w2
w3   name address (call)
     jd 1<11+14
name address: process name (call)
```

Excludes an internal process with a given *name* as a user and reserver of a peripheral device with a given *number*. The internal process must be a child of the calling process and the latter must be a user of the device.

Messages received by the peripheral process before exclusion of the user will still be processed and answered.

```
result =   0  child excluded as a user
           2  calling process is not a user
           3  described process is not a child
           4  device number does not exist
```

Parameter error: process name or return address outside calling process.

12.8. Procedure Send Message

```
send message (name address, message address, buffer address)
w0
w1    message address (call)
w2    buffer address (return)
w3    name address (call)
      jd 1<11+16
name address: process name (call)
      + 8      name table entry (return)
message address: 8 words (call)
```

Selects an available *message buffer,* decreases the buffer claim of the calling process by one, and copies a *message* of eight words into the buffer. The message is delivered in the queue of a receiving process with a given *name.* The receiving process is activated if it is waiting for a message or an event. The calling process continues after being informed of the address of the message buffer.

If the receiving process does not exist, the monitor generates a dummy answer to the message as described in Section 12.9.

The format and interpretation of a message depend on the kind of the receiving process. For external processes details are given in Chapter 13.

After a call of send message the monitor stores an address, the *name table entry,* after the process name. This address is used in subsequent calls of send messages to speed up identification of the receiver. The name table entry can be destroyed by the calling process, but this will slow down the next call of send message.

The *process description address* of the receiver can be loaded by indirect addressing of the name table entry, for example:

```
rl  w1 (name address+8)
```

If the buffer claim of the calling process is exceeded the buffer address is zero:

```
buffer address  =0  buffer claim exceeded
                >0  selected buffer address
```

Parameter error: process name, message, or return address outside calling process.

12.9. Procedure Wait Answer

```
wait answer (result, answer address, buffer address)
w0   result (return)
w1   answer address (call)
w2   buffer address (call)
w3
     jd 1<11+18
answer address: 8 words (return)
```

Delays the calling process until an answer arrives in a given message buffer. On arrival the *answer,* consisting of 8 words, is copied into the calling process. The *message buffer* is released and the buffer claim of the calling process is increased by one.

The message buffer must be one that has been assigned to the calling process during a previous call of send message.

The format of an answer depends on the kind of the process that has received and answered the message. For external processes details are given in Chapter 13.

The *result* specifies whether a normal or dummy answer was received. A *normal answer* is delivered by a process in response to a received intelligible message. A *dummy answer* is generated by the monitor when a message is addressed to a non-existent process. A dummy answer can also be generated by the receiving process in response to an undesired or unintelligible message, or when a malfunction (such as a disconnected peripheral device) prevents completion of an operation specified in a message.

```
result =   1 normal answer
           2 dummy answer, message rejected
           3 dummy answer, message unintelligible
           4 dummy answer, receiver malfunction
           5 dummy answer, receiver does not exist
```

Parameter error: buffer address does not point at message buffer assigned to calling process; answer area or return address outside calling process.

12.10. Procedure Wait Message

```
wait message (name address, message address, buffer address, result)
w0   result (return)
w1   message address (call)
w2   buffer address (return)
w3   name address (call)
     jd 1<11+20
name address: process name (return)
message address: 8 words (return)
```

Delays the calling process until a message arrives in its queue. On arrival the *name* of the sending process and the *message,* consisting of 8 words, are copied into the calling process. The address of the *message buffer* in which the message was transmitted is delivered as a return parameter. The buffer is removed from the queue and is now ready to transmit an answer.

The *result* is the process description address of the sender, if it still exists. If the sender has been removed, the result is the negative process description address of its parent; in this case the sender name is not copied into the calling process.

```
result    >0   process description address
          <0   -parent description address
```

Parameter error: name area, message area, or return address outside calling process.

12.11. Procedure Send Answer

```
send answer (result, answer address, buffer address)
w0   result (call)
w1   answer address (call)
w2   buffer address (call)
w3
     jd 1<11+22
answer address: 8 words (call)
```

Copies an *answer* of eight words into a *message buffer* in which a message has been received and delivers it in the queue of the original sender. The sender of the message is activated if it is waiting for an answer or event. The *result* defined by the calling process is delivered to the sender as the result of wait answer; it specifies whether the answer is normal or dummy.

If the sender no longer exists, the message buffer is released and the buffer claim of the parent of the sender is increased by one.

At present the sender can be either an internal process or a typewriter. An answer to a *typewriter* message merely causes the message buffer and the associated line buffer to be released (see Section 13.6).

result =	1	normal answer
	2	dummy answer, message rejected
	3	dummy answer, message unintelligible
	4	dummy answer, receiver malfunction

Parameter error: buffer address does not point at message buffer received by calling process; answer area or return address outside calling process; illegal result value.

12.12. Procedure Wait Event

```
wait event (last buffer address, next buffer address, result)
w0    result (return)
w1
w2    last buffer address (call)
      next buffer address (return)
w3
      jd 1<11+24
```

Delays the calling process until either a message or an answer arrives in its queue after the buffer given by the *last buffer address.* The process is supplied with the address of the *next buffer* and a *result* indicating whether it contains a message or an answer. If the last buffer address is zero, the queue is examined from the start.
The wait event procedure does not remove the next buffer from the queue or in any other way change its status. This must be done by one of the procedures wait answer or get event.

```
result =  0  message
          1  answer
```

Parameter error: last buffer address does not point at message buffer in the queue of the calling process; return address outside calling process.

12.13. Procedure Get Event

```
get  event (buffer address)
w0
w1
w2   buffer address (call)
w3
     jd 1<11+26
```

Removes a given *buffer* from the queue of the calling process. If the buffer contains a message, it is made ready for the sending of an answer. If the buffer contains an answer, it is released and the buffer claim of the calling process is increased by one.
The copying of information from the buffer (sender description address, message or answer, and result) must be done by the calling process itself before get event is called. This requires a knowledge of the format of a message buffer (see Appendix A.3).

Parameter error: buffer address does not point at message buffer in the queue of the calling process; return address outside calling process.

12.14. Procedure Type Working Register

```
type w0        jd 1<11+28
type w1        jd 1<11+30
type w2        jd 1<11+32
type w3        jd 1<11+34
```

These four procedures print the contents of a working register as a signed integer preceded by the letter w, x, y, or z, followed by a new line character. The register value is printed in disabled mode on the main typewriter. These procedures are only used during testing of the system; they are not included in final versions of the monitor.

Parameter error: return address outside calling process; procedure not included in monitor.

12.15. Procedure Get Clock

```
get clock (clock)
w0    clock(0:23) (return)
w1    clock(24:47) (return)
w2
w3
      jd 1<11+36
```

Delivers an updated value of the 48-bit *clock* with unit position 0.1 millisecond.

Parameter error: return address outside calling process.

12.16. Procedure Set Clock

```
set clock (clock)
w0      clock (0:23) (call)
w1      clock (24:47) (call)
w2
w3
            jd 1<11+38
```

Initializes the 48-bit *clock* with unit position 0.1 millisecond. This is a privileged function.

It should be noted that clock values stored in the process descriptions of internal processes *(start time* and *wait time)* are unchanged after a call of set clock (see Appendix A.6).

Parameter error: return address outside calling process; function forbidden in calling process.

12.17. Procedure Create Entry

```
create entry (name address, tail address, result)
w0    result (return)
w1    tail address (call)
w2
w3    name address (call)
      jd 1<11+40
name address:  entry name (call)
tail address:  number of segments (call)
      + 2      9 optional words (call)
```

Creates a new entry in the catalog with *name* and *tail* as specified in the call. If the first word of the name is zero, the monitor generates a unique entry name and stores it in the name area specified in the call. If the first word of the tail is greater than zero, an *area* of that size is reserved on the backing store and its first segment number placed in the entry head. The entry is created as a *temporary entry* with the *catalog key* zero and supplied with the *creation number* of the calling process. This is a privileged function.

```
result =   0   entry created
           1   catalog function forbidden in calling process
           2   catalog input/output error
           3   entry with same name already exists
           4   catalog full
           5   contiguous area of requested size not available
           6   name format illegal
```

Parameter error: name, tail area, or return address outside calling process.

12.18. Procedure Look Up Entry

```
look up entry (name address, tail address, result)
w0    result (return)
w1    tail address (call)
w2
w3    name address (call)
      jd 1<11+42
name address:  entry name (call)
tail address:  number of segments (return)
    + 2        9 optional words (return)
```

Looks up an entry in the catalog with a given *name* and copies the entry *tail* into the calling process.

result =	0	entry looked up
	2	catalog input/output error
	3	entry does not exist
	6	name format illegal

Parameter error: name, tail area, or return address outside calling process.

12.19. Procedure Change Entry

```
change entry (name address, tail address, result)
w0          result (return)
w1          tail address (call)
w2
w3          name address (call)
            jd 1<11+44
name address:   entry name (call)
tail address:   number of segments (call)
    + 2         9 optional words (call)
```

Changes the *tail* of an entry in the catalog with a given *name* provided the catalog register permits the calling process to modify the entry, and provided the entry is not a temporary entry created by another existing process.

If the entry describes an *area* on the backing store, the size of the area, given by the first word of the tail, can be reduced, but not increased. The area is reduced from the upper end, that is, the first segment number remains unchanged.

If the area is used as a *process*, it must not be reserved by another internal process. The area process description is changed in accordance with the new size.

If the entry does not describe an area, the first word of the new tail must be less than or equal to zero.

This is a privileged function.

```
result =  0   entry changed
          1   catalog function forbidden in calling process
          2   catalog input/output error
          3   entry does not exist
          4   entry is protected against calling process
          5   area process reserved by another user
          6   name format or new size illegal
```

Parameter error: name, tail area, or return address outside calling process.

12.20. Procedure Rename Entry

```
rename entry (name address, new name address, result)
w0      result (return)
w1      new name address (call)
w2
w3      name address (call)
        jd 1<11+46
name address: old entry name (call)
new name address: new entry name (call)
```

Changes the *name* of an entry in the catalog provided the catalog register permits the calling process to modify the entry, and provided the entry is not a temporary entry created by another existing process.

This procedure leaves the catalog key, the creation number, the first segment number, and the entry tail unchanged.

If the entry describes an *area* used as a *process*, it must not be reserved by another internal process. The area process description is removed and the area claim of all users is increased by one.

This is a privileged function.

result =	0	entry renamed
	1	catalog function forbidden in calling process
	2	catalog input/output error
	3	old entry does not exist or entry with new name already exists
	4	old entry is protected against calling process
	5	area process reserved by another user
	6	old or new name format illegal

Parameter error: old or new name or return address outside calling process.

12.21. Procedure Remove Entry

```
remove entry (name address, result)
w0      result (return)
w1
w2
w3      name address (call)
        jd 1<11+48
name address: entry name (call)
```

Removes an entry in the catalog with a given *name* and its associated area on the backing store provided the catalog register permits the calling process to modify the entry, and provided the entry is not a temporary entry created by another existing process.

If the entry describes an *area* used as a *process*, it must not be reserved by another internal process. The area process description is removed and the area claim of all users is increased by one.

This is a privileged function.

result =	0	entry removed
	1	catalog function forbidden in calling process
	2	catalog input/output error
	3	entry does not exist
	4	entry is protected against calling process
	5	area process reserved by another user
	6	name format illegal

Parameter error: name or return address outside calling process.

12.22. Procedure Permanent Entry

```
permanent entry (name address, catalog key, result)
w0      result (return)
w1      catalog key (call)
w2
w3      name address (call)
        jd 1<11+50
name address: entry name (call)
```

Changes the *catalog key* of an entry in the catalog and sets the *creation number* to zero provided the catalog register permits the calling process to modify the entry both before and after the change of the key, and provided the entry is not a temporary entry created by another existing process.

This procedure leaves the entry name, the first segment, and the entry tail unchanged. If the entry describes an *area* used as a *process*, it must not be reserved by another internal process. The area process description is changed in accordance with the new catalog key.

This is a privileged function.

result =	0	entry permanent
	1	catalog function forbidden in calling process
	2	catalog input/output error
	3	entry does not exist
	4	old or new catalog key protects entry against calling process
	5	area process reserved by another user
	6	name format illegal

Parameter error: name or return address outside calling process.

12.23. Procedure Create Area Process

```
create area process (name address, result)
w0          result (return)
w1
w2
w3          name address (call)
            jd 1<11+52
name address: process name (call)
```

Makes an *area* on the backing store available to the calling process as an *external process* with the same *name* as the corresponding catalog entry. If the area process does not exist, it is looked up in the catalog and described as a process within the monitor with the calling process as a user whose area claim is decreased by one. If the area process already exists, the calling process is defined as a user and its area claim decreased by one.

result =	0	area process created
	1	area claim exceeded
	2	catalog input/output error
	3	entry does not exist; process with same name already exists
	4	entry does not describe area
	6	name format illegal

Parameter error: name or return address outside calling process.

12.24. Procedure Create Peripheral Process

```
create peripheral process (name address, device number, result)
w0   result (return)
w1   device number (call)
w2
w3   name address (call)
     jd 1<11+54
name address: process name (call)
```

Assigns a process *name* to a peripheral *device.* If the first word of the name is zero, the monitor generates a unique process name and stores it in the name area specified in the call. It is required that the calling process is a user of the device and that no other process has reserved it.

This is a privileged function.

result =	0	peripheral process created
	1	function forbidden in calling process
	2	calling process is not a user
	3	process with same name already exists
	4	device number does not exist
	5	device is reserved by another user
	6	name format illegal

Parameter error: name or return address outside calling process.

12.25. Procedure Create Internal Process

```
create internal process (name address, parameter address, result)
w0   result (return)
w1   parameter address (call)
w2
w3   name address (call)
     jd 1<11+56
name address: process name (call)
parameter address:    first storage address (call)
     + 2              top storage address (call)
     + 4              buffer claim, area claim (call)
     + 6              internal claim, function mask (call)
     + 8              catalog mask (call)
     + 10             protection register, protection key (call)
```

Creates a description of an internal process with a given *name* and *parameters.* If the first word of the name is zero, the monitor generates a unique process name and stores it in the name area given in the call.

The storage area must be within the storage area of the calling process. The *first and top storage addresses* point to the first storage word and the last storage word + 2 of the new process.

The *buffer claim* and *area claim* must be less than or equal to the claims of the calling process, which are decreased accordingly.

The *internal claim* must be less than the claim of the calling process, which is decreased by the value specified + 1 to include the new process itself.

In the *function and catalog masks* a bit equal to one indicates that the process can call the privileged function or change the catalog entries with the corresponding key. Consequently the mask bits equal to one must be a subset of those defined for the calling process.

In the *protection register* a bit equal to zero indicates the ability to change or execute storage words with the corresponding key. Consequently the protection bits equal to zero must be a subset of those defined for the calling process.

The *protection key* is the key that is set in all storage words of the new process when it is started. The corresponding bit in the protection register must be zero; except when the protection key of the calling process is zero; in that case any protection situation of the new process is accepted.

The new process is a *child* of the calling process. It is created in the state *waiting for start by parent,* with *stop count, interrupt address, time quantum,* and *run time* equal to zero, while the *instruction counter* is equal to the first storage address. The *interrupt mask* is initialized to prevent internal interruption on arithmetic overflow. The *start time* of the process is set equal to the current value of the 48-bit clock. Finally the *creation number* is increased by one and stored in the process description.

result = 0 internal process created
1 storage area outside calling process; claim exceeded; illegal protection
3 process with same name already exists
6 name format illegal

Parameter error: name, parameters, or return address outside calling process.

12.26. Procedure Start Internal Process

```
start internal process (name, address, result)
w0          result (return)
w1
w2
w3          name address (call)
            jd 1<11+58
name address: process name (call)
```

Starts an internal process with a given *name* provided it is a *child* of the calling process in the state *waiting for start by parent.* Also started are all *descendants* of the child that were stopped previously along with it. The start includes a setting of the *protection key* in the storage area of the involved processes. It is ensured that no process is started before its parent has been started. The processes are set in the *running* state.

Finally the *stop count* of the calling process is increased by one to indicate that a process, the child, is modifying its storage area.

result =	0	internal process started
	2	state of process does not permit start
	3	described process is not a child
	6	name format illegal

Parameter error: name or return address outside calling process.

12.27. Procedure Stop Internal Process

```
stop internal process (name address, buffer address, result)
w0   result (return)
w1
w2   buffer address (return)
w3   name address (call)
     jd 1<11+60
name address: process name (call)
```

Initiates a stop of an internal process with a given *name* provided it is a *child* of the calling process. Also stopped are all running *descendants* of the child. When the stop is initiated, the child is placed in the state *waiting for stop by parent,* while its descendants are set to *waiting for stop by ancestor.* A process can be in any state when the stop is initiated. If the process is waiting for a message, an answer, an event, or a process function, its state is changed to waiting for stop, as explained above, but at the same time its instruction counter is decreased by two to ensure that it will repeat the call of the corresponding monitor procedure when it is started again.

Finally, the stop procedure selects a *message buffer,* decreases the buffer claim of the calling process by one, and returns.

The monitor completes the stop operation as follows: Each time the *stop count* of one of the involved processes becomes zero (after the completion of high-speed input/output), the process is transferred to the state *waiting for start by ancestor* (or *parent*) and the stop count of its parent is decreased by one. When all involved processes are stopped, the monitor delivers an *answer* in the selected message buffer to the calling process, which in turn receives it by calling either wait answer or wait event.

If the buffer claim of the calling process is exceeded the buffer address is zero:

buffer address	=0	buffer claim exceeded
	>0	selected buffer address
result	=0	stop initiated
	3	described process is not a child
	6	name format illegal

Parameter error: name or return address outside calling process.

12.28. Procedure Modify Internal Process

```
modify internal process (name address, register address, result)
w0    result (return)
w1    register address (call)
w2
w3    name address (call)
      jd 1<11+62
name address:      process name (call)
register address:  working register 0 (call)
      +2           working register 1 (call)
      +4           working register 2 (call)
      +6           working register 3 (call)
      +8           exception register (call)
      +10          instruction counter (call)
```

Initializes the *register values* of an internal process with a given *name* provided it is a child of the calling process in the state *waiting for start by parent.*

```
result =   0    internal process modified
           2    state of process does not permit modification
           3    described process is not a child
           6    name format illegal
```

Parameter error: name, registers, or return address outside the calling process; instruction counter outside child process.

12.29. Procedure Remove Process

```
remove process (name address, result)
w0    result (return)
w1
w2
w3    name address (call)
      jd 1<11+64
name address: process name (call)
```

Removes an internal or external process from the monitor. Removal depends on the *kind* of the process:

Peripheral Process: The calling process must be a user of the peripheral process and no other process must have reserved it. In this case the name of the peripheral process is removed, but the device itself remains described within the monitor. This is a privileged function.

Area Process: The calling process must be a user of the area process. In this case the calling process is removed as a user and reserver of the area process, and its area claim is increased by one. If the area process has no other users, the process description is also removed.

Internal Process: The internal process must be a *child* of the calling process in the state *waiting for start by parent.* In this case the child and its *descendants* are removed as follows: the process descriptions are removed one by one starting with the youngest descendants, and the *buffer claims, area claims,* and *internal claims* are added to those of their parents. In the storage area of the child, the *protection key* is reset to the value used within the calling process.

All *area processes* used by the involved internal processes are removed as described above.

In all *peripheral processes* the internal processes are removed as users and reservers, but the peripheral processes themselves are not removed.

Finally all *message buffers* are examined for pending messages or answers involving the processes to be removed. There are the following possibilities:

a) A buffer contains a pending or received message to a removed process. A dummy answer (receiver does not exist) is delivered if the sending process still exists; otherwise the buffer is released and the buffer claim of the parent of the sender is increased by one.

b) A buffer contains a pending answer to a removed process. The buffer is released and the buffer claim of the parent of the removed process is increased by one.

c) A buffer contains a pending or received message from the removed process to another process. The message is left undisturbed with an indication that the buffer claim of the parent of the removed process should be increased by one when the receiving process sends an answer.

d) A buffer contains an answer from the removed process to another process. The answer is left undisturbed.

result =	0	process removed
	1	function forbidden in calling process
	2	state of internal process does not permit removal; calling process is not a user of external process
	3	described process is not a child or does not exist
	5	peripheral process reserved by another process
	6	name format illegal

Parameter error: name or return address outside calling process.

12.30. Procedure Testcall

```
testcall
w0    1 (return)
w1
w2
w3
      jd 1<11+66
```

Types internal testoutput of process functions in disabled mode on the main typewriter. This procedure is dummy in final versions of the monitor.

Parameter error: return address outside calling process.

12.31. Procedure Generate Name

```
generate name (name address, result)
w0   result (return)
w1
w2
w3   name address (call)
     jd 1<11+68
name address: generated name (return)
```

Generates a *name* with the format:

```
wrk<6 digits>
```

followed by three NULL characters and stores it within the calling process. The six digits form an octal number modulo 8.777 777.

The generated name does not exist as a process name or a catalog entry name.

This is a privileged function.

result =	0	name generated
	1	function forbidden in calling process
	2	catalog input/output error

Parameter error: name or return address outside calling process.

Chapter 13

DEFINITION OF EXTERNAL PROCESSES

This chapter defines the functions of external processes and the conventions for calling them within Slang programs.

13.1. Process Kind

At present the following *kinds* of processes are defined:

0	internal process
2	interval clock
4	backing store area
6	drum (RC 4320)
8	typewriter (RC 315)
10	paper tape reader (RC 2000)
12	paper tape punch (RC 150)
14	line printer (RC 610)
16	punched card reader (RC 405)
18	magnetic tape station (RC 747)

13.2. Input/Output Messages

Input/Output is initiated when an internal process sends a *message* to an external process. A typical message has the following format:

message address:	operation
+ 2	first storage address
+ 4	last storage address

It defines an input/output operation and the first and last address of a data block that must be within the storage area of the internal process.

After completing the operation the external process returns an *answer*, which, typically, has the following format:

answer address:	status word
+ 2	number of bytes transferred
+ 4	number of characters transferred

It contains a status word sensed from the device after the operation, the actual number of bytes input or output (always an even number), and the corresponding number of characters (a word can contain 2, 3, or 4 characters depending on the kind of device).

Dummy answers to input/output messages are delivered in the following situations:

result = 2, *message rejected:* the external process is reserved by another process, or demands a reservation that has not been made by the sending process;

result = 3, *message unintelligible:* the operation parameter has an illegal value, or the storage area for input/output is outside the sending process;

result = 4, *receiver malfunction:* the peripheral process is disconnected during the input/output operation;

result = 5, *receiver does not exist.*

13.3. Interval Clock

General Rules.

The process *kind* is 2. Operations can be initiated by any internal process. *Initialization* has no effect, *reservation* is forbidden. The clock accepts messages simultaneously from more than one internal process.

Delay Operation.

A message to the clock specifies a time interval either in seconds or in 0,1 milliseconds. After the elapse of the interval the clock returns an answer. The time interval must be from 0 to 24 hours, otherwise the message is treated as unintelligible.

The clock process is synchronized with the hardware interval timer; it is activated when a multiple of time slices exceed the monitor constant, *inspection interval;* it then decreases all delays in its queue by this value and returns answers for delays that become zero or negative.

Thus the actual delay can have a maximum error equal to the value of the *inspection interval* depending on whether a message is sent at the beginning or at the end of the *inspection interval.* Exact time intervals can only be measured by means of the monitor procedure *get clock.*

The value of the *inspection interval* is normally one second but can be redefined during translation of the monitor. The lower limit for the value is one time slice but for practical use the value should not be less than 0.1 second.

Message and Answer:

operation:	message:	answer:
delay in seconds	0	0
	seconds	0
		0
delay in 0.1 msecs	2	0
	interval (0:23)	0
	interval (24:47)	0

13.4. Backing Store Area

General Rules.

The process *kind* is 4. Sense and input operations can be initiated by an internal process that is a *user* of the area process. An area process accepts input messages simultaneously from more than one process provided no process has *reserved* it. Output operations require that the area process has been reserved.

Sense Operation.

The device on which the area is stored is sensed and the status word is delivered as an answer.

Input Operation.

A number of consecutive segments of 256 words each is input to a storage area within the sending process. The first segment number relative to the beginning of the area on the backing store is also given in the message.

The operation transfers the maximum number of segments for which there is room within the storage area, i.e.

number of segments =
(last storage address + 2 − first storage address)//512

If the input block thus specified exceeds the upper limit of the area on the backing store, input is performed only of that part of the block that is within the area. In any case the actual number of bytes transferred is given in the answer.

The number of bytes transferred is a multiple of 512. The number of characters is defined as three times the number of words transferred.

If the answer indicates *parity error*, *synchronization error*, or *data overrun*, the operation has been performed three times without success.

If the first segment is outside the limits of the area, input is not initiated, but the answer contains a status bit, *end of area*, and the block length zero.

Output Operation.

Equivalent to the input operation.

Status Bits.

1	parity error
2	synchronization error
3	data overrun
5	end of area (generated by monitor)

Messages and Answers.

operation:	message:	answer:
sense	0	status word
		0
		0

input	3<12	status word
	first storage address	number of bytes
	last storage address	number of characters
	first segment number	
output	5<12	status word
	first storage address	number of bytes
	last storage address	number of characters
	first segment number	

13.5. Drum (RC 4320)

General Rules.

The process *kind* is 6. At present the drum is only used in connection with areas described in the catalog. The drum itself is therefore not accessible as a separate process within internal processes.

13.6. Typewriter (RC 315)

General Rules.

The process *kind* is 8. Operations can be initiated by any internal process that is a *user* of the device. *Initialization* has no effect; *reservation* is forbidden. A typewriter accepts messages simultaneously from more than one process.

Setting the device in the *local* state has no effect other than delaying input/output until the device is set *remote* again.

Sense Operation.

The device is sensed and the status word delivered as an answer.

Input Operation.

If the *name* of the sender differs from the name of the last internal process that communicated with the typewriter, the name of the sender is output in the following format:

<new line> to <name of sender> <new line>

The operator can now input one line of characters to a storage area within the sending process. Characters are represented in the ISO 7-bit character code with three characters per word. Unused character positions in the last input word are filled with NULL characters.

Input characters with *parity error* are replaced by SUBSTITUTE characters, after which input is continued.

If the operator waits more than 2 seconds between the typing of two characters, the typewriter repeats the input operation of the missing character unless the interrupt key has been depressed or the sending process has been stopped. Associated with each typewriter is a constant defining the maximum number of *timer interrupts* allowed during input of one line (0 to 2047 times 2 seconds).

Input is terminated in the following situations: the area is full; after input of a NEW LINE, END MEDIUM, or CANCEL character; after depression of the interrupt key; after a maximum number of timer interrupts; when the sending process is stopped, whichever occurs first. In the latter case the last input character is lost.

In all cases input is terminated by an answer defining the actual number of characters input.

Output Operation.

If the *name* of the sender differs from the name of the last internal process that communicated with the typewriter, the name of the sender is output in the following format:

<new line> from <name of sender> <new line>

A textstring consisting of one or more lines is output from a storage area within the sending process. Characters must be represented in the ISO 7-bit character code with three characters per word.

After a *parity error* output continues.

Output is terminated in the following situations: the area is empty; after depression of the interrupt key; after a timer interrupt; when the sending process is stopped, whichever occurs first.
In all cases a multiple of three characters has been output, and output is terminated by an answer defining the actual number of characters output.

Status Bits.

0 intervention
2 timer

Messages and Answers.

operation:	message:	answer:
sense	0	status word
		0
		0
input	3<12	status word
	first storage address	number of bytes
	last storage address	number of characters
output	5<12	status word
	first storage address	number of bytes
	last storage address	number of characters

Operator Input Request.
The typewriter is not only able to receive messages from internal processes, it can also, on request from an operator, *send messages* to internal processes.
An operator message is transmitted in a typewriter storage area selected from a common pool within the monitor. Each typewriter area consists of an ordinary *message buffer* (which identifies the sender and the address of the input text) and a *line buffer* (in which the text is stored).
An operator request is made by depressing the *interrupt key* on the typewriter. The typewriter responds by outputting the following:

<new line> to

The operator can now type the name of the receiving process terminated by a NEW LINE character. If the operator only types a NEW LINE character, the receiving process is the last internal process with which the typewriter has communicated.
After this the operator can input one line of text. Characters are represented in the ISO 7-bit character code with three characters per word. Unused character positions in the last input word are filled with NULL characters.
Input characters with *parity error* are replaced by SUBSTITUTE characters, after which input is continued.

Input is terminated in the following situations: the area is full; after input of a NEW LINE, END MEDIUM, or CANCEL character; after depression of the interrupt key; after a maximum of timer interrupts; whichever occurs first.

The typewriter now examines whether the receiving process exists. If it does, the message is linked to its queue. Otherwise the typewriter releases the storage area and announces that the receiver is:

<new line>unknown<new line>

Associated with each typewriter is a *buffer claim* defining the maximum number of messages that can be sent simultaneously from the typewriter. The buffer claim is decreased by one when the typewriter sends a message, and increased by one when the receiving process sends an answer in the selected message buffer.

A typewriter can only send its last buffer to an internal process that has a one in the function mask bit called *can receive last console buffer* (see Section 11.3).

If all buffers are used or an attempt is made to send the last one to an internal process that cannot receive it, the following is output:

<new line> wait <new line>

The operator must then repeat the request later.

Operator Message.

An operator message, as received by an internal process, has the following format:

5<12
first storage address
last storage address
status word

If no characters are input and input is terminated by depressing the interrupt key or after a maximum number of timer interrupts, the last storage address equals first storage address – 2.

13.7. Paper Tape Reader (RC 2000)

General Rules.

The process *kind* is 10. Operations can only be initiated by an internal process that has *initialized* or *reserved* the device.

Sense Operation.

The device is sensed and the status word delivered as an answer.

Input Operation.

Characters are input to a storage area within the sending process. Three 8-bit characters are packed in each word. Unused character positions in the last input word are filled with NULL characters.

Input is terminated when the area is full, at the end of the paper tape, or when the sending process is stopped, whichever occurs first.

Input Mode.

Characters can be input after removal of an odd or even parity bit, or directly as 8 bits without parity checking.

Finally they can be input and converted from the *Flexowriter code* to the ISO 7-bit code as defined in Section 13.13. The *case* situation is set to lower case when the reader is initialized or reserved.

The input mode is part of the message:

mode:	0	odd parity
	2	even parity
	4	no parity
	6	Flexowriter to ISO conversion

In modes 0, 2, and 6 input is terminated after a *parity error*. The erroneous character is replaced by a SUBSTITUTE character. In mode 6 however ALL HOLES characters are skipped.

Status Bits.

1	parity
5	end of paper

It should be noted that the input block can have a length greater than zero when the status word indicates an *end of paper*.

Messages and Answers.

operation:	message:	answer:
sense	0	status word
		0
		0

input	3<12 + mode	status word
	first storage address	number of bytes
	last storage address	number of characters

13.8. Paper Tape Punch (RC 150)

General Rules.

The process *kind* is 12. Operations can only be initiated by an internal process that has *initialized* or *reserved* the device.

Setting the device in the *local* state has no effect other than delaying output until the device is set *remote* again.

Sense Operation.

The device is sensed and the status word delivered as an answer.

Output Operation.

Characters are output from a storage area within the sending process. Each storage word is output as three 8-bit characters.

Output is terminated when the area is empty, after a timer error, or when the sending process is stopped, whichever occurs first.

In all cases a multiple of three characters has been output, and output is terminated by an answer defining the actual number of characters output.

Output Mode.

Characters can be output after the addition of an odd or even parity bit, or directly as 8 bits without a parity bit.

Finally they can be converted from the ISO 7-bit code to the *Flexowriter code* and output as defined in Section 13.12. The *case* situation is set to lower case when the punch is initialized or reserved.

The output mode is part of the message:

mode:	0	odd parity
	2	even parity
	4	no parity
	6	ISO to Flexowriter conversion

Status Bits.

0	intervention
2	timer
5	paper tape supply low

Messages and Answers.

operation:	message:	answer:
sense	0	status word
		0
		0
output	5<12 + mode	status word
	first storage address	number of bytes
	last storage address	number of characters

13.9. Line Printer (RC 610)

General Rules.

The process *kind* is 14. Operations can be initiated by an internal process that has *initialized* or *reserved* the device.

Setting the device in the *local* state has no effect other than delaying output until the device is set *remote* again.

Sense Operation.

The device is sensed and the status word delivered as an answer.

Output Operation.

Characters are output from a storage word within the sending process. Characters must be represented in the ISO 7-bit code with three characters per word.

Output is terminated when the area is empty, when the sending process is stopped, after a parity error, after a timer error, or at end of paper, whichever occurs first. In the two first cases a multiple of three characters has been output.

Output is terminated by an answer defining the actual number of characters output. However at end of paper this number is decreased by three so no characters will be skipped when the rest of the buffer is output.

Status Bits.

0	intervention
1	parity
2	timer
5	end of paper

Messages and Answers.

operation:	message:	answer:
sense	0	status word
		0
		0
output	5<12	status word
	first storage address	number of bytes
	last storage address	number of characters

13.10. Punched Card Reader (RC 405)

General Rules.

The process *kind* is 16. Operations can be initiated by an internal process that has *initialized* or *reserved* the device.

When the device is *local* input operations will be terminated after a period of approximately one second with the intervention bit set. However if no cards are read the answer will be delivered after a number of periods defined by the fourth word of the message or when the device is set *remote* again, whichever occurs first. The device is set *local* when the local push-button is activated, after a timer error, when the input tray is empty, when the output tray is full, or when the reject tray is full. The device is set *remote* when the remote push-button is activated.

Sense Operation.

The device is sensed and the status word delivered as an answer.

Input Operation.

A number of punched cards of 80 or 51 characters each are input to a storage area within the sending process.

Input is terminated when there is no room for an entire card, when any status bit is set, or when the sending process is stopped, whichever occurs first.

In all cases input is terminated by an answer defining the actual number of characters transferred to the storage area.

Input Mode.

A card can be input in binary mode with two 12-bit characters per word. It can also be input in decimal or converted mode with three 8-bit characters per word. The input mode is part of the message. When it is zero, 80 column cards are read in binary mode without any conversion and all cards are returned to the output tray. The mode can be changed by adding the following values:

2	EBCD to ISO conversion
4	51 column cards
8	decimal
16	card to reject tray if reading error
32	card to reject tray if parity error

The EBCD to ISO conversion has a meaning only together with decimal mode. The conversion is defined in section 13.14.

Depending on the mode modulo 16 the following is inserted after each card:

0	nothing
4	a character with value = 0
8	a character with value = 0
10	a NL character (value = 10)
12	nothing
14	CR, NL, and CR characters (values = 13, 10, and 13)

In case of parity error or reading error the erroneous character is set to all ones in decimal mode and to a SUBSTITUTE character in conversion mode. The remaining characters are set to zero in decimal mode and in conversion mode. The above mentioned characters will, however, still be inserted after the card. The fourth word of the answer will contain the number of error-free characters. The number of characters transferred includes the entire card.

Status Bits.

0	intervention
1	parity error (conversion error)
2	timer (fail to feed after 500 msec)
3	data overrun
5	end of deck
6	output tray or reject tray full
10	reading error
11	card rejected

Messages and Answers.

operation:	message:	answer:
sense	0	status word
		0
		0
input	3 < 12 + mode	status word
	first storage address	number of bytes
	last storage address	number of characters
	local periods	number of error-free characters

13.11. Magnetic Tape Station (RC 747)

General Rules.

The process *kind* is 18. Operations can be initiated by an internal process that has *initialized* or *reserved* the device.

The device is sensed before each operation. If the status word indicates an intervention by the operator in the *local* mode, the name of the process is removed (together with the present reservation) and all messages are answered with the result: *receiver does not exist.* The device is now in the state: *document removed.*

When the operator switches back to *remote* mode, the device produces an interrupt. This causes the setting of the device state to: *unidentified document mounted.*

When the device is named by the procedure *create peripheral process,* the device state becomes: *identified document mounted.*

All answers from magnetic tapes contain a *file* and *block number* defining the position of the tape after the operation.

At load point both file and block numbers are zero.

The file number is increased by one after the output of a tape mark or the sensing of a tape mark during a forward operation. It is decreased by one when a tape mark is sensed after a backward operation.

The block number is increased by one when a block is input, output, or upspaced. If a tape mark is sensed during a forward operation, however, the block number is set to zero. It is decreased by one when a block is backspaced. If a tape mark is sensed during a backward operation, however, the block number is set to -1 indicating that the position is unknown.

These simple file and block numbers are based solely on a count of initiated operations and sensed status bits. There is no check against file and block numbers recorded in labels and data blocks.

Sense Operation.

The device is sensed and the status word delivered as an answer.

Input Block Operation.

A block of characters is input to a storage area within the sending process. Each storage word contains four 6-bit characters. Unused character positions in the last input word are filled with NULL characters.

Input is terminated when the storage area is full or the whole block has been transferred, whichever occurs first.

Output Block Operation.

A storage area within the sending process is output as one block. Each storage word is output as four 6-bit characters.

Erase Operation.

Erases a length of tape.

Move Operation.
Moves the tape in accordance with the move operation defined in the message (if the move operation is negative it is treated as a sense operation):

move operation:	0	upspace file
	1	upspace block
	2	backspace file
	3	backspace block
	4	rewind tape
	5	unwind tape
	>5	sense tape

Output Tape Mark.
Outputs a block consisting of four characters with the decimal value 15 with even parity.

Input/Output Mode.
Characters can be input and output with even or odd parity as defined by a mode in the message:

mode:	0	odd parity
	2	even parity

Status Bits.

1	parity error (after input and output)
2	timer (after input)
3	data overrun (after input and output)
4	block length error (after input)
5	EOT sensed (after input, output, and upspace)
6	BOT sensed (after rewind and backspace)
7	tape mark sensed (after input, backspace, and upspace)
8	write-enable sensed (permanent)
9	high density selected (permanent)

Messages and Answers.

operation:	message:	answer:
sense	0	status word
		0
		0
		file number
		block number

input block	3<12 + mode	status word
	first storage address	number of bytes
	last storage address	number of characters
		file number
		block number
output block	5<12+mode	status word
	first storage address	number of bytes
	last storage address	number of characters
		file number
		block number
erase	6<12	status word
		0
		0
		file number
		block number
move	8<12	status word
	move operation	0
		0
		file number
		block number
output mark	10<12	status word
		0
		0
		file number
		block number

13.12. ISO to Flexowriter Conversion

In the table below, each element corresponds to an ISO character with a decimal value equal to the sum of the column and row numbers. To the left in the element is shown the ISO character, while the corresponding Flexowriter character is shown to the right. The decimal value of Flexowriter characters is defined in Section 13.13. Flexowriter to ISO Conversion.

As an example, the ISO character & has the decimal value 32 +6 = 38, and is converted to the upper case Flexowriter character ^

	0	16	32	48	64	80	96	112
0	NUL	DLE	SP SP	0 0	@	P P	`	p p
1	SOH	DC1	! •	1 1	A A	Q Q	a a	q q
2	STX	DC2	"	2 2	B B	R R	b b	r r
3	ETX	DC3	£	3 3	C C	S S	c c	s s
4	EOT	DC4	$	4 4	D D	T T	d d	t t
5	ENQ	NAK	%	5 5	E E	U U	e e	u u
6	ACK	SYN	& ^	6 6	F F	V V	f f	v v
7	BEL	ETB	' 10	7 7	G G	W W	g g	w w
8	BS	CAN	((	8 8	H H	X X	h h	x x
9	HT TAB	EM END	))	9 9	I I	Y Y	i i	y y
10	NL CAR	SUB	* ×	: :	J J	Z Z	j j	z z
11	VT	ESC	+ +	; ;	K K	Æ Æ	k k	æ æ
12	FF STP	FS	, ,	< <	L L	Ø Ø	l l	ø ø
13	CR	GS	– –	= =	M M	Å Å	m m	å å
14	SO	RS	. .	> >	N N	^	n n	—
15	SI	US	/ /	?	O O	– •	o o	DEL

• An exclamation mark ! and an underline _ are converted to the special characters | and _ followed by a space.

Unassigned codes and codes > 127 are skipped during output.

13.13. Flexowriter to ISO Conversion

In the table below, each element corresponds to a Flexowriter character in lower or upper case with a decimal value equal to the sum of the column and row numbers. To the left in the element is shown the Flexowriter character, while the corresponding ISO character is shown to the right. The decimal value of ISO characters is defined in Section 13.12. ISO to Flexowriter Conversion.

As an example, the Flexowriter character Λ is an upper case character with the decimal value 16 + 0 = 16; it is converted to the ISO character &

	0				16				32				48			
	LC		UC		LC		UC		LC		UC		LC		UC	
0	SP	SP	SP	SP	0	0	^	&	–	–	+	+	æ	æ	Æ	Æ
1	1	1	v	!	<	<	>	>	j	j	J	J	a	a	A	A
2	2	2	×	*	s	s	S	S	k	k	K	K	b	b	B	B
3	3	3	/	/	t	t	T	T	l	l	L	L	c	c	C	C
4	4	4	=	=	u	u	U	U	m	m	M	M	d	d	D	D
5	5	5	;	;	v	v	V	V	n	n	N	N	e	e	E	E
6	6	6	[	Æ	w	w	W	W	o	o	O	O	f	f	F	F
7	7	7	]	Å	x	x	X	X	p	p	P	P	g	g	G	G
8	8	8	(	(	y	y	Y	Y	q	q	Q	Q	h	h	H	H
9	9	9	)	)	z	z	Z	Z	r	r	R	R	i	i	I	I
10													LC		LC	
11	STP	FF	STP	FF	,	,	10	'	o	o	O	O	.	.	:	:
12	END	EM	END	EM									UC		UC	
13	å	â	Å	Â												
14	•	–	•	!	TAB	HT	TAB	HT								
15																
16													NL	CAR	NL	CAR

• An underline_ or a bar | followed by a space are converted to the special characters_ and ! respectively. Multiple underlines or bars, case shifts, and unassigned codes are skipped between the first underline or bar and the space. An underline or bar followed by a graphic character are converted to the SUB character.

Unassigned codes and codes 65 - 127 are skipped during input.

13.14. EBCD to ISO Conversion

In the table below, each element corresponds to an EBCD character with a decimal value equal to the sum of the column and row numbers. In the element is shown the corresponding ISO character. The decimal value of ISO characters is defined in Section 13.12. ISO to Flexowriter Conversion.

As an example, the character + has the decimal value 64 + 14 = 78 and the hole pattern 12-8-6.

			0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
	holes			0	11	11 0	12	12 0	12 11	12 11 0	9	0 9	11 9	11 0 9	12 9	12 0 9	12 11 9	12 11 0 9
0			SP	0	–		&				9	z	r	z	i	i	r	
1		1	1	/	j		a	a	j				DC1		SOH			
2		2	2	s	k	s	b	b	k		SYN	FS	DC2		STX			
3		3	3	t	l	t	c	c	l						ETX			
4		4	4	u	m	u	d	d	m									
5		5	5	v	n	v	e	e	n		DC4		NL		HT			
6		6	6	w	o	w	f	f	o			ETB	BS					
7		7	7	x	p	x	g	g	p		EOT	ESC	NUL		DEL			
8	8		8	y	q	y	h	h	q				CAN					
9	8	1											EM			NUL	DLE	
10	8	2	:		!													
11	8	3	æ	,	å		.								VT			
12	8	4	ø	%	*		<				DC4				FF			
13	8	5	'	_	)		(				NAK	ENQ			CR			
14	8	6	=	>	;		+					ACK			SO			
15	8	7	”	?	¯						SUB	BEL			SI			

Unassigned codes are converted to the SUB character.

A NL character is inserted after each card during input.

PART III. CATALOG INITIALIZATION

Chapter 14

INITIALIZING FUNCTIONS

This chapter illustrates by means of examples the initialization of the backing store after system loading.

14.1. Introduction

After autoloading of the system tape the basic operating system is started. Initially the operating system executes a program that can initialize the backing store with catalog entries and binary Slang programs input from paper tape.

The initializing program starts by typing:

initialize catalog

on the main console. The operator can now insert a binary paper tape in the reader and type a NL character on the console.

The program reads one Slang segment at a time and interprets it as a sequence of initializing commands. Basically there are four kinds of initializing functions:

definition of backing store and catalog,
creation and modification of catalog entries,
loading of data areas, and
end of initialization.

14.2. Definition of Backing Store

The initializing commands are identified by textstrings followed by a fixed number of parameters. Normally the first Slang segment begins with the command<:newcat:>:

```
s.w.<:newcat:>
    ----
e.
```

This causes the program to copy a description of the backing store into the monitor and clear all entries in the catalog. Following this it creates the entry named <:catalog:> which describes the catalog itself.

The backing store consists of one or more drums and disks divided into segments of 256 words. The system makes these appear as a single backing store with segments numbered from 0 to N across all devices. This *logical backing store* is described by two tables: a device table and a segment table.

The *device table* describes the actual configuration of the backing store. Each entry in this table defines the device number of a drum or disk and the logical number of its first segment. This table is copied from the initializing program into the monitor.

The *segment table* contains one bit for each segment of the logical backing store defining whether the segment is available or reserved. This table is initializing to all segments available. To prevent future areas from extending over more than one drum or disk, a non-existing logical segment is reserved at the end of each device. Finally the segments occupied by the catalog itself are reserved.

The first Slang segment input from paper tape can also begin with the command <:oldcat:>:

```
s.w. <:oldcat:>
     ----
e.
```

In this case the program copies the device table into the monitor and re-establishes the segment table by scanning the existing catalog.

The first initializing command must always be either <:newcat:> or <:oldcat:>; otherwise the state of the multiprogramming system is undefined.

14.3. Loading of Backing Store

A set of commands permits the creation and modification of catalog entries. These commands have exactly the same effect as the monitor procedures:

```
create entry
change entry
rename entry
remove entry
permanent entry
```

A load command enables the loading of binary segments from paper tape to an area described in the catalog.

As an example consider a paper tape with a command segment followed by two program segments and terminated by another command segment. Let all segments be surrounded by a global block with a-names, which are used to transmit Slang parameters from the program to the catalog:

```
b. a9
s. w.    <:newcat:>
         <:create:>,<:editorprog:. >,10.0,r.9
         <:load:>,<:editorprog:>,2
e.
s.       (program segment 1)
e.
s.
         (program segment 2)
e.
s. w.    <:change:>,<:editorprog:>,a0,a1,---,a9
         <:perman:>,<:editorprog:>,1
         <:end:>
e.
e.
```

The <:newcat:> command creates a new catalog.

The <:create:> command creates a catalog entry named <:editorprog:>, which describes an area of 10 backing store segments; the nine optional words in the entry are set to zero.

The <:load:> command reads the following two program segments and moves them to the beginning of the area named <:editorprog:>.

The first command segment has now been processed; the program continues to interprete the next segment as commands:

The <:change:> command modifies the entry named <:editorprog:> in accordance with a-names defined during assembly of the program segments: the area size is reduced to a0 and the optional words become a1 to a9.

The <:perman:> command makes the entry named <:editorprog:> permanent with the catalog key 1.

Finally the <:end:> command terminates initialization of the catalog by typing:

ready

on the main console. The operating system is now ready to receive normal operator requests from consoles.

Chapter 15

DEFINITION OF INITIALIZING COMMANDS

In the following the syntax and functions of the initializing commands are described in detail.

15.1. Command Language

The program reads a sequence of binary Slang segments terminated by checksums from paper tape. The first segment is input and interpreted as a sequence of initializing commands. When a command segment has been processed, the program continues to read and interpret the next segment. After a load command a number of segments following the present command segment is input and moved to a backing store area. The following segments are then interpreted as commands until another load command occurs, and so on. At the end of a paper tape the program repeats input until another paper tape is inserted in the reader.

This cycle continues until an <:end:> ends initialization and starts the normal function of the basic operating system.

A *command segment* must not exceed 256 words; a *program segment* to be loaded can be of any length.

A *command* is identified by a textstring followed by a fixed number of *parameters*. Parameters can be catalog *entry names* and *Slang words*. An entry name is a textstring of 12 ISO characters beginning with a small letter followed by a maximum of 10 small letters or digits terminated by NULL characters.

15.2. Newcat Command

<:newcat:>

Copies the device table into the monitor, initializes the segment table, clears all catalog entries, and creates the entry describing the catalog itself.

15.3. Oldcat Command

<:oldcat:>

Copies the device table into the monitor and re-establishes the segment table by scanning the existing catalog.

15.4. Create Command

<:create:>,<name>,<number of segments>,<9 optional words>

Creates a temporary catalog entry with the name and contents as specified. If the number of segments is greater than zero, an area of that size is reserved on the backing store and described in the entry.

15.5. Change Command

<:change:>,<name>,<number of segments>,<9 optional words>

Changes an existing catalog entry with a given name as specified. If the entry describes an area on the backing store, the number of segments is reduced to the value specified.

15.6. Rename Command

<:rename:>,<name>,<new name>

Renames an existing entry with a given name as specified.

15.7. Remove Command

<:remove:>,<name>

Removes an existing entry with a given name along with its associated area on the backing store.

15.8. Perman Command

<:perman:>,<name>,<catalog key>

Makes an existing catalog entry with a given name permanent with the catalog key as specified.

15.9. Load Command

<:load:>,<name>,<number of slang segments>

Loads a number of Slang segments following the present command segment to an existing area with a given name. The Slang segments are loaded word by word from the beginning of the area, ignoring the boundaries of backing store segments. Checksums are controlled during input, but not transferred to the backing store.

15.10. Console Messages

When the initializing program is started, it prints the message:

initialize catalog

The operator should now insert the first binary tape in the reader and type a NL character.

After reading the <:end:> command, the message:

ready

is output, and the operating system is now ready to receive normal operator requests from consoles.

After detection of an error the program types an error message and returns to the initial situation, in which the operator can repeat the entire initialization of the catalog. The following is a list of error messages printed by the program. The meaning of the result values and status bits are defined in Chapters 12 and 13 (Definition of Monitor Procedures and External Processes).

error message:	meaning:
<command name> syntaxerror	undefined command
reader result <result>	dummy answer after reader input
reader status <bit no>	status bit after reader input
reader sumerror	sumerror in Slang segment
reader sizeerror	command segment > 256 words
catalog result <result>	dummy answer after catalog input/output
catalog status <bit no>	status bit after catalog input/output
catalog error	old catalog inconsistent
create <name> result <result>	create entry, result <> 0
change <name> result <result>	change entry, result <> 0
rename <name> result <result>	rename entry, result <> 0
remove <name> result < result>	remove entry, result <> 0
perman <name> result <result>	permanent entry, result <> 0
load <name> result <result>	create area process, result <> 0
<name> result <result>	dummy answer after area output
<name> status <bit no>	status bit after area output

PART IV. BASIC OPERATING SYSTEM

Chapter 16

OPERATING SYSTEM FUNCTIONS

This chapter illustrates by means of examples the functions of the basic operating system, which can initiate and control the execution of parallel programs on request from typewriter consoles.

16.1. Introduction

After initial system loading, the internal store contains the monitor and the basic operating system s. S enables independent operators to initiate and control internal processes from typewriter consoles. In addition to this, s can name peripheral devices and keep track of the date and time.

S is the "pater familias" of the family tree of internal processes. Initially it owns all system resources such as storage, protection keys, peripherals, message buffers, and process description tables. Apart from being a permanent process in the system, s has no special status, but is treated by the monitor as any other internal process. In the present implementation s shares the protection key zero with the monitor. This is only done, however, to save a protection key for the user processes. Although the protection key enables it to do so, s does not execute privileged instructions or modify process descriptions within the monitor. Thus it is possible on the system tape to replace s with another basic operating system.

16.2. Control of Internal Processes

In the following the creation and control of internal processes from consoles is explained. An operator sends a message to operating system s by depressing the interrupt key on a console and typing the name s followed by a command line. A message, such as the following:

```
to s
new pbh run
```

causes s to create an internal process with the name pbh, load a program into it from the backing store, and start its execution. When the process has been created, s outputs the message:

```
ready
```

In this case the process was created with a standard set of resources, which enables it to execute system programs such as the Editor, Slang assembler, or Algol compiler. The program loaded into the process was the File processor, which can input and interpret further job control statements.

The operator can also explicitly specify the resources he wants; for example, the size of the storage area, the number of message buffers, and the program to be loaded:

```
to s
new pbh size 16000 buf 18 prog opsys2 run
```

Resources not mentioned (e.g. the number of area processes) are still given standard values.

Normally s chooses the actual location of storage areas and the values of protection keys. The operator can, however, specify these absolutely:

```
to s
new pbh addr=13500, pr=2/3/4, pk=2 run
```

but s will only accept this if the storage area is available.

After creation and start the user process can communicate with the console according to its own rules:

```
from pbh
------
```

When the operator wants to stop program execution temporarily within his process, he types:

```
to s
stop
```

He can start it again at any time by the command:

```
to s
start
```

If the user process sends a message to operating system s, the process is stopped by s, and the following message is output:

```
from s
pause pbh
```

At this point the operator has the choice of starting the process again or removing it completely from the system:

```
to s
remove
```

During execution a breakpoint can be made from the console by typing:

```
to s
break
```

This causes the operating system to interrupt the user process and continue execution within the internal interrupt procedure of the user process.

It is possible to create and control more than one process from the same console, for example:

```
to s
new ls run new pbh run
```

But in this case the operator must preface subsequent commands with the name of the process he is referring to:

```
to s
proc ls stop
```

Actually the operating system remembers the name of the last process referred to from a console. It is therefore only necessary to mention the process name explicitly each time the operator refers to a new process.

16.3. Control of External Processes

After its creation an internal process is included as a user of a standard configuration defined within s; but the operator can also explicitly include or exclude his process as a user of other devices as well:

```
to s
include 7, 9, 13 exclude 5, 4
```

After mounting documents the operator can assign names to peripherals, for example:

```
to s
call 5/printer, 8/magtape3
```

This must always be done after mounting magnetic tapes because the monitor removes their names when the stations are set to local.

16.4. Date and Time

The monitor updates an integer clock of 48 bits with units of 0.1 milliseconds. The operator can initialize this clock from a console. The clock is input as day, month, year followed by hour, minute, second:

```
to s
newdate 31.3.69 14,55,42
```

Operating system s converts this to a 48-bit clock value, which defines the time interval elapsed since midnight 31 December 1967.

The operator can also ask s to output the clock with the same format by typing a date command:

```
to s
date
```

16.5. System Status

The operating system prints an error message when it is unable to honor a request, for example:

```
to s
new pbh size 20000 run
no core
```

In this situation the operator can ask s to list the maximum number of each resource available at present:

```
to s
max
max 18000 18 14 2 6
```

In this example the largest available storage area contains 18000 bytes, whereas the number of message buffers, area process descriptions, internal process descriptions, and protection keys available are 18, 14, 2, and 6. Note that this is a snapshot of available resources. It can happen that some of them will be reserved from other consoles before the operator in question makes another request.

Finally the operating system can output a list of all internal processes created by it. These are listed in the order their storage areas follow each other from low toward high

addresses. Each process is described by name, first storage address, size of storage area, and the protection key set within the area:

```
to s
list
ls 12216 10000 3
pbh 22216 6000 1
jz 40000 15000 2
```

16.6. Communication Strategy

Commands from a console are served in the order of their arrival. The processing of a command line is terminated by a short reply printed on the console. The normal, but not required, mode of operation should therefore be to wait for this response before sending the next message to s from this console.

The monitor permits simultaneous input of messages from all consoles. The operating system, however, can only respond simultaneously to a limited number of messages. For each simultaneous conversation s uses a working area to process a command line. When s must wait for console output, the current value of registers and the address of the message buffer involved are stored in the working area before s inspects its event queue for other messages or answers. An answer to s causes retrieval of the corresponding working area and continuation of the interrupted action.

A message to s is only processed when a working area is available and all previous messages from the same console have been served.

The maximum number of simultaneous conversations within s is an assembly option.

Chapter 17

DEFINITION OF CONSOLE COMMANDS

In the following the syntax and functions of the console command language are described in detail.

17.1. Console Parameters

The basic purpose of the command language is to enable an operator to describe the resource requirements of an internal process and to create and control the process from a console. The operating system attempts to minimize the amount of operator input in two ways: by supplying standard values of parameters not defined explicitly; by storing the latest value of all parameters defined from a console.

Associated with each console is a description within s of an internal process consisting of the following parameters:

textstrings:	process name
	program name
integers:	first storage address
	size of storage area
	buffer claim
	area claim
	internal claim
	function mask
	catalog mask
	protection register
	protection key
	number of keys
booleans:	absolute address
	absolute protection

The process name, storage address, size, buffer claim, area claim, internal claim, function mask, catalog mask, protection register, and protection key are the parameters required by the monitor when an internal process is created. The booleans specify whether the storage area and protection keys should be allocated absolutely as stated by the operator; if not, the operating system can assign available values to them according to its own strategy. Finally the program name is the name of an entry in the catalog on the backing store, which describes the program to be loaded after the creation of an internal process.

We emphasize that the console parameters do not necessarily describe an internal process. The command language distinguishes between commands that merely assign values to the console parameters and commands that check parameter values during the creation and control of an actual process.

17.2. Console Classification

Also associated with each console is a fixed bit pattern, the *command mask*, which defines the set of commands accepted from the console. This assembly option makes it possible to give various degrees of freedom to different consoles.

At present the ability to issue commands is defined by six bits within the command mask:

bit a:	privileged console
bit b:	new, proc, prog, create, load, start, init, run, stop, break, and remove allowed
bit c:	size, addr, buf, area, internal and key allowed
bit d:	function, catalog, pr, and pk allowed
bit e:	call allowed
bit f:	list, max and date allowed

Normally an internal process can only be controlled and removed from the console where it was created. *Privileged consoles* are, however, able to control and remove any internal process created by operating system s. Furthermore privileged consoles are the only ones able to initiate date and time.

17.3. Command Syntax

A message to the operating system is one *line*, which can be empty or contain a number of commands. A line is either terminated by a new line character or by a semicolon followed by a *comment* consisting of any string of characters terminated by a new line character.

A *command* is identified by a name followed by a number of parameters. *Parameters* can be names or numbers.

A *name* is a textstring of small letters and digits beginning with a small letter. Its maximum length is 11 characters.

A *number* is a string of digits interpreted as an unsigned integer modulo 2**12 or 2**24 depending on context.

Commands and parameters are *separated* by one or more spaces, or by one of the special characters , . / = followed by zero or more spaces.

In the following definitions of commands, separators are not shown explicitly. It is understood that a definition like the following:

call <device number> <device name>

strictly speaking means:

call<separator><device number><separator><device name>

In some cases a command can be followed by a variable number of parameters, for example:

include <device number> ...

This notation means that parameters between the command name and the three dots can be omitted or typed one or more times as illustrated below:

```
include
include 5
include 3, 7, 1
```

The first of these commands has no effect.

17.4. New Command

new <process name>

Assigns a name to the console parameter: *process name*, and initializes the rest of the console parameters with standard values as required by the system programs (Editor, Slang, Algol, etc.). The booleans *absolute address* and *absolute protection* are set to false. The *program name* is initialized with the name of the File Processor on the backing store.

Example: new ts

17.5. Proc Command

proc <process name>

Assigns a name to the console parameter: *process name.*

Example: proc sl

17.6. Prog Command

prog <program name>

Assigns a name to the console parameter: *program name.*

Example: prog batchsystem

17.7. Size Command

size <size of storage area>

Assigns a 24-bit number to the console parameter: *size of storage area.* An odd number is decreased by one.

Example: size 12000

17.8. Addr Command

addr <first storage address>

Assigns a 24-bit number to the console parameter: *first storage address,* and sets the boolean *absolute address* to true. An odd number is decreased by one.

Example: addr 16320

17.9. Buf Command

buf <buffer claim>

Assigns a 12-bit number to the console parameter: *buffer claim.*

Example: buf 7

17.10. Area Command

area <area claim>

Assigns a 12-bit number to the console parameter: *area claim*.
Example: area 0

17.11. Internal Command

internal <internal claim>

Assigns a 12-bit number to the console parameter: *internal claim.*
Example: internal 2

17.12. Function Command

function <bit number> . . .

Defines a sequence of bit numbers that are set to one in the 12-bit console parameter: *function mask.* The rest of the bits are set to zero.
Example: function= 4, 5, 6
results in the function mask 2.000011 100000

17.13. Catalog Command

catalog <bit number> . . .

Defines a sequence of bit numbers that are set to one in the 24-bit console parameter: *catalog mask.* Bit 23 (corresponding to the catalog key of the catalog itself) is always set to zero. The rest of the bits are also set to zero.
Example: catalog 1/3/5/8/12/22/23
results in the catalog mask 2.010101 001000 100000 000010

17.14. Key Command

key< number of keys>

Assigns a 12-bit number to the console parameter: *number of keys,* and sets the boolean *absolute protection* to false.
Example: key 2

17.15. Pr Command

pr <bit number> . . .

Defines a sequence of bit numbers that are set to zero in the 12-bit console parameter: *protection register.* Bit 0 (corresponding to the protection key used within the monitor) is always set to one. The rest of the bits are also set to one.
Example: pr= 2, 5, 3
results in the protection register 2.1100 1011

17.16. Pk Command

pk<protection key>

Assigns a 12-bit number to the console parameter: *protection key,* and sets the boolean *absolute protection* to true.
Example: pk=5

17.17. Create Command

create

Creates an internal process as described by the console parameters.

If *absolute addressing* is specified, the availability of the storage area is checked. If only the *size* is specified, the operating system scans the store from low to high addresses and chooses the first available area of sufficient size.

If *absolute protection* is specified, the protection register and the protection key are accepted as stated. (The user is warned that the entire system can collapse, if an erroneous program is executed in monitor mode, i.e. with a protection register in which the bit corresponding to the protection key used within the process is one). If only the *number of keys* is specified, the operating system scans all internal processes and creates a protection register with zeroes for all keys that have not been assigned to any process. From this possible register an actual register with zeroes corresponding to the specified number of keys is selected from left to right, and the protection key is set equal to the right-most assigned key.

The ability of the operating system to supply the number of *message buffers, area process* descriptions, and *internal process* descriptions specified is then checked.

During the actual creation of the process the monitor checks that its *name* is unique.

After creation the operating system includes the process as a user of a standard configuration of peripherals.

Finally the *registers* of the internal process are initialized as follows:

w0 = 0
w1 = process description of s
w2 = process description of console
w3 = process description of child
EX = 0
IC = first storage address

17.18. Load Command

load

Loads a program defined by the console parameter: *program name*, from the backing store into the beginning of an internal process defined by the console parameter: *process name*.

The program name must refer to a *catalog entry* that describes an area on the backing store, and the entry must not exist as an area process reserved by another internal process.

The optional words in the entry must describe the contents of the area as a directly executable program in accordance with the specifications given in the File Processor manual.

The number of bytes to load and the relative entry point of the program must not exceed the size of the internal process.

The console must be either the console from which the internal process was created or a privileged console.

After loading, the *registers* of the process are initialized as follows:

w0 = 0
w1 = process description of s
w2 = process description of parent console
w3 = process description of child
EX = 0
IC = absolute address of entry point

17.19. Start Command

start

Starts the execution of an internal process defined by the console parameter: *process name.*

The console must be either the console from which the process was created or a privileged console.

17.20. Init Command

init

The initialize command has the same effect as a *create* command followed by a *load* command (Sections 12.17 and 12.18)

17.21. Run Command

run

The run command has the same effect as a *create* command followed by a *load* command and a *start* command (Sections 12.17, 12.18, and 12.19).

17.22. Stop Command

stop

Stops the execution of an internal process defined by the console parameter: *process name.*

The console must be either the console from which the process was created or a privileged console.

17.23. Break Command

break

Stops the execution of an internal process, defined by the console parameter: *process name,* stores its registers and an interrupt cause (= 8) at the head of its *internal interrupt* procedure, and continues the execution of the process within the interrupt procedure.

The console must be either the console from which the process was created or a privileged console.

After the breakpoint the *registers* of the process have the following contents:

w0 = 0
w1 = process description of s

w2 = process description of parent console
w3 = process description of child
EX = 0
IC = internal interrupt address + 14

The break command has no effect if the internal interrupt address of the process is zero.

17.24. Remove Command

remove

Stops and removes an internal process defined by the console parameter: *process name.*

The console must be either the console from which the process was created or a privileged console.

The console parameters are unchanged.

17.25. Include Command

include <device number> . . .

Includes an internal process, defined by the console parameter: *process name,* as a user of a sequence of peripheral devices.

The console must be either the console from which the process was created or a privileged console.

Example: include 3, 6, 7

17.26. Exclude Command

exclude <device number> . . .

Excludes an internal process, defined by the console parameter: *process name,* as a user of a sequence of peripheral devices.

The console must either be the console from which the process was created or a privileged console.

Example: exclude 1

17.27. Call Command

call <device number> <device name> . . .

Assigns names to a sequence of peripheral devices. The devices must not be reserved by internal processes.

Example: call 0/reader 5/magtape1 7/printer

17.28. Newdate Command

newdate <day> <month> <year> <hour> <minute> <second>

Initializes the clock with the number of 0.1 milliseconds elapsed since midnight 31 December 1967. The conversion algorithm is valid in the interval: 68 <= year < = 99

The console must be privileged.

Example: newdate 29.2.72 13,30,0

17.29. Date Command

date

Prints the current value of the clock with the format:

date <day>.<month>.<year> <hour>,<minute>,<second>

17.30. Max Command

max

Prints the maximum number of available resources with the format:

max <max storage area> <buf claim> <area claim> <internal claim> <keys>

17.31. List Command

list

Prints a list of all internal processes created by operating system s in the same order their storage areas follow each other from low toward high addresses. Each process is described on one line with the format:

<process name> <first storage address> <size of storage area> <key>

17.32. Console Messages

When the operating system has processed a line of commands, it prints the message:

ready

After detection of an error the rest of the command line is ignored, and the console returns to the ready situation. The following is a list of error messages printed by the operating system:

error message:	meaning:
syntax error	illegal command syntax
not allowed	command forbidden from this console
not implemented	optional command not assembled
no core	storage area not available
no buffers	message buffers not available
no areas	area process descriptions not available
no internals	internal process descriptions not available
key trouble	incorrect use of protection keys
process exists	process name not unique
area unknown	area not described correctly in catalog
area reserved	area reserved by another process
program too big	program size or entry point exceeds storage area
catalog error	input/output error during catalog look up
area error	input/output error during area loading
process unknown	process does not exist
device unknown	device number does not exist
device reserved	device reserved by another process

17.33. Child Messages

When the operating system receives a message from one of its child processes, the following is done: the child process is stopped, an answer is sent to the process, and a message is printed on the console from which the process was created:

pause <process name>

The operator can now start, break, or remove the process.

APPENDIX

IMPLEMENTATION DETAILS

The appendix defines the format of queues, buffers, process descriptions, and the catalog on the backing store. It also defines how these entities can be addressed within internal processes.

A.1. Administration of Queues

The monitor administers the following queues:

a time slice queue
a message buffer pool
a console buffer pool
an event queue for each process

A *queue* consists of a *head* and a number of *elements*, which are linked cyclically. A head or element consists of two consecutive words defining the addresses of the next and last elements in the queue, for example:

head q:	elem a:	elem b:	elem c:
a	b	c	q
c	q	a	b

When the queue is *empty*, the head points to itself:

head q:
q
q

Also when an element is *removed* from the queue without being linked to another queue, the element points to itself:

elem b:
b
b

A.2. Administration of Time Slices

The queue of running internal processes is described in the following storage locations within the monitor:

absolute address:	contents:
66	<current internal process>
68	<next running process>
70	<last running process>
102	<time slice>

Current internal process is the process description address of the process executing instructions right now.

Next and *last running process* are the head of the time slice queue that links the process descriptions of all running processes cyclically. Note that these links are not process description addresses, but process description addresses + 18 (see Section A.6. Format of Internal Process Description).

Time slice is a constant defining the maximum time quantum, in units of 0.1 millisecond, allotted to each process in turn.

A.3. Administration of Message Buffers

A.3.1. Message Buffer Pool

The message buffer pool is described in the following storage locations within the monitor:

absolute address:	contents:
82	<next message buffer>
84	<last message buffer>
86	<message pool start>
88	<message pool end>
90	<message buffer size>

Next and *last message buffer* are the head of the queue of available message buffers that are linked cyclically.

Message pool start and *end* are the first and last addresses of the contiguous storage area containing all message buffers.

Message buffer size is the number of bytes per message buffer (at present = 24 bytes).

A.3.2. Message Buffer Format

A message buffer has the following format:

buffer address:	contents:
+0	<next buffer>
+2	<last buffer>
+4	<receiver>
+6	<sender>
+8	<message or answer>
----	----
+22	<message or answer>

Next and *last buffer* are the link that connects the buffer cyclically to other buffers in its present queue.

Receiver and *sender* define the state of the buffer as described in Section A.3.3.

Message or *answer* are eight words exchanged between two communicating processes.

A.3.3. Message Buffer States

The possible states of a message buffer are defined by the values of the sender and receiver parameters:

sender:	receiver:	state:
0	0	buffer available
sender addr	receiver addr	message from existing sender pending
sender addr	–receiver addr	message from existing sender received
–parent addr	receiver addr	message from removed sender pending
–parent addr	–receiver addr	message from removed sender received
sender addr	1	normal answer pending
sender addr	2	dummy answer, message rejected
sender addr	3	dummy answer, message unintelligible

sender addr	4	dummy answer, receiver malfunction
sender addr	5	dummy answer, receiver unknown

The *sender* and *receiver addresses* are the process description addresses of the two processes that communicate in the buffer.
The *parent address* is the process description address of the process whose buffer claim should be increased by one when the receiver sends an answer after the removal of the sender.
In pending answers the receiver address has been replaced by the *result* parameter of the procedure wait answer.
The transitions between these states are clarified by the following scheme:

after calling:	buffer state:	queue situation:
send message	message pending	buffer in receiver queue
wait message	message received	buffer in no queue
send answer	answer pending	buffer in sender queue
wait answer	buffer available	buffer in pool

A.4. Administration of Console Buffers

A.4.1. Console Buffer Pool

The console buffer pool is described in the following storage locations within the monitor:

absolute address:	contents:
92	<next console buffer>
94	<last console buffer>
96	<console pool start>
98	<console pool end>
100	<console buffer size>

The meaning of these parameters is the same as for message buffers (see Section A.3.1). At present the size of a console buffer is 66 bytes.

A.4.2. Console Buffer Format

A console buffer has the following format:

buffer address:	contents:
+ 0	<next buffer>
+ 2	<last buffer>
+ 4	<receiver>
+ 6	<sender>
+ 8	<message>
----	----
+ 16	<input line>
----	----
+ 64	<input line>

The first part has the same format as a *message buffer* except that the message itself occupies four instead of eight words. The meaning of these parameters is defined in Section A.3.2.

The second part is a *line buffer* that contains the textstring input by the operator. At present it has room for 50 bytes = 75 characters.

A.4.3. Console Buffer States

The possible states of a console buffer are the same as for a message buffer except that the sender (i.e. the console) cannot be removed (see Section A.3.3).

A.5. Administration of Process Descriptions

The format of a process description depends on the kind of the process; common to all process descriptions, however, is a head defining the *kind* and *name* of the process:

process description address:	contents:
+0	<kind>
+2	<name>
+10	<other parameters>

A process is considered *removed* if the first word of its name is zero.

The monitor has a *name table* containing the base addresses of all process descriptions (including those of removed processes). The name table is described in the following storage locations:

absolute address:	contents:
72	<name table start>
74	<first device in name table>
76	<first area in name table>
78	<first internal in name table>
80	<name table end>

Name table start is the address of the first name table entry. This and the following name table entries contain process description addresses of pseudo-processes, used during debugging of the system, and external processes corresponding to pure interrupt sources.

First device in name table is the address of the name table entry that contains the process description address of peripheral device number 0. The following name table entries contain process description addresses of peripheral devices 1,2,3, etc.

First area in name table is the address of the name table entry that contains the process description address of the first area process. The following name table entries contain the rest of the area process description addresses.

First internal in name table is the address of the name table entry that contains the process description address of the first internal process. The following name table entries contain the rest of the internal process description addresses.

Name table end is the address of the last name table entry. This entry does not contain a process description address.

A.6. Format of Internal Process Description

An internal process is described in a table with the following format:

relative address:	contents:
0	<kind>
2	<name>
10-11	<stop count><state>
12	<identification bit>
14	<next event>
16	<last event>
18	<next process>
20	<last process>
22	<first storage address>
24	<top storage address>
26-27	<buffer claim><area claim>
28-29	<internal claim><function mask>
30	<catalog mask>
32-33	<protection register><protection key>
34	<interrupt mask>
36	<internal interrupt address >
38	<working register 0>
40	<working register 1>
42	<working register 2>
44	<working register 3>
46	<exception register>
48	<instruction counter>
50	<parent description address>
52	<time quantum>
54	<run time>
58	<start time>
62	<waiting time>
66	<wait address>
68	<creation number>

Kind is 0 for an internal process.

Name is a textstring of 12 ISO characters beginning with a small letter followed by a maximum of 10 small letters or digits terminated by NULL characters.

Stop count defines the number of internal and external processes modifying the storage area of the present process.

State has one of the following values:

2.0100 1000	running
2.0000 1000	running after error
2.1011 0000	waiting for stop by parent
2.1010 0000	waiting for stop by ancestor
2.1011 1000	waiting for start by parent

2.1010 1000	waiting for start by ancestor
2.1100 1100	waiting for process function
2.1000 1101	waiting for message
2.1000 1110	waiting for answer
2.1000 1111	waiting for event

The meaning of these states is defined in Sections 6.3, 8.1, and 9.1.

Identification bit is a word in which a single bit is one. It is used to check the access to external processes (see Section A.7).

Next and *last event* are the head of the event queue that links all pending messages and answers cyclically to the process.

Next and *last process* are the element that links the present process to other internal processes in the time slice queue.

First and *top address* point to the first storage word and the last storage word + 2 of the process.

Buffer claim, area claim, and *internal claim* define the number of message buffers, area process descriptions,and internal process descriptions that are still available to the process.

Function mask defines the privileged monitor functions that can be called by the process. The meaning of the mask bits is given in Section 11.3.

Catalog mask is a bit pattern in which a one in bit number N indicates that the process can change or remove catalog entries with the catalog key N.

Protection register is a bit pattern in which a zero in bit number N indicates that the process can change or execute storage words with the protection key N.

Protection key is the key that is set in all storage words of the process when it is started.

Interrupt mask is the value of the interrupt mask register used during execution of the process.

Interrupt address is the address of the internal interrupt procedure.

Working registers, exception register, and *instruction counter* contain a dump of register values when the process does not execute instructions.

Parent description address is the process description address of the process that has created the present process.

Time quantum is the amount of execution time used by the process during the last time slice.

Run time is the total execution time used by the process since its creation.

Start time is the value of the clock when the process was created.

Waiting time is the value of the clock when the process started to wait for something.

Wait address is the address of the buffer in which the process awaits an answer.

Creation number is used to identify temporary catalog entries created by the process.

A.7. Format of Peripheral Process Description

A peripheral process is described in a table with the following format:

relative address:	contents:
0	<kind>
2	<name>
10	<device number*64>
12	<reserver>
14	<users>
16	<next message>
18	<last message>
20	<interrupt address>
22	<other parameters>

Kind is greater than zero.

Name is a textstring of 12 ISO characters beginning with a small letter followed by a maximum of 10 small letters or digits terminated by NULL characters.

Device number is the data channel identification of the peripheral device.

Reserver contains the identification bit of the internal process having reserved the peripheral process.

Users contains the identification bits of all internal processes that are potential users of the peripheral process.

Next and *last message* are the head of the queue that links all messages to the process cyclically.

Interrupt address is the address of the monitor code to be executed when the device delivers an interrupt.

Other parameters are an optional number of working locations that are specific for each kind of peripheral process.

A.8. Format of Area Process Description

An area process is described in a table with the following format:

relative address:	contents:
0	<kind>
2	<name>
10-11	<device number*2><catalog key>
12	<reserver>
14	<users>
16	<first absolute segment number>
18	<number of segments>

Kind is 4.

Name is a textstring of 12 ISO characters beginning with a small letter followed by a maximum of 10 small letters or digits terminated by NULL characters.

Device Number is the data channel identification of the peripheral device on which the area is stored.

Catalog Key is the number of a bit within the catalog mask of internal processes that defines whether these can change or remove the area on the backing store.

Reserver contains the identification bit of the internal process having reserved the area process.

Users contains the identification bits of all internal processes that are potential users of the area process.

First segment number is the absolute number of the first area segment on the given backing store device.

Number of segments defines the size of the area.

The area process description is only used to check the validity of messages to the area. If a message is accepted, the device number is used to find a peripheral process description of the drum or disk on which the area is stored, and the message is linked to the queue of this device:

```
entry:= first device in name table + device number*2;
backing store description:= name table (entry);
```

A.9. Administration of Backing Store

A.9.1. Device and Segment Tables

The composition of the backing store is described by a *device table.* Each device table entry defines the device number of a drum or disk and the logical number of its first segment. Segments are numbered consecutively from 0 to N across all backing store devices. The device table is terminated by a description of a dummy device zero:

first device:	<device no*8192>
	<first logical segment>
----	----
last device:	<device no*8192>
	<first logical segment>
dummy device:	0
	<last logical segment>

A *segment table* with one bit for each logical segment is used to distinguish between available (bit = 1) and reserved (bit = 0) segments. To prevent data areas from extending across more than one drum or disk, a non-existing logical segment is reserved at the end of each device.

The device and segment tables are kept in the top of the internal store, and describes as follows:

	<segment table>
	<device table>
– 20	<number of catalog segments>
– 18	<last generated name>
– 10	<catalog device number>
– 8	<device table address>
– 6	<segment table address>
– 4	<number of available catalog entries>
– 2	<number of available area segments>

top address = storage capacity

Number of catalog segments defines the size of the catalog.
Catalog device number is the number of the device on which the catalog is stored.

A.9.2. Format of Catalog

A *catalog entry* has the following format:

relative address:	contents:
0-1	<name key> <catalog key>
2	<creation number>
4	<first logical segment>
6	<name>
14	<number of segments>
16	<optional word 1>
----	----
32	<optional word 9>

Name key is a hashed value of the entry name formed as follows:

```
double word sum:= (name(0:47) + name(48:95)) modulo 2**48;
word sum:= (double word sum(0:23) + double word sum(24:47))
        modulo 2**24;
byte sum:= word sum(0:11) + word sum(12:23);
name key:= byte sum modulo number of catalog segments;
```

During creation of an entry the monitor scans the catalog cyclically, starting with the relative segment number given by name key, until it finds an available entry (the first word of an available entry is – 1). The same strategy is used to look up an entry.

Catalog key is the number of a bit within the catalog mask of internal processes that defines whether these can change or remove the entry.

Creation number identifies the internal process having created the entry; it is zero in a permanent entry.

First segment is the logical number of the first segment of an area described by the entry. During creation of an area process the logical segment number is transformed to a device number and an absolute segment number within the device by means of the device table. The value of the first segment is undefined if the entry does not describe an area.

Name is a textstring of 12 ISO characters beginning with a small letter followed by a maximum of 10 small letters or digits terminated by NULL characters.

Number of segments defines the size of an area described by the entry. If it is less than or equal to zero, the entry does not describe an area.

Optional words have no pre-assigned meaning within the monitor, but are defined by the internal process during creation or modification of the entry.

The catalog itself is an area on the backing store. Each *catalog segment* contains 15 entries of 17 words each plus one word that defines the number of entries created with the name key equal to the relative segment number. This is used to limit the search for a non-existing entry.

The catalog describes itself in an entry named <:catalog:> with the catalog key 23. It is also described permanently as an *area process* of the same name within the monitor.

A.10. Selected Execution Times

The following is a list of typical execution times of monitor procedures and external processes. It is based on a manual count of instructions and should be taken with some caution as explained in Section 9.4.

Disabled Monitor Procedures:	msec:
set interrupt	0.2
process description	0.6
initialize, reserve, and release process	0.6
include and exclude user	1.0
send message, send answer	0.6
wait message, wait answer	0.4
wait event, get event	0.3
get clock, set clock	0.2

Enabled Monitor Procedures:	msec:
create, change, remove, and permanent entry	8 + 2 segment transfers
look up entry, create area process	5 + 1 segment transfer
rename entry	10 + 5 segment transfers
create peripheral process	3
create internal process	3
start internal process	2 + 1.2 per core segment
stop internal process	4
modify internal process	2
remove process (external process)	2
remove process (internal process)	6 + 1.2 per core segment

The following execution times are required to process interrupts from external processes (the figures in parentheses apply if the interrupt terminates an operation with an answer):

clock (after 25.6 msec)	0.5	(1.0)
typewriter (after 1 character)	0.2	(0.7)
paper tape reader (after 64 ISO characters)	3.2	(3.7)
paper tape reader (after 64 Flexowriter characters)	4.5	(5.0)
paper tape punch (after 1 ISO character)	0.2	(0.5)
paper tape punch (after 1 Flexowriter character)	0.25	(0.5)
line printer (after 70 characters)	3.0	(0.5)
backing store, magnetic tape		(0.6)

VOCABULARY OF MONITOR CONCEPTS

ACTIVATE PROCESS (TO) To link an *internal process* to the *time slice queue* in order to make it *running*.

ANCESTOR PROCESS An *internal process* described as the grandparent, or great- grandparent, etc. of another internal process in the *process hierarchy*.

AREA PROCESS Input/output of a data area on the backing store identified by name.

CATALOG A fixed part of the backing store divided into named entries. An entry can describe a data area on the backing store or anything else.

CHILD PROCESS An *internal process* created by another internal process. The latter is described as the *parent* in the *process hierarchy*.

CREATE PROCESS (TO) To create a table within the monitor describing a *process* by its kind, name, *resources, event queue,* and current state.

DELAY PROCESS (TO) To remove an *internal process* temporarily from the *time slice queue* in order to make it *wait* for an event outside the process.

DESCENDANT PROCESS An *internal process* described as a grandchild, or great-grandchild, etc. of another internal process in the *process hierarchy*.

DOCUMENT A physical medium in which a specific collection of data is stored, e.g. a roll of paper tape, a deck of punched cards, a printer form, a reel of magnetic tape, or a data area on the backing store.

EVENT QUEUE The queue in which a *process* receives messages and answers from other processes.

EXTERNAL PROCESS A general term for an *area process* or a *peripheral process*.

INTERNAL INTERRUPT An interruption of an *internal process* caused by protection violation, arithmetic overflow, or erroneous monitor call, or caused by the *parent* of the process.

INTERNAL PROCESS The execution of one or more interruptable *programs* in a contiguous storage area identified by name.

MONITOR A resident *program* with complete control of storage protection, input/output, and interrupts. It contains descriptions of all *processes* and controls the sharing of computing time among them. It also contains procedures that processes can call in order to *create* and control other processes and communicate with them.

MULTIPROGRAMMING Simultaneous execution of several *programs* loaded in the store by multiplexing of the central processor controlled by clock interrupts.

OPERATING SYSTEM A *program* that controls the scheduling and resource allocation of other programs in order to obtain a specific mode of operation, e.g. batch processing, *real-time scheduling,* or *time-sharing.* During execution an operating system is synonymous with a *parent process.*

PARENT PROCESS An *internal process* that *creates* and controls another internal process. The latter is described as the *child* in the *process hierarchy.*

PERIPHERAL PROCESS Input/output or interrupt signals of a peripheral device identified by name. It usually involves the use of a specific *document* mounted on the device.

PROCESS A general term for an *external process* or an *internal process.*

PROCESS HIERARCHY A family tree describing the control relations among *internal processes.* Processes are called *ancestors, parents, children,* and *descendants* depending on their position in the hierarchy relative to a given

process. An internal process can only *start*, *stop*, and *remove* its own child processes and their descendants.

PROGRAM A collection of instructions specifying a computational process.

REAL-TIME SCHEDULING A mode of operation in which program execution is synchronized with real time or external events supervised by the computer.

REMOVE PROCESS (TO) To remove a table within the monitor describing a *process* by its kind, name, *resources*, *event queue*, and current state.

RESOURCES A general term for the amount of computing time, storage, protection keys, peripheral devices, message buffers, and process descriptions available to an *internal process*.

RUNNING PROCESS An *internal process* in the *time slice queue* which is executing instructions or ready to do so.

START PROCESS (TO) To *activate* an *internal process* on request from its *parent*.

STOP PROCESS (TO) To *delay* an *internal process* on request from its *parent*.

TIME-SHARING A mode of operation in which several programs share a common storage area during execution by frequent swapping of programs between the internal store and the backing store.

TIME SLICE QUEUE A queue of *internal processes* that share computing time in a cyclical manner.

WAITING PROCESS An *internal process* that has been removed temporarily from the *time slice* queue in order to wait for an event outside the process.

INDEX

Photografic Reprint by
R. Roussell, Denmark

A/S REGNECENTRALEN

SCANDINAVIAN INFORMATION PROCESSING SYSTEMS

HEADQUARTERS: FALKONER ALLÉ 1 · DK-2000 COPENHAGEN F · DENMARK
TELEPHONE: (01) 105366 · TELEX: 6232 RCHQ DK · CABLES: REGNECENTRALEN

AUSTRIA
BELGIUM
DENMARK
FINLAND
GERMANY
HOLLAND
NORWAY
SWEDEN