Title:

EXTENDED RC3600 COROUTINE MONITOR

MUSIL Programmer's Manual

A/S REGNECENTRALEN

RC SYSTEM LIBRARY: FALKONERALLE 1 DK-2000 COPENHAGEN F

RCSL No: 43-GL 3168
Edition: April 1978
Author: Klaus Hansen

Keywords:

Coroutine Monitor, MUS, MUSIL.

Abstract:

Reference manual for MUSIL programmers, using the new set of code procedures to access the functions in the extended coroutine monitor.

Printed by A/S Regnecentralen, Copenhagen

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

CONTENTS PAGE

LIST OF ILLUSTRATIONS.

This page is intentionally left blank.

1. PREFACE. 1.

The present maunal is a reference manual for the extended coroutine monitor on RC3600, and is intended for MUSIL programmers.

The manual is new, as no manual has been available for earlier coroutine facilities in MUSIL, and the use of these facilities has been restricted to internal use at the development department in RC.

The overall design may present some conceptual difficulties for the reader of this manual, but these problems arise from difficulties in the implementation caused by the limitations present in the implementation of MUSIL, and are difficult to explain for any reader but those with a very detailed knowledge of MUSIL. We hope that any problems arising in this connection will only be of small significance, and may be solved by examining the examples in chapter 7 or by trying to run a simple program.

The following manuals may be of additional interest:

RCSL: 43-GL 4715	Extended RC3600 Coroutine Monitor Programmer's Manual (October 1977)
RCSL: 43-GL 4475	Coroutine Monitor Testoutput Program User's Guide (July 1977)

2. INTRODUCTION TO RC3600 COROUTINES.

The use of the term 'coroutine' has not yet been standardized in the technical litterature, and consequently appear confusing, as the word 'coroutine' is used about programming tools ranging from subroutine - like modules calling each other in a highly symmetric way, to tasks running in a general multiprogrammed system which interact in a certain simple way.

The name 'coroutine' was coined by Conway in 1958. He used it for subroutines in a system, where each subroutine is written as if it is the main program and the others simply subroutines called by it (fig. 1). This is done by merging of call and return, and dynamically changing entry points in the routine. The mechanism is described in detail in D.E.Knuth: The Art of Computer Programming, vol. 1. The concept is found as well in SIMULA 67.

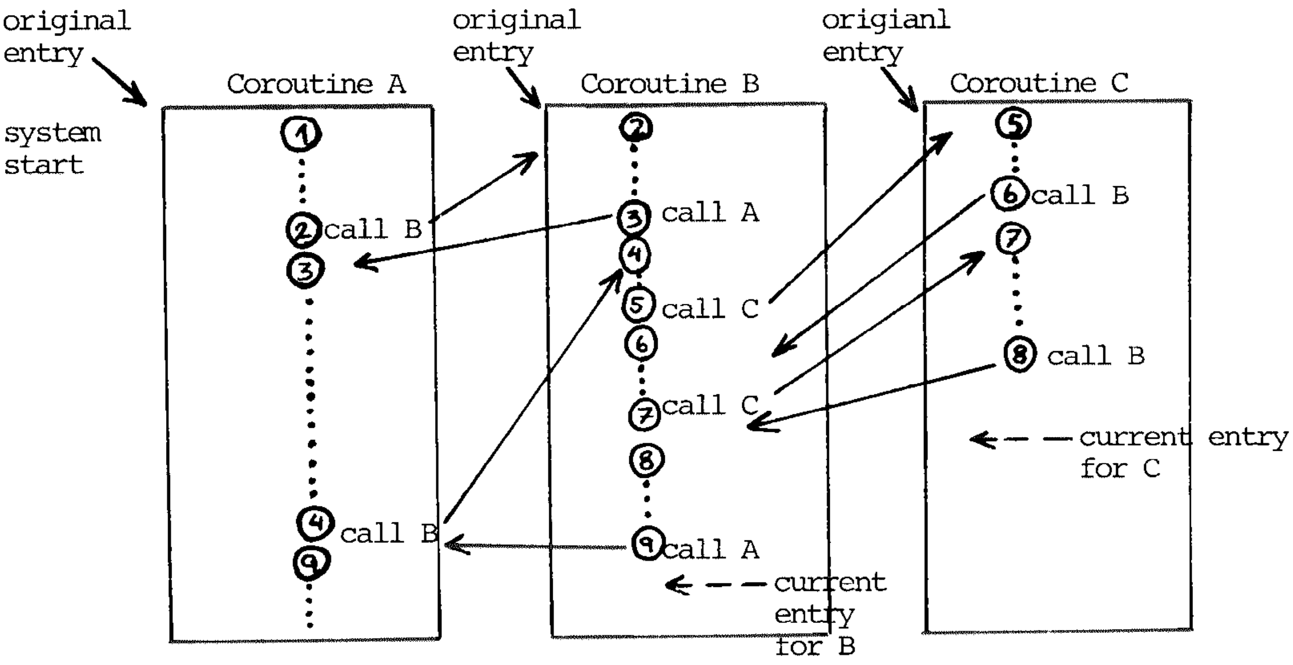


Figure 1. Conway's coroutines. Call mechanism.

Some of the characteristics of such a system are:

- one coroutine at a time is active (using the CPU). This means that a coroutine always runs in 'disabled' mode and consequently has free and exclusive access to any variable in the system.

- one coroutine at a time may wait for external events as interrupts and other I/O events. That coroutine is the active coroutine.

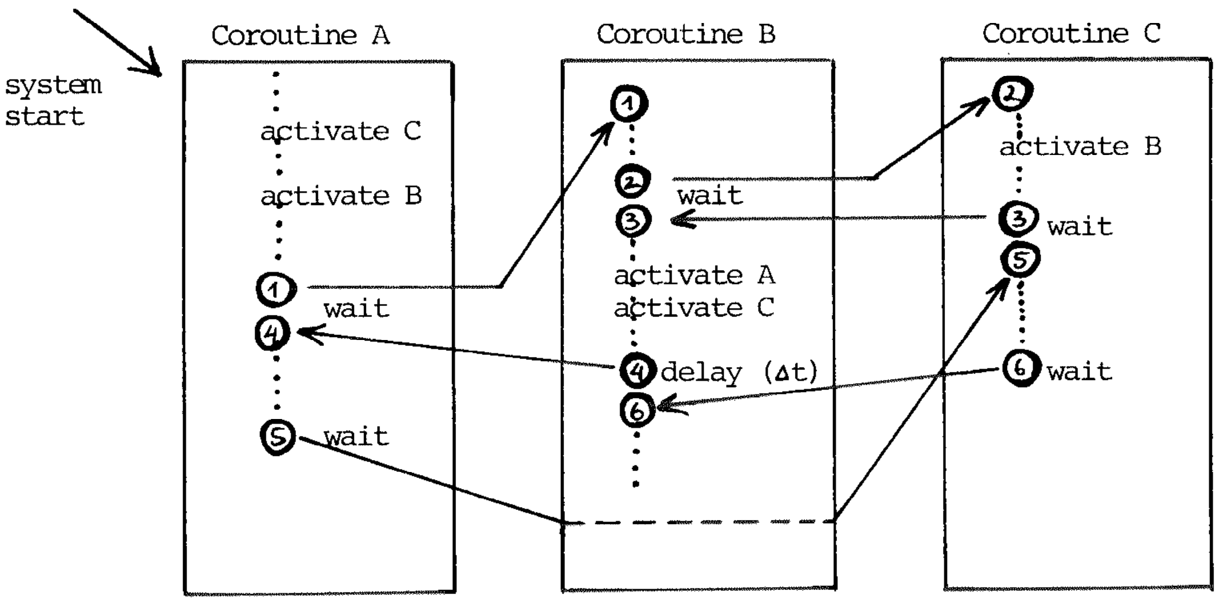
- protection of data areas shared between coroutines against simultaneous access is not critical.

- the scheduling of CPU-time is 'primitive' - each coroutine uses as much CPU-time as needed. No protection exists against one coroutine monopolizing the CPU. Implementation of the call mechanism is very simple.

- as each coroutine may be written as if it is the main program calling subroutines, a certain amount of modularity and independence between modules is enforced on the programmer.

An important extension to this simple coroutine system is found in the book by D.E. Knuth. A coroutine is allowed to activate several other coroutines before stopping, i.e. in the terminology of subroutines simultaneously to branch to several entry points. Activation of another coroutine and suspending own executing are thus separate functions (fig. 2). The 'current entry point' (fig 1) is replaced by a 'waiting point'.

Selection of the coroutine which is allowed to run next (in case of several activated coroutines), is done by a central logic.



① B & C activated, B selected

④ A & C activated, A selected

⑥ No coroutines active. B activated when requested.

Figure 2. Coroutine System, several activated coroutines.

A primitive sort of internal event has been added in form of the activate/wait pair. A coroutine may further wait for a specified time ('delay 2 secs'), which can be considered as a hidden timing coroutine activating the caller after the specified interval of time. Scheduling of the CPU is still 'primitive', but the central logic has decisional abilities, and is extended with a timing function.

The RC3600 Coroutine Monitor consists of such a central logic and a collection of reentrant functions. If has developed from the system mentioned above be emphasizing the role of internal and external events, and by making the coroutines more independent. These design principles are found in the RC4000 operating system BOSS2 too.

An RC3600 coroutine is either active (activated) or waiting for some internal or external event. An active coroutine is either placed in the active queue, i.e. activated and waiting for access to the CPU, or is the singular coroutine executing instructions.

A waiting coroutine may wait for timer, internal events signalled by other coroutines (see below), or external events such as interrupt or incoming messages and answers. It is possible to wait for more than one type of event at a time.

The concept of internal events needs a little expansion.
The activate/wait pair (fig. 2) functions as a sort of synchronization between the 'activate' coroutine and the 'wait' coroutine, informing the waiting coroutine that a certain mutually specified event has occurred thus enabling it to resume its activities. This simple coroutine interaction has been extended in several ways:

- the wait/active pair need not be executed in that order.

- the number of wait and activate calls need not match.

- the activation event may be provided with one of several datatypes.

- activation is done by a sort of indirect addressing, and several coroutines may compete for activation by a certain internal event.

The extension is linked to the introduction of the semaphore concept and some functions (signal/wait) working on semaphores.

A semaphore is a data structure containing a state variable and some additional information about queues of waiting coroutines and signalled events. A semaphore is in one of three states:

NEUTRAL: The number of signals equal the number of waits. The semaphore is initially in this state.

OPEN: There has been more signals than waits. This means that a coroutine is not delayed when executing a wait.

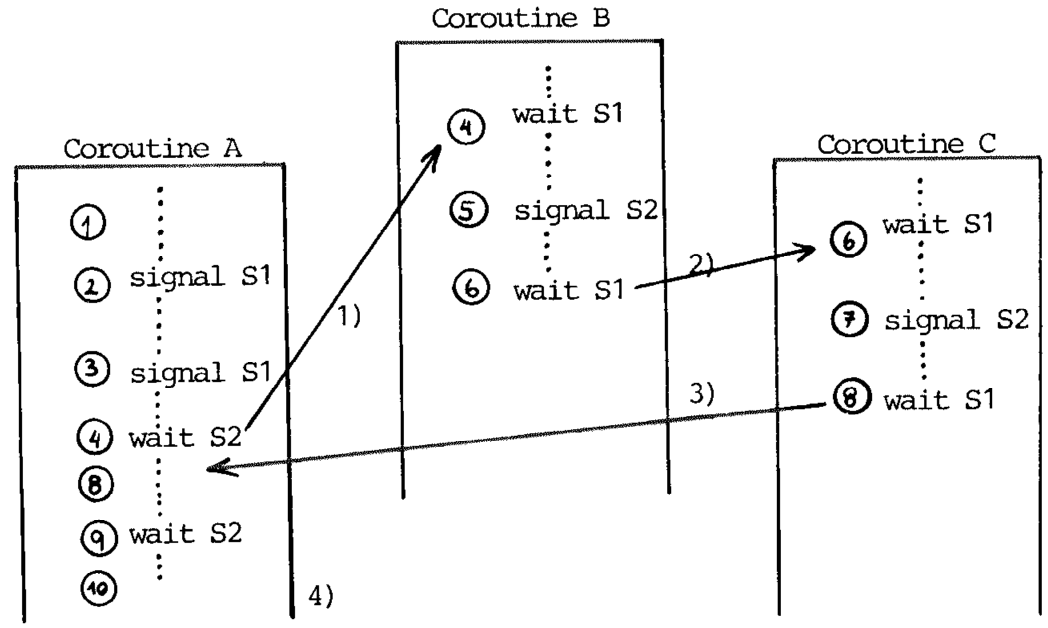
CLOSED: There has been more waits than signals. Consequently a signal will activate one waiting coroutine.

The associated operations signal/wait will work like this:

```
Signal (sem):   if sem. state <> CLOSED then
                begin
                    remember one more activation (sem);
                    if sem.state = NEUTRAL then sem.state = OPEN
                end
                else
                begin
                    activate one waiting (sem);
                    if last one activated then sem.state = NEUTRAL
                end;

Wait (sem):     if sem. state <> OPEN the
                begin
                    if sem.state = NEUTRAL then sem. state = CLOSED;
                    delay until activation (sem, this coroutine);
                    ! which activates next !
                end
                else
                begin
                    delete one activation (sem);
                    if no left then sem.state = NEUTRAL
                end;
```

Figure 3 shows three coroutines using signal/wait. Internal events with data are called operations. The signal/wait pair used in connection with operations have an extra parameter for the operation and the 'remember one more activation' and 'delete one activation' routines user these parameters.



1) B&C activated, B allowed to run,

2) C&A activated, C allowed to run,

3) A activated,

4) S2 is OPEN, so A is not stopped.

Result of step:		1	2	3	4	5	6	7	8	9
Semaphore	S1:	C	C	N	N	N	C	C	C	C
	S2:	N	N	N	C	N	N	O	O	N
Active queue	:	A	A	A	B	B	C	C	A	A
	:		B	B	C	C	A	A		
	:			C		A				

C = CLOSED
N = NEUTRAL
O = OPEN

Figure 3. 3 interacting coroutines.

The RC3600 extended coroutine monitor supports three types of semaphores with corresponding signal/wait pairs:

- simple semaphore: No datatypes are connected with a signal/wait. The semaphore contains the number of signals not waited for, and a queue of waiting coroutines, if this number is negative.

- chained semaphore: A single type of operation is used with optional data field. The strategy is: first signalled, first delivered to a waiting coroutine.

- general semaphore: The operations have a 16-bit type field. Some events have preassigned types as timer and the external events message, answer and interrupt, which leaves 12 bits for user-specified internal events. An operation is delivered to a waiting coroutine if the type and a mask specified in the call has common bits in a first come, first delivered fashion.

In addition to the semaphore functions, the RC3600 coroutine monitor supports functions as:

- delay: Delay a coroutine a certain amount of time.

- pass: Allow other activated coroutines to run. Can be used when a time-consuming operation is executed in a coroutine.

- send message
- wait answer : Interface to the MUS I/O system.

The characteristics for a system of coroutines using the coroutine monitor facilities are (compared with the primitive coroutine system):

- One coroutine at a time uses the CPU. Other active coroutines are placed in the active queue waiting for the CPU to be passed over.

- Coroutines which are not active, are in a waiting point and are placed in one of the following queues:

    1) delay queue, waiting for timer,
    2) answer queue, waiting for an external answer by means of coroutine wait answer,
    3) waiting queue for a semaphore, waiting for some event.

- One coroutine in the system may wait for external events like messages and interrupts.

- Scheduling of CPU time is 'primitive' - each coroutine uses as much CPU time as needed and allows other coroutines to get the CPU time by executing a wait, delay or pass. No protection exists against a coroutine monopolizing the CPU. Implementation of the CPU switching mechanism is kept simple.

- One coroutine system equals one MUS process. The different coroutine systems communicate by means of the standard MUS communication primitives. Scheduling of the CPU between the different 'active' coroutine systems (processes) is done by the standard MUS scheduler, using a priority 'round-robin' method.

A system of coroutines resembles a multiprogramming system (like RC3600 Multiprogramming Utility System = MUS), where the coroutines are equivalent to concurrent processes and the signal/ wait pairs correspond to sendmessage/waitanswer and waitevent/ returnanswer. There are, however, significant differences, which make a coroutine system (= one process) a useful and significant alternative to a system consisting of several cooperating processes. The following table outlines the differences between processes and coroutines, intended for the reader, who is familiar with elementary concepts as exclusive access to shared data, synchronization of parallel processes etc., as found in e.g.

P. Brinch Hansen: Operating System Principles
(Prentice-Hall 1973)

Concept	Multiprogrammed solution, several processes.	Coroutine solution, several coroutines in one process.
Module	= Process.	= Coroutine.
Communication between modules:	Other process known by name. Messages/answers are exchanged.	Other coroutines only known to a limited extent. Communication done by signal/wait using common semaphores.
Critical region and shared data with exclusive access:	Exclusive access to data areas is ensured by associated access with a certain message/answer. The current owner of the message/answer has exclusive access.	Exclusive access is ensured between waiting points. If exclusive access is wanted for a piece of code containing several waiting points, signal/wait is used.
	Critical regions, i.e. non-entrant code shared between several processes may be protected in the same way or by preventing a process shift by disabling interrupts.	Critical regions are protected automatically by placing them between waiting points.

CPU overhead:	Low to high depending on hardware facilities.	Very small.
CPU guaranteed:	Yes, depending on relative priority. A loop in another process with same or lower priority can not prevent execution.	No, a loop in a coroutine will stop the whole system of coroutines.
Short realtime response guaranteed:	Processes with high priority have immediate response.	No.
CPU usage:	Any range of CPU load.	CPU load preferably low.
Core overhead:	Process descriptor - contains a save area for all hardware and software registers used by the processes, as an interrupt and shift to other process may happen at any instance of the execution.	As shift to antoher coroutine happens at well defined places, overhead can be kept small.
Use when:	Closely interacting activities with medium to high CPU usage.	Closely interacting activities, I/O or event bound, low CPU usage.

3. SURVEY OF FUNCTIONS AND DATA. 3.

The present manual contains in chapter 5 functional descriptions for the set of codeprocedures and in chapter 4 description of the different data structures used in the MUSIL implementation of the extended coroutine monitor for RC3600. The descriptions are centered towards the functional descriptions of the procedures which occupy a greater part of the manual.

The data structures include:

- Some locations in the process descriptor for the program (which is generated by the compiler). These locations are initialized by means of INITCOSYS and SETUSEREXIT (see 5.4).

- One coroutine descriptor for each coroutine in the program. The system part is initialized by DEFCOROUT (see 5.4.2), and is 18 bytes long.

- Operations, which are exchanged between coroutines. System operations(of size 26 bytes) used when processing messages and answers are initialized by INITCOSYS. Other internally used user operations (at least 4 bytes) are created by CREATEOPS (see 5.3.5).

- Operation descriptors used to access operations. The size is minimum 6 bytes.

- Semaphores. Both simple (2 bytes)and general semaphores (10 bytes) are supported. General semaphores are initialized by INITGENSEM (see 5.1.2).

- Stacks, used when calling reentrant MUSIL procedures, are initialized by RESETSTACK (see 5.3.1).

- Files. When used in connection with reentrant multiple-incarnation coroutines, the buffer part of a file needs special initialization by INITZONE (see appendix D). Files may additionally cause problems in connection with the allocation of message buffers done by the compiler.

Appendix C contains a summary of data formats which is intended to be used especially with program dumps.

The set of procedures includes:

- Synchronization primitives, section 5.1:

simple semaphores	: signal/waitsem
general semaphores	: initgensem/waitgeneral/signal general.

- Interface to other MUS programs, section 5.2:

sending messages	: csendmessage/releaseanswer
receiving messages	: returnanswer

- Coroutine utilities, section 5.3:

procedure reentrancy	: resetstack/savelink/return
miscellaneous	: pass/cdelay createops changemask

- Initialization procedures, section 5.4:

system areas	: initcosys/setuserexit
coroutines	: defcorout

- Testoutput procedures, chapter 6:

user testpoints	: testpoint

- A selection of non-coroutine utilities, appendix D.

Appendix B contains a summary of the procedure declarations.

4. DATA DECLARATIONS.

The codeprocedures use various data structures such as semaphores, coroutine descriptions, stacks and system areas. These data structures are declared mainly in the variable section (after VAR) in a MUSIL program, and are composed of the standard MUSIL types integer, string, file and record. The codeprocedures make use of knowledge about the storage allocation algorithm in the MUSIL compiler, and the sequence of declaration may in several cases by significant for the proper functioning of a coroutine system. The concept of swapping data areas may in some connections have effect on where certain user variables are allowed to be placed.

The formats of the various data structures and restrictions in their use are described in the following sections.

4.1 System Data Areas.

The system data areas comprises words in page zero of the RC3600 storage, 12 words in the process descriptor of the actual coroutine program, and a 20 words long area containing the testrecord and some anonymous variables. The locations in page zero contains entrypoints for the coroutine procedures, a pointer to current active coroutine and a location shared with the optional testoutput program, and these locations are initialized when the coroutine monitor is loaded.

The remaining areas are initialized by INITCOSYS (see 5.4.1), and SETUSEREXIT (see 5.4.3) if necessary. Note that the program should be compiled with MODIF C (see 8) to make room in the process descriptor.

4.2 Coroutine Variables. 4.2

Each coroutine in the program is described by a coroutine descriptor which is 9 words or 18 bytes long. In addition to the coroutine descriptor, which provides place for system information, the user may specify some variables used by the code executed by the coroutine.

It may be convenient by some applications to use one piece of code as the body of several coroutines, acting on separate data areas with the same structure. This is typical for applications where several, nearly identical devices are processed by each one coroutine, and the differences in the treatment are so small that the code executed is nearly identical, too. The implementation of such systems is not trivial, as addressing of many identical items, e.g. in form of indexing in arrays, is not present in MUSIL.

The present coroutine implementation supports, however, such reentrant code acting on incarnations of a set of variables by simulating arrays using swapped data areas. The user declares his data area and a swapping area, and the system will now ensure that the data in the user area always belongs to the current incarnation. This is done by swapping old data out and actual data in, before the coroutine code is reactivated (figure 4). The procedure SETUSEREXIT (see 5.4.3) sets a system variable to point to the code performing this swapping.

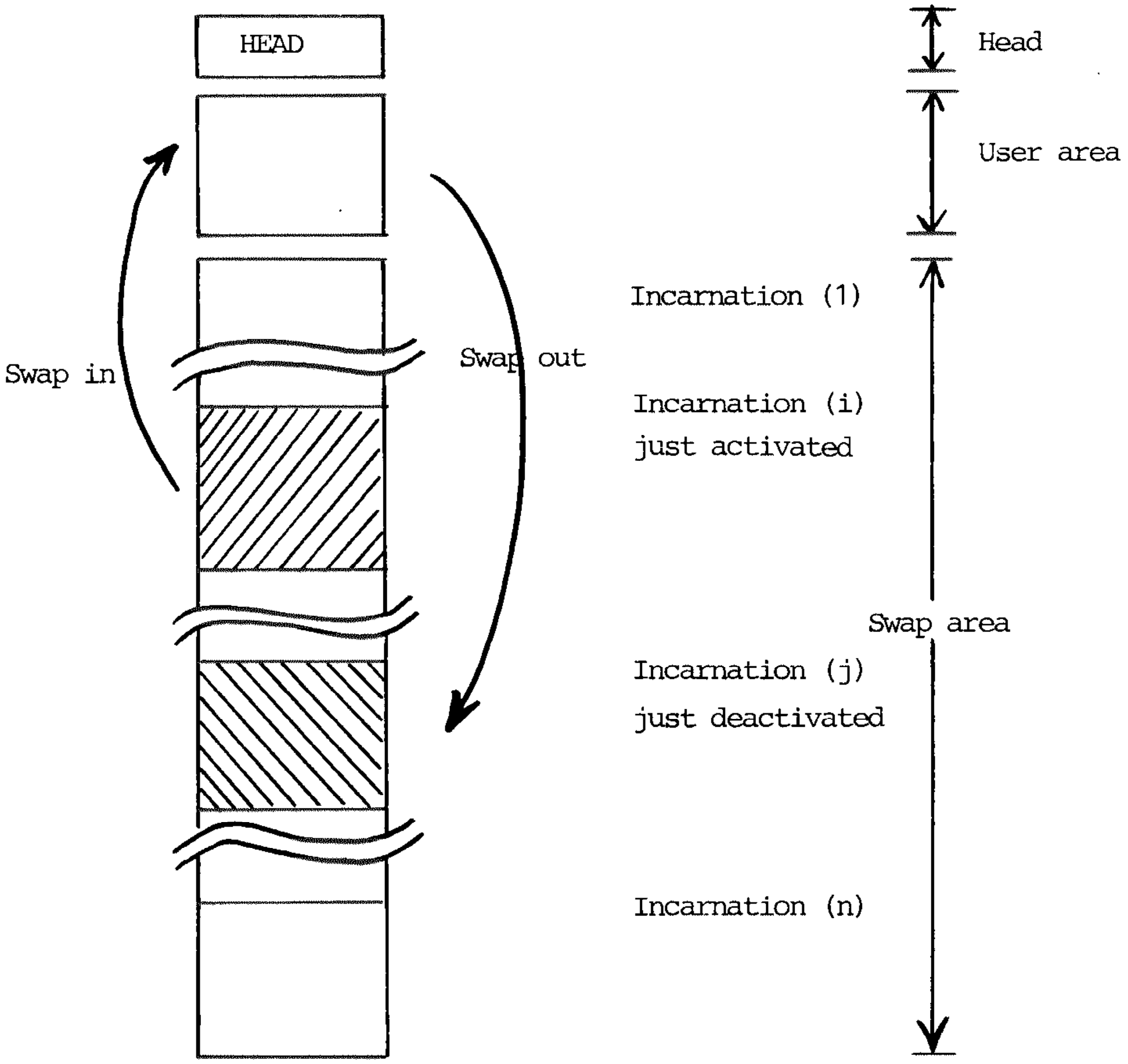


Figure 4. Multiple incarnations.

4.2.1 Single Coroutine.

The coroutine descriptor for a single coroutine, i.e. a coroutine using one piece of code and one data area, is composed of a system part declared as string (18), and a user part of any structure and length. The user part may be declared anywhere. The system part uniquely defines the coroutine, and is the first parameter in the defining call of DEFCOROUT (see 5.4.2.). If the program consists solely of single coroutines, a call of SETUSEREXIT (see 5.4.3) is not needed.

4.2.2 Multiple-Incarnation Coroutine.

Multi-incarnation coroutines are coroutines, using the same (reentrant) code acting on several identically structured data areas. It is not possible to select a certain incarnation of data in a MUSIL program at run-time, as all addresses are fixed at compile-time and compiled into the code as an integral part of the statements. The present solution to this addressing problem is to simulate arrays by swapping. The reentrant statements are coded as if there were only one coroutine, acting on a dummy area declared by the user. The actual data for a specific incarnation are stored in the swap area and are moved into the dummy area immediately before the execution of the corresponding code is resumed. This swapping action is done by means of code initialized by SETUSEREXIT (5.4.3). If any coroutines in a program uses the multi-incarnation facilities then this procedure should be called after INITCOSYS. The variables of multi-incarnation coroutines are declared as follows.

For each set of coroutines sharing code and data structure, three areas are declared in exactly this order with no other declarations between:

(1) Header - a dummy coroutine descriptor, as string (18).

(2) User working area - contains all variable declarations for variables owned by an incarnation.

(3) Swap area - contains actual values for the variables owned by the different incarnations, and the associated coroutine descriptors. The length of this area is a multiple of the combined length of (1) and (2).

The user area (2) may contain declaration of variables of any MUSIL type with the following restrictions:

(a) The body of a general semaphore (see 4.4.2) is not allowed, but must be placed in a fixed area (see 5.1.2).

(b) The buffer part of a file can not be allocated in a swapped data area, but has to be placed in a fixed area (see 5.2.3).

The length of the user variable area is used in the call of DEFCOROUT (see 5.4.2). It is given in words, and does not include the length of the coroutine descriptor. It is computed as the sum of the lengths of all variables declared, by using the following information.

An integer is one word long.

A string (l1) is (l1+1)//2 words long.

The length of records is computed in this way:
The declaration is of the general format

```
record
      V1     : T1 ;
      V2,V3  : T2 ;      !two fields of same type!
         ...
      Vi     : Ti ;
      Vj     : Tj  from pj ;
      ...
      Vk     : Tk ;
      Vn     : Tn  from pn
      ...
      Vz     : Tz
end;
```

First compute the sum of the lengths (in bytes) for all tields without the keyword from, using the length of integers= 2 and length of string (l1) = l1. This sum gives the combined length of sequential fields, LS in bytes.

Second, compute the maximum value of lj + pj - 1 where lj and pj are the length in bytes resp. the position of fields with the keyword from. This maximum value gives the maximum extend of the positional fields, LP, in bytes.

The greatest of the two values, LS and LP, is called LTOTAL, and indicates the length of the record in bytes. The length in words is (LTOTAL+1)//2. An example: the length of

```
record
      A,B :  INTEGER;     !always at odd addresses!
      C   :  STRING (9);
      D   :  STRING (1);
      E   :  INTEGER;     !odd address              !
      F   :  STRING (7) from 2;
      G   :  STRING (3);
      H   :  INTEGER from 5;
end
```

is 10 words. (LS = 19, LP = 8, LTOTAL = 19).

The length of a file is as follows. If the declaration is

```
file "...",....,.n1,l1,<format>
of   ....... ;!length = 12!
```

then the length in words is 26 + n1 * (7 + (l1+1)//2). Note that the length 12 is only significant when <format> is F or FB, where the relation l1 = p * 12 shall be satisfied for some p $\geq$ 1.

The length of the swap area (3) is

n * (L + 9) * 2 bytes

where n is the number of incarnations, and L the total length in words of the user area.

When files are used by multiple-incarnation coroutines, action should be taken to allocate a proper number of extra messages buffers. This is needed because the MUSIL compiler only recognizes one file declaration where actually several files may be active. The message buffers needed have no connection at all with the system operations used by CSENDMESSAGE or WAITGEN, and have to be allocated by a call of the codeprocedure CREATEMESSBUFS. This codeprocedure is described in appendix D.

The number of message buffers needed is determined by the number of buffers used in the files. A file declaration like

```
file "....",...., n1, ....
of   .......
```

needs n1 message buffers. The total number of message buffers to be created is computed in this way.

First, for each multi-incarnation coroutine declaration, compute the sum of buffers used (as n1 above) and call it stotal. If the number of incarnations is m, then (m - 1) x stotal additional message buffers may be needed for these m coroutines. Actually fewer may be in question, depending on the number of files being used simultaneously.

Second, take the sum of these needs to get the total number to be created.

4.3 Operations.

Operations are used by SIGGEN and WAITGEN (see5.1.2) to exchange data between coroutines and by CSENDMESSAGE, RELEASEANSWER (see 5.2.1) and RETURNANSWER (see 5.2.2) when communicating with other processes. The procedures INITCOSYS (see 5.4.1) and CREATEOPS (see 5.3.5) are used to create such operations.

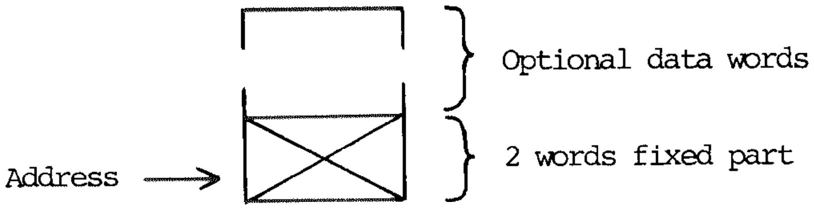
Operations may be divided in three kinds:

(1) Standard operations, defined in the coroutine monitor. The timer operation is a standard operation.

(2) System operations, created by INITCOSYS. These operations have a fixed format and is used to send messages, and receive events like messages and answers.

(3) User defined operations, created by CREATEOPS. The format and use is defined by the user.

Operations have the following general layout:



They are addressed by a word address pointing to the last word of the operation.

The use of word addresses in MUSIL programs is rather cumbersome, as pointer types are not a language feature, and codeprocedures or coding tricks has to be used instead. The operation descriptor has been introduced to provide an easy way of using operations. An operation descriptor is a record containing a reference to the operation in question, a field reflecting the type, and a data field with an image of the data part of the operation, if present. All procedures use this record when referencing operations, and moves an appropriate number of words between the image and the data part of the operation.

4.3.1 Operation Descriptor.

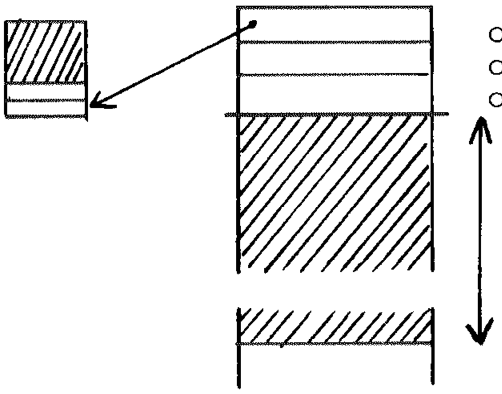
An operation descriptor consists of 3 integer fields and an optional data field of appropriate, free format and length. Note that a whole number of words, or an even number of bytes, are moved by the procedures. The operation descriptor is specified to a procedure as parameter by the first, integer field.

The format is

```
record
      opadr : integer;
      optype: integer;
      opsize: integer;

   ! and optional data area !
end;
```

The contents is



opadr, reference to an operation or 0
optype, a copy of the type in the operation.
opsize (in words)

opsize words are moved by a procedure

An operation descriptor shall be placed either at a fixed place, or in the user part of the coroutine descriptor (see 4.2) when used by WAITGEN. The opadr and opsize fields should be initialized, to 0 and a proper length respectively.

4.3.2 Operations. 4.3.2

Operations are provided with a 16 bit long type. The bit assignments are

bit	name	use
15	TIMER	Type of the standard operation signalled in case of time-out.
14	ANSWER	Type of the system operation signalled to an answer semaphore when a message sent by CSENDMESSAGE is answered.
13	MESSAGE	Type of the system operation signalled when a message arrives.
12	INTERRUPT	Cannot be used by a MUSIL program. Reserved for system use.
11 . . . 0	USER	User defined operation types. Any combination of these 12 bits may be used.

A pool of free system operations is created by INITCOSYS (see 5.4.1) and user operations may be created by CREATEOPS (see 5.3.5).

4.4 Semaphores. 4.4

The coroutine monitor supports two kinds of semaphores, simple and general, and each kind has a set of procedures associated with it. Simple semaphores are used by SIGNAL and WAITSEM, section 5.1.1, and general semaphores by SIGGEN, WAITGEN, INITGENSEM and CSENDMESSAGE, section 5.1.2 and 5.2.1.

4.4.1 Simple Semaphore.

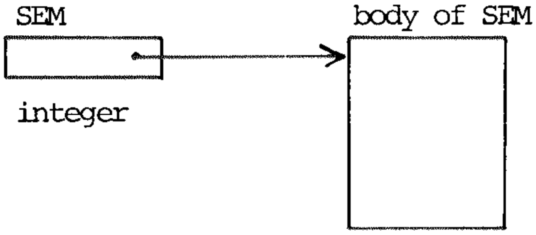
A simple semaphore is of type integer, and the 16 bits are divided into two fields

semaphore state	bit 0-14	bit 15
OPEN	count > 0	1
NEUTRAL	0	1
CLOSED	head of waiting queue	0

The count determines how many WAITSEM calls that may be executed before the semaphore becomes closed. The semaphore can be initialized to NEUTRAL, value 1, or OPEN.

4.4.2 General Semaphore.

A general semaphore consits of a body of length 10 bytes and a reference to this body of type integer.



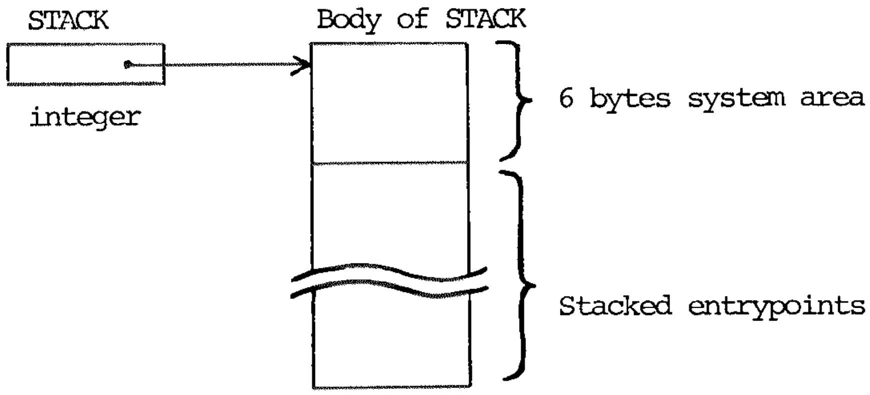
The body contains a waiting queue of coroutines waiting for some combinations of operations, and an operations queue containing operations of different types not yet waited for. As the body may appear in the delay queue, it must occupy a fixed location and can consequently not be placed in swapped data areas, e.g. the user part of a multi-incarnation coroutine descriptor.

A general semaphore has no specific status. The procedure INITGENSEM (section 5.1.2) initializes the semaphore so the queues are empty. The reference is always used to address the semaphore. It is allowed to have multiple references to the body.

4.5 Stacks.

Stacks are used in connection with reentrant procedures, and are used by RESETSTACK, SAVELINK and RETURN, section 5.3.1.

A stack consists of a body of variable length, and a reference of type integer to the body.



The system part contains current position of next free, maximum extent and an underflow/overflow trap. The stack is reset to empty by RESETSTACK. The stacked entrypoints are MUSIL interpreter program addresses, and are only accessible to the user by SAVELINK and RETURN. Each stacked entrypoint occupies 2 bytes.

5. PROCEDURES.

This chapter contains declarations, parameter conventions and functional descriptions for the MUSIL codeprocedures implementing the facilities in the extended coroutine monitor, and for some other procedures which might be of use.

Parameters refer the definitions in chapter 4: declarations. The internal names of as well procedures as parameters in a specific program may of course be changed to whatever else the programmer wants as a consequence of the MUSIL implementation of codeprocedures. Only the external P00.. names cannot be altered. It may, however, be convenient to use the names from this manual to increase readability.

5.1 Synchronization Primitives.

5.1.1 Simple Semaphores.

SIGNAL(SEM),
WAITSEM(SEM);

Declarations:

```
procedure SIGNAL (var SEM : integer);
codebody P0128;

procedure WAITSEM (var SEM : integer);
codebody P0127;
```

This pair of procedures use simple semaphores (see 4.4.1) to synchronize two or more coroutines.

Functional Descriptions:

SIGNAL(SEM): if SEM.state <> CLOSED then SEM.count:= SEM.count + 1 else remove and activate (SEM.queue);

WAITSEM(SEM): if SEM.state <> OPEN then insert and stop (SEM.queue, COROUT) else SEM.count:= SEM.count -1;

SIGNAL removes the coroutine which has waited for the longest time.

WAITSEM may delay the calling coroutine an undefined time, until a matching SIGNAL from another coroutine.

5.1.2 General Semaphores.

```
INITGENSEM(SEM,SEMAREA,DISP);
SIGGEN(SEM,OPDESCR.OPADR);
WAITGEN(SEM,EVENTMASK,OPDESCR.
        OPADR,DELAY);
```

Declarations:

```
procedure INITGENSEM (var    SEM       : integer;
                      var    SEMAREA   : string (1);
                      const  DISP      : integer;
codebody P0091;

procedure SIGGEN     (var    SEM       : integer;
                      var    OPADR     : integer);
codebody P0093;

procedure WAITGEN    (var    SEM       : integer;
                      const  EVENTMASK : integer;
                      var    OPADR     : integer;
                      var    DELAY     : integer);
codebody P0092;
```

These procedures are associated with the use of general semaphores (see 4.4.2). INITGENSEM generates and initializes one general semaphore, and SIGGEN and WAITGEN are used to synchronize two or more coroutines, and exchange operations of different, mixed types between the coroutines. WAITGEN is in addition used to synchronize with external events as messages and answers, and to support a timing facility. When accessing messages, one coroutine at a time may wait for the messages, using a general semaphore.

The procedure CSENDMESSAGE (section 5.2.1) has a general semaphore as parameter, to which the answer to the message will be signalled.

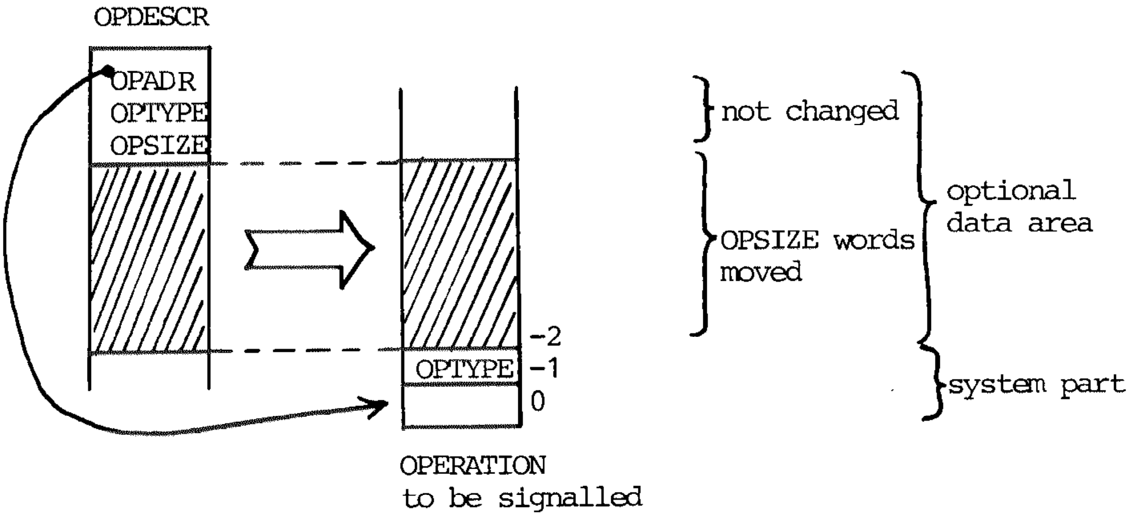
Functional Descriptions:

INITGENSEM(SEM,SEMAREA,DISP):
Initializes the body of the general semaphore SEM found in the 10 bytes SEMAREA(DISP).... SEMAREA(DISP + 9) to neutral. The variable used as first parameter SEM contains a reference to the body, and this reference is used in all codeprocedures requiring a general semaphore as parameter.

Note: The body cannot be a part of any swapped data areas as it requires a fixed location. Swapped data areas are e.g. user part of coroutine descriptors, when the coroutine in question is a member of a set (see DEFCOROUT, section 5.4.2).

SIGGEN(SEM,OPDESCR.OPADR);
The operation described by OPDESCR (see section 4.3.1) is signalled to the general semaphore referenced by SEM, by means of the following algorithm:

```
    P:= SEM.nxtco;
    while p <> nil do     ! search for someone waiting !
    begin
         if p.opmask and OPDESCR.TYPE <> 0 then goto 10 ! found !
         p:= p.next
    end;
    ! not found !
    chain (SEM.nxtop, OPDESCR.OPADR);
    exit;
    ! found     !
10: remove and activate (p, OPDESCR.OPADR);
    exit;
```



The field OPADR points to an operation, which has a data part of length at least OPSIZE words. OPSIZE words are moved from the data part of OPDESCR to the data part of the operation, as shown, the optype of the operation is changed to the value of optype in OPDESCR, and the operation is signalled to the semaphore. OPADR is then set to zero.

If OPADR was initially zero, the call of SIGGEN is dummy. If OPSIZE is too large, some words outside the operation will be destroyed, and this may cause unpredictable results.

WAITGEN (SEM,EVENTMASK,OPDESCR.OPADR,DELAY);
Performs a wait for operations as specified by EVENTMASK on the general semaphore referenced by SEM, with timer. The operation resulting is described in OPDESCR (see section 4.3.1). The algorithm executed is:

```
    p:= SEM.nxtop ;
    while p <> nil do        !scan for an appropriate operation!
    begin
         if p.opmask and EVENTMASK <> 0 then goto 10;
         p:= p.next;
    end;
    !not found !
    if DELAY=0 then
    begin
         OPDESCR.OPADR:=0; OPDESCR.OPTYPE:=1b15; !timer!
         exit;
    end;
    if EVENTMASK and (1b12+1b13) <> 0 then
    begin  !message or interrupt !
         CUR.MSEM:=SEM;
         CUR.MCOROUT:= COROUT;
    end;
    insert in end of chain (SEM.nxtco);  ! this coroutine is      !
                                         ! inserted in waitqueue  !
    if DELAY < $\infty$then
    begin  !set semaphore to wait for timer!
         insert in chain (SEM,CUR.DELAY QUEUE);
    end;
    activate next;     !next coroutine is allowed to run !
    !operation found  !
10: remove (SEM.nxtop,p);
    OPDESCR.OPADR:= p;
    !transfer contents of operation !
    exit;
```

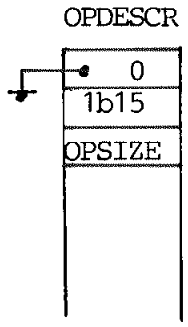
The values of parameters EVENTMASK, OPDESCR and DELAY on call and return is as follows:

Call:
EVENTMASK is a sum of events:

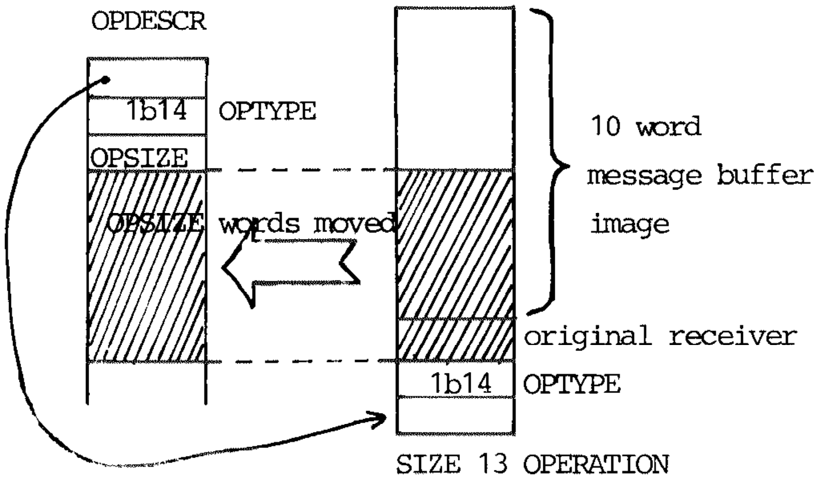
bit in EVENTMASK	expected event
1b15	timer. This bit is always considered set to one, independent of the value supplied in the call. DELAY specifies max. interval length: DELAY = 0 no delay, immediate return = -1 infinite delay value <> 0 or -1: number of 20 ms periods to wait.
1b14	answer.
1b13	message (only one coroutine at a time), (see figure 6).
1b12 NOTE:	cannot be used - may cause unpredictable results.
1b11 . . 1b0	user defined operations.

Return:
The value of DELAY is changed to remaining number of 20 ms. periods, zero if the reslut was timer, -1 if DELAY originally was -1. If message was waited for, the timing may be inaccurate in case the bufferpool with size 13 buffers was empty.

Resulting optype = 1b15 - TIMER:

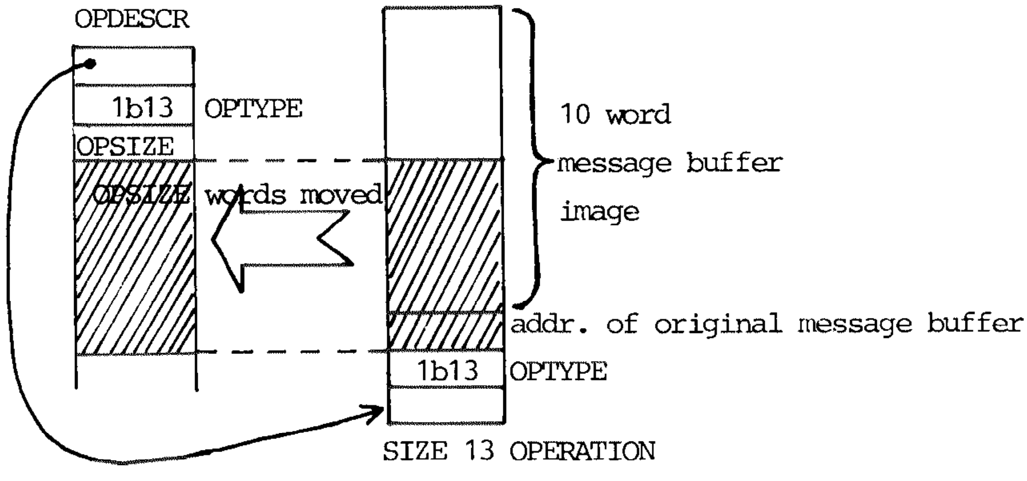


Resulting optype = 1b14 - ANSWER:



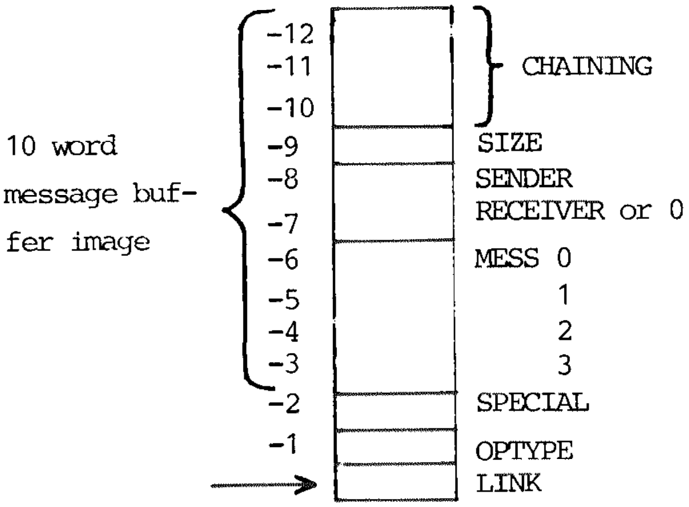
The size 13 operation is released after use by means of RELEASEANSWER (section 5.2.1).

Resulting optype = 1b13 - MESSAGE:

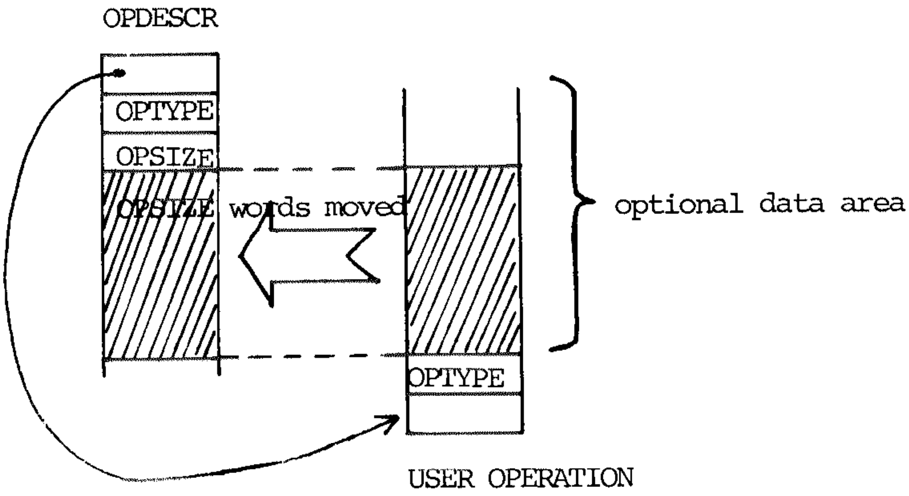


The answer is returned by means of RETURNANSWER (section 5.2.2).

A SIZE 13 operation has the following structure:



Resulting optype contains 1b0..1b11



5.2 Interfacing other Processes.

The procedures described in this section have two purposes:

1) to improve the facilities for sending messages to other processes and receiving the corresponding answers.

2) to allow easier access to messages received from other processes.

The two figures 5 and 6 show the general principles for 1) and 2). The numbering refers to the different steps taken.

Figure 5 shows how a message is sent, answered and the answer processed. The central point is that the answer can be recognized by the coroutine monitor central event logic and signalled to a semaphore specified by the user, who thus may wait for other answers and operations as well.

The message is sent (I) by means of CSENDMESSAGE, which uses one operation from the system operations pool (see section 4.3). As a result of CSENDMESSAGE, the message is chained into the event queue owned by the receiver (II). In due time the message is fetched (III), processed and returned (IV). The answer is chained into the event queue belonging to the sender, where the coroutine central logic will fetch it (V). The inspection of the event queue is done when all coroutines are waiting, or as a consequence of a call of PASS. The answer is recognized as sent by CSENDMESSAGE and is released from the event queue and signalled to the general semaphore given as the answer semaphore as an operation with type = answer (VI). This operation may be fetched by WAITGEN (VII) and used in one or more coroutines before it eventually is released and returned to the pool (VIII). It may then be used by another CSENDMESSAGE (IX).

Figure 6 is an illustration to 2) processing messages arriving from other processes. Messages are transferred to operations which may then be signalled between several coroutines.

A message is sent by another process (I) and will be placed in the event queue belonging to the receiving program (II). The coroutine central logic is activated when PASS is called, or whenever all coroutines are waiting and will inspect the event queue. When the message is found, and a coroutine is actually waiting for messages, a system operation is taken from the pool (III). The system operations used in this way are managed by a simple semaphore (BSEM), and the coroutine waiting for messages will use this semaphore to ensure at least one operation is present. The contents of the original message is copied into this operation (IV), which then is signalled to the general semaphore used by the message-waiting coroutine (V). The operation may now be used by one or more coroutines (VI). Eventually the last coroutine using the operation will call RETURN ANSWER (VII), which will return the answer to the original message (VIII) and release the operation for future use (IX).

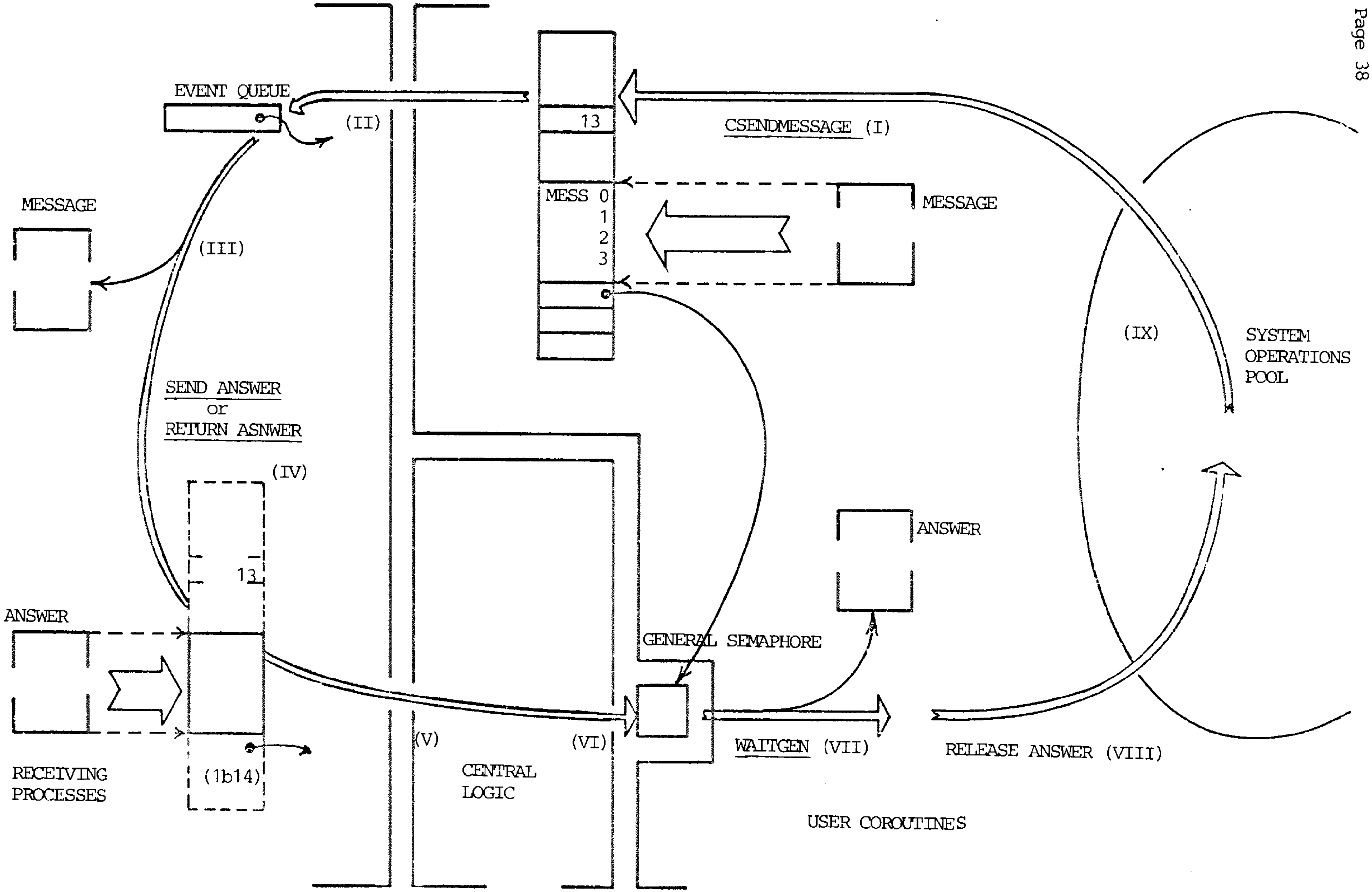


Figure 5. Senc g Messages.

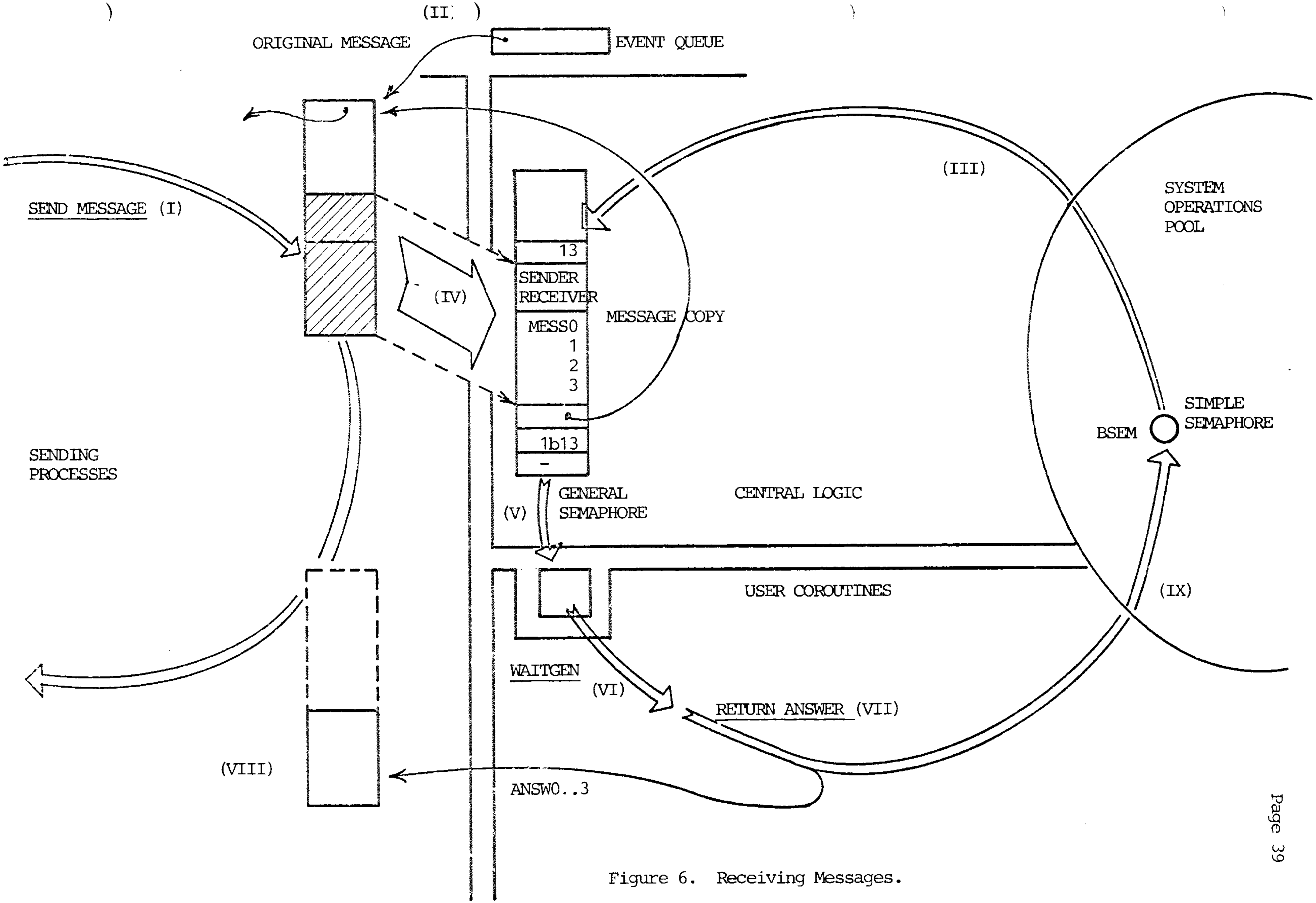


Figure 6. Receiving Messages.

5.2.1 Sending Messages.

```
CSENDMESSAGE (NAME,SEM,OPDESCR.OPADR);
WAITGEN (SEM,EVENTMASK,OPDESCR OPADR,DELAY);
RELEASEANSWER (OPDESCR.OPADR);
```

Declarations:

```
procedure CSENDMESSAGE     (const  NAME      : string(6);
                             var    SEM       : integer;
                             var    OPADR     : integer);
codebody P0095;

procedure WAITGEN          (var    SEM       : integer;
                             const  EVENTMASK : integer;
                             var    OPADR     : integer;
                             var    DELAY     : integer);
codebody P0092;

procedure RELEASEANSWER    (var    OPADR     : integer);
codebody P0098;
```

These procedures are used to send messages to other processes, and to accept the answers and return them to the pool (see figure 5). WAITGEN may be used for various other purposes, a description is found in section 5.1.2.

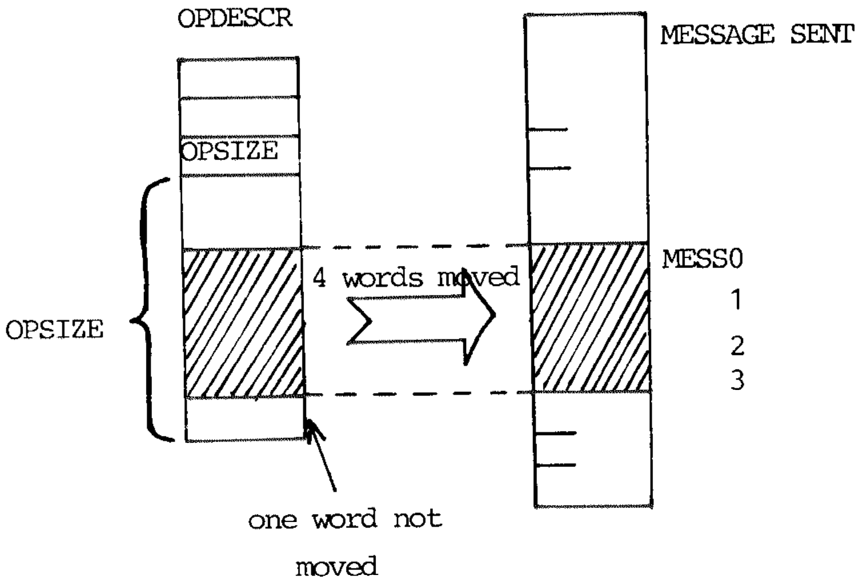
Functional Descriptions:

CSENDMESSAGE (NAME,SEM,OPDESCR.OPADR);

Sends a message to the process with name as given in the string NAME. One size 13 buffer from the pool is used and the message content is taken from OPDESCR (see section 4.3.1). The answer will be signalled to the general semaphore referenced by SEM.

A BREAK -3 is the result if no size 13 buffers are available. An answer with status 1b13 (NOT FOUND) and bytecount = 0 is generated if the receiver does not exist.

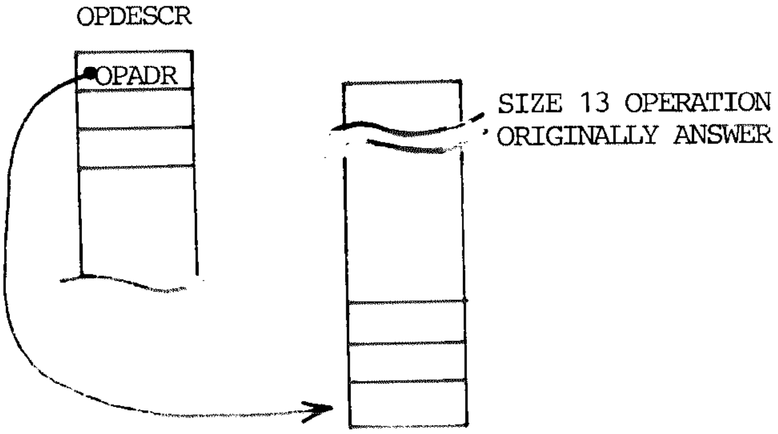
Data are moved from OPDESCR as shown:



The fields of OPDESCR are not changed.

WAITGEN (SEM,EVENTMASK,OPDESCR.OPADR,DELAY):
See section 5.1.2 for description.

RELEASE ANSWER (OPDESCR.OPADR);
The parameter OPDESCR (see section 4.3.1) describes a size 13 operation, which originally has been received as answer to a CSENDMESSAGE. No checking is, however, done to verify this.



The operation will be returned to the common pool, and OPADR will be set to zero. All other fields of OPDESCR are irrelevant and not changed. If OPADR was originally zero, the procedure is dummy.

5.2.2 Answering Messages. RETURNASWER (OPDESCR.OPADR);

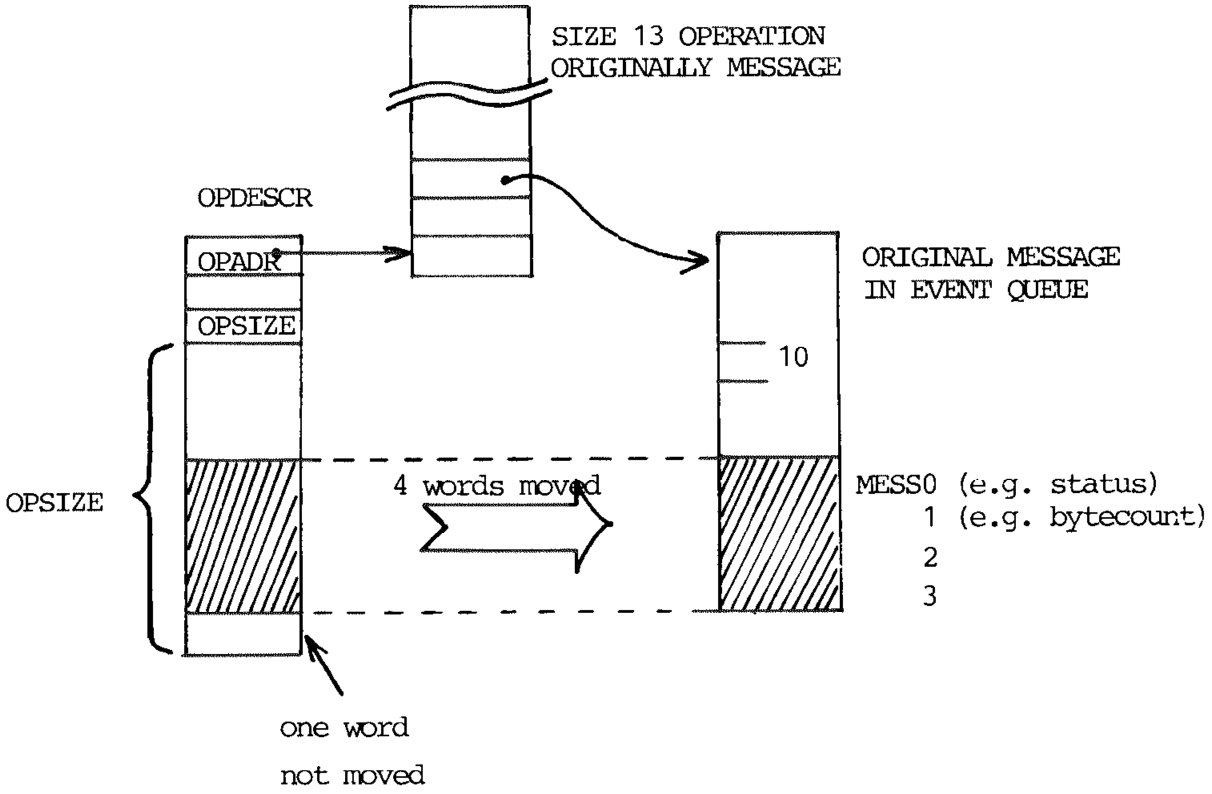
Declaration:

```
procedure RETURNANSWER    (var    OPADR    : integer);
codebody P0094;
```

Functional Description:

The parameter OPDESCR (section 4.3.1) describes a size 13 operation originally received as a message by means of WAITGEN (see figure 6). No checking is, however, done to verify this.

The operation is returned to the common pool, and the original message buffer is returned to the sender by means of MUS send-answer, with mess0.. mess3 set to the answer as shown below. OPADR is set to zero. All other fields are left unchanged. If OPADR was zero, the procedure is dummy.



5.2.3 MUSIL Standard Input/Output.

Input and output in MUSIL can be done in two different ways:

1) by calling standard input/output procedures like getrec, putrec, open or close.

2) by calling operator communication procedures like opmess, opin/opwait or opstatus.

The MUSIL implementation has been designed to allow normal use of the standard I/O-procedures. The implementation does, however, restrict the use of the procedures in (2), in consequence of the way operator communication is coded in the MUSIL interpreter.

All standard input/output activity ends up with sending messages and waiting for answers. To allow other coroutines in a system to run while one is waiting for answers, the I/O procedures have to be forced to exit to the coroutine central logic, when meeting such a waiting point. The central logic will then reactivate the coroutine when the answer arrives, and normal I/O processing will then proceed. This is done by setting bit 0 (octal value 100000) in the kind of the zone, in addition to the bits designating blocked, positionable etc., in the declaration of the file.

When files are used in connection with multi-incarnation coroutines, or other applications where data containing a file is swapped, the following precautions should be taken.

First, as the buffer area in a file is used by e.g. driver processes which runs asynchronously in respect to the program, the buffers have to occupy a fixed location. This means that they cannot be allocated in the swapped area, but must be situated outside. This is done by allocating one single byte as buffersize (as the compiler does not allow zero), in the declaration and initialize the file description by means of appropriate procedures to describe the actual buffers.

A possible procedure for the purpose is INITZONE, codeprocedure P0155 (see appendix D). It will allocate as well buffers as the socalled share-descriptors (describing buffers) in a separate area, thus decreasing the number of bytes to be swapped. If INITZONE is used, the number of shares should be one (as the compiler does not allow a value of zero).

Second, as the MUSIL compiler only will recognize one file, where actually more are used, the number of message buffers used by the I/O system to send messages, which is allocated will be too small. The user must therefore allocate additional message buffers e.g. by means of CREATEMESSBUFS, codeprocedure P0054 (see appendix D). The number of extra message buffers needed is computed as follows. For each file declaration to be used by a set with many incarnations, find the number of shares, i.e. buffers. Add these numbers and multiply the result by the number of incarnations decreased by one. The result of the multiplication gives the number of extra messages needed for this specific multi-incarnation coroutine.

The procedures which may be used in this way includes:

CLOSE
GETREC
INBLOCK
INCHAR
OPEN
OUTBLOCK
OUTCHAR
OUTTEXT
PUTREC
SETPOSITION
TRANSFER
WAITTRANSFER
WAITZONE

The operator communication procedures make direct send message and wait answer, and cannot be forced to call the coroutine central logic. As input is split into two procedures, OPIN sending a message and OPWAIT waiting for answer, with OPTEST telling when it is convenient to wait, these procedures may be used with a little caution. The remaining procedures, OPMESS and OPSTATUS will, however, delay the whole coroutine system and can only be used where the parallel running of the coroutines is of no importance.

5.3 Utilities.

The procedures described in this section are (with the exception of the delay and pass procedures) somewhat arbitrarily selected from the host of applicable procedures already existing in the code procedure library. A description of these procedures should as a consequence be consulted when actual needs arises (see appendix D for descriptions of some selected procedures). The procedures described below comprises code-procedures to be used when a MUSIL procedure containing a coroutine waiting point may be called (directly or indirectly) from several coroutines a procedure to change the testmask in the coroutine ident field and a procedure to generate internal operations. Utility procedures not included in the present description features array simulation by means of swapped data areas and queue handling.

5.3.1 Reentrancy in Procedures.

```
RESETSTACK(STAKC,AREA,DISP,DEPTH);
SAVELINK   (STACK);
RETURN     (STACK);
```

Declaration:

```
procedure RESETSTACK        (var    STACK  : integer;
                             const  AREA   : string (1);
                             const  DISP   ,
                                    DEPTH  : integer);

codebody P0073;
```

```
procedure SAVELINK          (const  STACK : integer);
codebody P0096;

procedure RETURN            (const  STACK : integer);
codebody P0097;
```

The procedures make use of a stack (see 4.5). These procedures are associated with call of and return from MUSIL procedures with a coroutine waiting point in the body of statements, when the procedure in question may be called, directly or indirectly, by several coroutines at the same time, thus implementing reentrant MUSIL procedures.

When a MUSIL procedure is called, the point of return is stored at the beginning of the procedure bodycode, to be used when the last END in the procedure is encountered. If several coroutines are executing the body at the same time, this will cause all these coroutines to return to the calling point of the latest caller unless savelink and return are used.

The procedures are intended to be used in a way so the call of savelink at the top of a procedure ultimately is followed by a matching call of return. If the calls of savelink and return do not match pairwise, e.g. when using goto to a label outside the procedure, this will lead to an unpredictable flow of control or to stack under - or overflow. If a jump to the main program has to be executed, then a call of resetstack is recommended to reset the stack to empty.

A procedure shall call savelink and use return to terminate execution of the procedure body when the body either containing a coroutine waiting point as waitgeneral, or calls another procedure, which needs to use savelink/return, and it may be called by several coroutines at the same time.

Functional Description:

RESET (STACK,AREA,DISP,DEPTH):

Initializes the body of the stack STACK found in the bytes

```
AREA(DISP) ... AREA(DISP+DEPTH-1)
```

to empty. The variable used as first parameter STACK contains a reference to the body, and this reference is used in the codeprocedures requiering a stack as parameter.

The value of DEPTH should be 6 + maxdepth * 2, where maxdepth is the maximum number of savelink calls exceeding the matching calls of return at any moment, with other words the maximum depth of procedure calls inside procedures. The 3 words are used by the procedures, and the minimum stack size is therefore 8 bytes. The value of maxdepth is in typical applications 3-5.

SAVELINK (STACK):

Note: Savelink must be called as the first statement in the procedure immediately at the first BEGIN.

The point of return is seized and put on top of the stack given as parameter. If the stack overflows, a program break is executed with break code 6.

RETURN (STACK):

The procedure fetches a return point from the stack given as parameter and executes a return jump to this point, in exactly the way the MUSIL interpreter does when encountering the final END statement in a procedure. The codeprocedure return may, however, be called anywhere in the procedure body. If the stack was empty, a stack underflow condition is signalled by means of a program break with break code 6.

5.3.2 Coroutine Descriptor.

CHANGEMASK(COROUT,CONO,MASK,LENGTH);

Declaration:

```
procedure CHANGEMASK          (var   COROUT : string(18);
                               const CONO,
                                     MASK,
                                     LENGTH : integer);
codebody P0079;
```

Functional Description:

The parameters but MASK are identical with the parameters used when calling DEFCOROUT (see 5.4.2) to define this coroutine. The seven least significant bits MASK 9-15 replaces the testmask in the COIDENT bit 1-7 of the coroutine, which are used to control the amount of testoutput generated by the various procedure calls.

5.3.3 Coroutine Delay.

CDELAY(TIME);

Declaration:

```
procedure CDELAY              (const  TIME : integer);
codebody P0080;
```

Functional Description:

Delays the calling coroutine TIME x 20 msec.

TIME	time waited
0	0
1	0-20 ms
2	20-40 ms
255	5,08-5,10 sec.
65535	21 min. 50,7 sec.

The timer has an inbuild inaccuracy of 0-20 ms.
The WAITGEN procedure may be used, too, to delay a coroutine.

5.3.4 Coroutine Pass. PASS; 5.3.4

Declaration:

```
procedure PASS;
codebody P0126;
```

Functional Description:

The procedure is intended to be used as 'breakpoint' in time consuming operations, thus allowing other coroutines to run as if the calling coroutine had entered a waiting point.

5.3.5 Create Internal Operations. CREATEOPS(AREA,OPDESCR.OPADR,NO, SEM); 5.3.5

Declaration:

```
procedure CREATEOPS            (var   AREA   : string(1);
                                 var   OPADR  : integer;
                                 const NO     : integer;
                                 var   SEM    : integer );
codebody  P0XXX;
```

The procedure is used to create a pool of internal operations, (see 4.3) linked to an administration semaphore (see 4.4.2).

Functional Description:

A number of operations given by NO are created and signalled to the general semaphore SEM, using the bytes

AREA(OPADR) ... AREA(OPADR + 2 * (OPSIZE+2) * NO -1)

where OPADR and OPSIZE are fields in the OPDESCR (see 4.3.1). The value of OPADR after the call is the previous value incremented by

2 * (OPSIZE + 2) * NO

The new operations are of type given in OPTYPE, and size OPSIZE + 2.

The data portion of the operations is initialized with the contents of the data area of OPDESCR, as in SIGGEN (see 5.1.2).

OPADR works as running displacement in the string AREA, initial value may be e.g. zero. OPSIZE determines the number of data words in the operation.

5.4 Initialization.

The procedures in this section are used to initialize various areas in the process descriptor and to define coroutine descriptors and initialize them.

5.4.1 Initialize System. INITCOSYS(AREA,SYSCO,IDENT,OPS,CSBUFS);

Declaration:

```
procedure INITCOSYS          (var   AREA     : string(1);
                              var   SYSCO    : string(18);
                              const IDENT    : integer;
                              const OPS      : integer;
                              const CSBUFS   : integer);
codebody P0088;
```

Functional Description:

The procedure initializes coroutine system variables in the process descriptor and initializes and starts the coroutine SYSCO with the identification given in IDENT, running as single coroutine active in the system. For the contents of IDENT see 5.4.2. The different queues in the process descriptor active queue, answer queue and delay queue are set to empty. The coroutine starts executing the code following the call. Any additional coroutines may be created by DEFCOROUT (see 5.4.2), and will be started when SYSCO executes its first wait or pass. The SYSCO is intended as a special control coroutine, for example acting as message receiver and distributor.

The procedure further initializes the system area given in parameter AREA, the size of which is at least

sysareasize = 40 + 26 * OPS   bytes.

The value of the parameter OPS is the number of system operations to be created. These operations are 13 words long and are used

1) by CSENDMESSAGE to send messages,
2) to signal incoming messages if WAITGEN is used to accept messages sent from other processes.

CSBUFS is the maximal number of operations used by CSENDMESSAGE calls at the same time. The program is breaked with cause -3 in case of lack of operations. The remaining number of system operations OPS - CSBUFS (if > 0) are used to signal incoming messages, and shall be set to the expected number of received messages in the system, which have not yet been answered. As well CSBUFS as this number may be zero, if the corresponding pool is superfluous.

The system uses 20 words for a testrecord and some variables managing the system operations pool.

5.4.2 Initialize Coroutine.   DEFCOROUT(COROUT,NO,IDENT,LENGTH);

Declaration:

```
procedure DEFCOROUT          (var    COROUT    : string(18);
                              const NO          : integer;
                              const IDENT       : integer;
                              const LENGTH      : integer);

codebody P0089;
```

The procedure is used to initialize the system part of a coroutine descriptor and to start the coroutine at an appropriate MUSIL statement. The coroutine may as well be an incarnation of a multi-incarnation coroutine, (where the incarnations execute the same reentrant code and use identical data structures), as a single-standing coroutine. See section 4.2 for a description of this concept. If multi-incarnation coroutines are used, SETUSEREXIT (section 5.4.3) should be employed.

Functional Description:

Note that the procedure call must always be followed by a GOTO statement:

```
DEFCOROUT(READER,6,8'077420,RLENGTH);
GOTO 1105;
```

The procedure will return to the statement after the GOTO, thus skipping it. The starting point of the coroutine defined will be the statement with label 1105.

The procedure defines a coroutine COROUT and initializes the system part with ident field IDENT, and starts it by queuing it into the active queue. The integer IDENT consists of three fields:

Bit 0	: priority	0 low priority 1 high priority
Bit 1-7	: testmask	testoutput is divided into seven classes, see chapter 6. A one bit in position 1-7 tells that testoutput in the corresponding class is wanted. The testmask may later be changed by a call of CHANGEMASK (see section 5.3.2).
Bit 8-15	: identification	the value is used to distinguish between coroutines, e.g. when testoutput is generated. A coroutine system will work with the same identtification for different coroutines. It is, however, recommended that coroutines have unique idents.

A value octal 177777 (all ones) for IDENT is reserved for system use and cannot be used. It will be changed to octal 177776 by the procedure.

The parameter NO determines whether the coroutine is single or an incarnation of a multi-incarnation coroutine.

NO = zero : The coroutine is a single coroutine. The actual coroutine descriptor is situated in COROUT, which may be followed by a number of user variables. The parameter LENGTH is not used.

NO > 0 : The coroutine is incarnation with number NO. All variables, system and user defined, are placed in the area following the declaration of the variables used by the coroutines, see section 4.2, and COROUT acts as a head with a description of the whole set of incarnations. The parameter LENGTH gives the length of the user area in words. The user area for this incarnation is initialized with the contents of the header area.

5.4.3 Multi-incarnation Coroutines: SETUSEREXIT;

Declaration:

```
procedure SETUSEREXIT;
codebody P0090;
```

Functional Description:

The procedure manages swapping of the user part of a coroutine descriptor necessary for multi-incarnation coroutines (see section 4.2). The procedure may, however, be used in any coroutine system whether using incarnations feature or not, without bad effects.

The procedure is implemented using the USER DEFINED EXIT facility in the extended coroutine monitor, which makes it possible for a userto execute some action immediately before the central logic transfers control to an activated coroutine selected as current. It is possible to implement other codeprocedures, which use the facility depending on the actions wanted, but this precludes the use of SETUSEREXIT.

The procedure should be called once in the program, and it is recommended to do this immediately after the call of INITCOSYS (section 5.4.1). The call initializes the CUDEX field of the process descriptor, which defines a user action to point to the code, which manages the multi-incarnation coroutines. This code will then be called immediately before any coroutine execution. The code determines whether the newly activated coroutine is a single coroutine or some incarnation.

When the latter is the case, the code determines if swapping of data is necessary and saves the contents of the old incarnation and loads the new if this is true.

6. TESTOUTPUT FACILITIES.

In order to aid the user in debugging, some facilities for generating testrecords have been built into the extended coroutine monitor. The testrecords contains the time, identification of origin and some data and are produced

(1) when coroutine functions are called,
(2) at exit to an activated coroutine,
(3) by calling a testoutput procedure.

The testoutput is processed by a separate program, and the amount of testoutput may be dynamically controlled. Figure 7 shows the configuration.

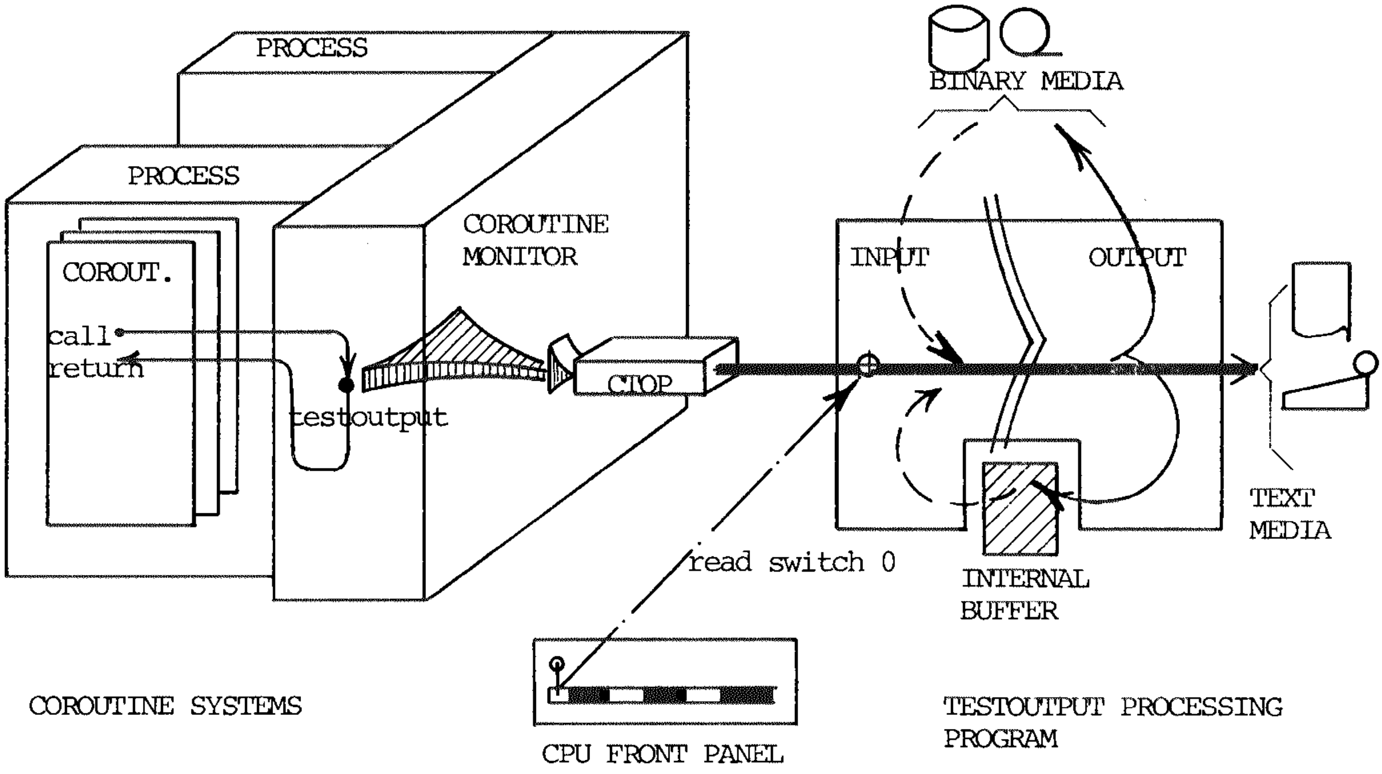


Figure 7. Testoutput.

Each process containing a number of coroutines. The testoutput routine in the coroutine monitor is called, and the value of CTOP in page zero determines whether the testoutput processing program is processing testrecords. The program is described in a separate manual and allows testoutput to be output to a text medium like printer, a binary medium like magnetic tape, or written into an internal, cyclic buffer. The program may in addition retrieve binary and internal testrecords for printing.

The amount of testdata produced is controlled in several ways:

(1) The inbuilt coroutine testoutput is divided into seven classes. The IDENT field in a coroutine descriptor contains a mask, which selects which classes are to be output (see section 5.4.2).

(2) Testoutput is only generated when the location CTOP in page zero contains an address. This location is reset to zero as the coroutine monitor is loaded, but is set to a proper value by the testoutput program.

(3) The testoutput program reads switch 0 on the RC3600 CPU front panel. Testoutput is processed when this switch is one.

(4) The user may produce varying amounts of testdata by means of TESTPOINT, which is described below.

Output generated by the coroutine monitor procedures contains absolute storage addresses, which are not available to a MUSIL programmer, except in connection with storage dumps. It is therefore recommended mainly to rely on TESTPOINT.

6.1 User Produced Output. TESTPOINT(KIND,DATA); 6.1

Declaration:

```
procedure TESTPOINT            (const  KIND : integer;
                                var    DATA : integer);

codebody P0072;
```

Functional Description:

The procedure generates one testrecord. The parameter KIND has four fields:

KIND bit 0 : LONG — 0 No data field is present.
1 Data are present in DATA and the variable declared after it. 7 or 11 words are output.

bit 1-7: CLASS — Determines the class of the testrecord. One of the bits is normally set, giving 7 classes. The testrecord is not processed it the IDENT for the coroutine has a zero in the mask for that class. The classes are shown in section 6.2.

bit 8 : REVERSE — Indicates whether the words following or preceding the second parameter are to be output as data. The value 0 indicates following words to be used, and is the normal value. The value 1 indicates preceeding words to be used, and in this case DATA and the word declared immediately before it will be skipped.

bit 9-15: FUNCTION Identifies the coroutine procedure or user origin. The values 1-13 are used to identify system functions and will be printed as corresponding names (see 6.2). The values 0 and 14-127 (decimal) are printed as a three-digit decimal number.

The procedure TESTPOINT is intended to output key variables in the program. As these variables very often are integers, the parameter type has been selected to be an integer. Depending on KIND, no, 7 or 11 words are output (for the format of a testrecord, see the appropriate manual):

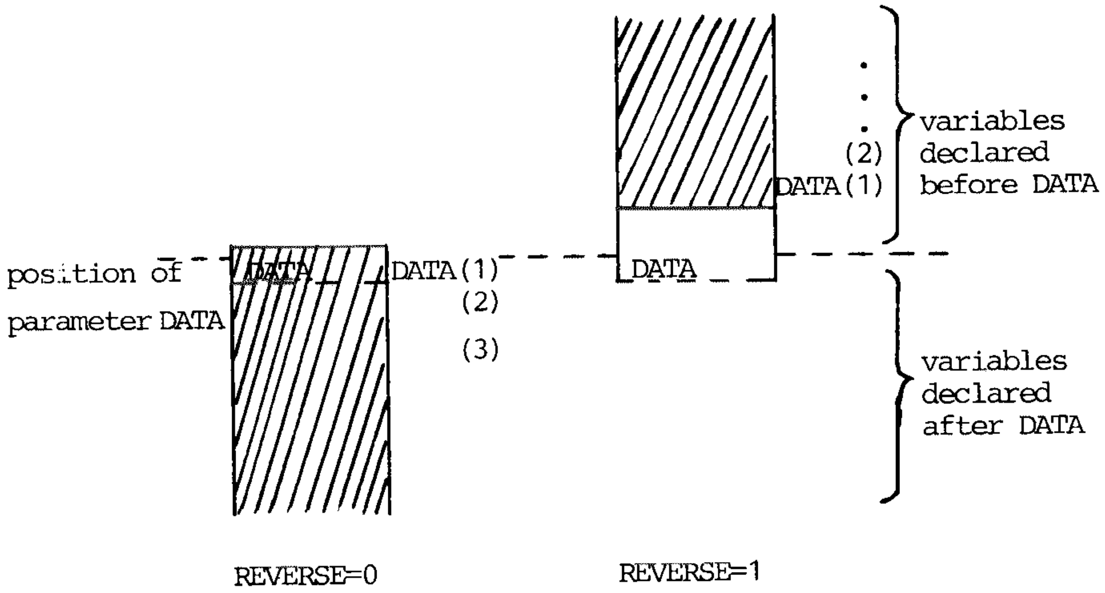
KIND		TOPT1-TOPT7	TAC0	TAC1	TAC2	TAC3
0	9-15					
0	>13 or 0	not output	(note1)	(note2)	undef.	undef.
1	>13 or 0	DATA(1)-DATA(7)	(note1)	(note2)	undef.	undef.
1	10	DATA(1)-DATA(7)	DATA(8)	-(9)	-(10)	DATA(11)

note 1: TAC0 The image of register 0 contains the number of times the system operations pool (size-13 operations) has been empty when a wait general was executed.

note 2: TAC1 The image of register 1 contains the position in the event queue of the next message expected, starting with one; the value is thus one greater than the number of message currently received but not answered.

The function codes in the interval 1-13 should be avoided not to confuse the reader of the testoutput.

The words DATA(1) ... depends on the REVERSE bit:



6.2 Built-in Testoutput.

The different procedures will normally generate testoutput. The following table contains a summary of what is generated by which procedure:

Procedure		FUNC		OUTPUT CLASS BIT	DATA
CDELAY	*)	8	DELAY	6	No!
CHANGEIDENT		-		-	-
CREATEOPS		11	SIGGE	1	Yes, as SIGGEN.
CSENDMESSAGE		13	CSEND	2	No.
	(and	11	SIGGE	1	Yes, as SIGGEN).
DEFCOROUT		-		-	-
INITCOSYS		-		-	-
INITGENSEM		-		-	-
I/O SYSTEM	*)	5	CWANS	7	No.
PASS	*)	6	PASS	6	No.
RELEASEANSWER		-		-	-
RESETSTACK		-		-	-
RETURN		-		-	-
RETURNANSWER	**)	2	SIGNA	2	No.
SAVELINK		-		-	-
SETUSEREXIT		-		-	-
SIGGEN		11	SIGGE	1	Yes, last 6 words and type of the operation.
SIGNAL		2	SIGNA	2	No.
TESTPOINT		Any		Any, prefer.4	Any - see section 6.1.
WAITSEM	*)	4	WAITS	3	No.
WAITGEN	*)**)	12	WAITG	3	No.

The procedures marked with *) contain a waiting point, and generates an EXIT testrecord, class bit 5, when reactivated. The procedures marked with **) handle incoming messages and generates testrecords in the following way:

WAITGEN: If the message type bit is not set in the mask, a simple WAITG testrecord appears. Otherwise two or three testrecords appear, first a WAITS on an anonymous semaphore to ensure a system operation is available, and then a WAITG. If the resulting operation is not a message, a third testrecord is generated by a signal (SIGNA) to the anonymous semaphore to release the system operation which was not used.

RETURNANSWER: Generates a SIGNA testrecord when the system operation is returned to the pool, by a signal to the anonymous semaphore.

7. CODING EXAMPLES.

The following three examples illustrates the use of procedures and data structures. As it is difficult to give simple examples of coroutine systems, only one is a total program, performing a simplified dataconcentrating task. The other examples illustrates message and answer handling.

7.1 Message Distributing.

This example shows a piece of code executed by message-distributing coroutines. Incoming messages are examined for a streamnumber in mess0 bits 0-7, and depending on the value signalled to the proper coroutine after changing the type to an internal type, as the message type only may be waited for by the distributing coroutine. The semaphores used are found in an array QUEUESEM, simulated by means of the codeprocedures LOAD and STORE (see appendix D). The variable declaration part includes:

```
CONST
     maxstreams   = ... ,  ! max number of streams used !
     maxextend    = ... ,  ! maxstreams * 2             !
     semareasize  = ... ,  ! maxstreams * 10            !
     ....
     syssize      = ... ,  ! 40 + 26 * no of system operations !
VAR
! two arrays containing variables known to sysco and the!
! processing coroutines.  In LOAD/STORE format          !

reserver: INGEGER; resarray : STRING (maxextend);
queuesem: INTEGER; qsarray  : STRING (maxextend);
....
```

```
! SYSCO descriptor and variables                          !

sysco    : STRING(18);      !coroutine descriptor !
sysdescr : RECORD
            opadr, optype, opsize : INTEGER;
            sender, receiver      : INTEGER;
            mess0, mess1,
            mess2, mess3          : INTEGER;
            special               : INTEGER
         end;

eventsem :  INTEGER;
streamno, infinite : INTEGER;   ! NOTE1 !
i, p               : INTEGER;
evarea   :  STRING(10);
...

! bodies of queuesemaphores                               !
semarea  :  STRING(semareasize);
sysarea  :  STRING(syssize);
```

The initialization includes:

```
BEGIN
         initcosys (sysarea, sysco, ..., ..., ...,);
         infinite:= -1                          ! NOTE 1!
         ...

         ! initialize semaphores          !
         initgensem (eventsem, evarea, 0);

         i:= 0; p:= 0;
         WHILE i < maxstreams DO
         BEGIN
            initgensem (queuesem, semarea, p);
            p:= p + 10;
            store (queuesem, i);
            ...

            reserver:= 0;  store(reserver, i);
            ...
            i:= i + 1
        END;
```

The code of the message distributor:

```
100:  sysdescr. opsize:= 7;          ! NOTE 2 !

      waitgen (eventsem, 8'000004, sysdescr.opadr, infinite);
      testpoint (8'104020, sysdescr.sender);    ! NOTE 3 !

      streamno:= sysdescr. mess0  SHIFT(-8);
      IF streamno >= maxstreams THEN
      BEGIN    ! reject and wait for the next !

105:        sysdescr.mess0:= 8'001000;        ! NOTE 4 !
            sysdescr.mess1:= 0        ;
            returnanswer (sysdescr.opadr);
            goto 100
      END;

      ! now streamno is allowed - check if stream reserved !
      load (reserver, streamno);

      IF reserver <> 0 THEN
      IF reserver <> sysdescr.sender THEN GOTO 105;

      ! the stream is not reserved by another              !
      ! change the type of the operation to an internal    !
      ! value:                                             !
      !          CONTROL    octal 400                      !
      !          INPUT      octal 200                      !
      !          OUTPUT     octal 100                      !

      IF sysdescr.mess0 extract 2 = 2'01 THEN sysdescr.optype:=8'200 ELSE
      IF sysdescr.mess0 extract 2 = 2'11 THEN sysdescr.optype:=8'100 ELSE
      sysdescr.optype:= 8'400;

      load (queuesem, streamno);
      siggen (queuesem, sysdescr.opadr);

      goto 100;
```

Notes:

(1) The delay parameter to waitgen shall be a variable. The value of infinite will, however, not be changed, and may be used by all coroutines in the system.

(2) The last 7 words are moved, but only mess0, mess1 and sender are used.

(3) This call of testpoint will produce a record which eventually will be printed like this:

```
proc cor func time AC0    AC1      AC2   AC3
...  ... 016  ...  ...    ...      ...   ...
                   sender receiver mess0 mess1 mess2 mess3 special
```

(4) The message is answered with mess0 and mess1 changed to (rejected, 0) and mess2, mess3 unchanged.

7.2 Sending Messages.

This example makes use of an invented set of communication rules between two processes, which will not normally be found in a real system to illustrate sending of messages and waiting for the answers. The receiver of the messages will return a message when receiving a special control message.

Declarations:

```
CONST
        procname= 'HEURE',
        xcontrol= 8'000000,
        input   = 8'000001,
        ...
        syssize = 92,          ! 40 + 26 * messpool    !
        messpool=  2,
        cspool  =  2;
VAR
infinite : INTEGER;                                    ! NOTE 1 !

! coroutine declaration            !

messco   : STRING(18);
dop      : RECORD                                      ! NOTE 2 !
                  opadr, type, size : INTEGER;
                  mess0, mess1,
                  mess2, mess3      : INTEGER;
                  special           : INTEGER
           END;

dsem     : INTEGER;

waittime, address : INTEGER;

dsarea   : STRING(10);
buffer   : STRING(80);
...
sysarea  : STRING(syssize);
```

Initializations:

```
BEGIN
      initcosys (sysarea, ..., ..., messpool, cspool);
      infinite:= -1;                                         ! NOTE 1 !
      ! initialize variables !

      initgensem (dsem, dsarea, 0);
      takeaddress (buffer, address);                         ! NOTE 3 !

       defcorout(messco, 0, 8'004001, 0);                    ! NOTE 4 !
      goto 1000;
```

The code sending messages and waiting for answers:

```
1000:                                                        ! NOTE 4 !
      ....
       dop.size:= 5;                                         ! NOTE 2 !
      dop.mess0:= input;
      dop.mess1:= 80;
      dop.mess2:= address;

      csendmessage (procname, dsem, dop.opadr);
      waittime:= 500;                                        ! NOTE 1 !
      waitgen (dsem, 8'000003, dop.opadr, waittime);         ! NOTE 6 !
                                                             ! NOTE 6 !
      IF dop.type = 8'000001 THEN
      BEGIN  ! take it home        !

            dop.mess0:= xcontrol;
            csendmessage (procname, dsem, dop.opadr);
            waitgen (dsem, 8'000002, dop.opadr, infinite);   ! NOTE 6 !
            releaseanswer (dop.opadr);                       ! NOTE 5 !
            dop.size:= 0;   ! ignore answer to regret operation!
            waitgen (dsem, 8'000002, dop.opadr, infinite);
      END;                                                   ! NOTE 2 !
      releaseanswer (dop.opadr);                             ! NOTE 5 !

      ! now use the answer !
```

Notes:

(1) The parameter to waitgen specifying delay has to be a variable. The value of infinite is, however, not changed and may be shared by several coroutines. The variable waittime will be decremented from 500 * 0.02 = 10 seconds, depending on when the answer arrives.

(2) The data part of the operation descriptor is 5 words long (compare example in 7.1). When the answer is of no importance, it is skipped by setting size to 0, leaving the old contents.

(3) See appendix D for description of takeaddress.

(4) The coroutine messco is a single coroutine with identification number 1, allowing testoutput controlled by bit 4. The starting point is the label 1000.

(5) Answers to messages sent by CSENDMESSAGE are released to return the system operation to the free pool.

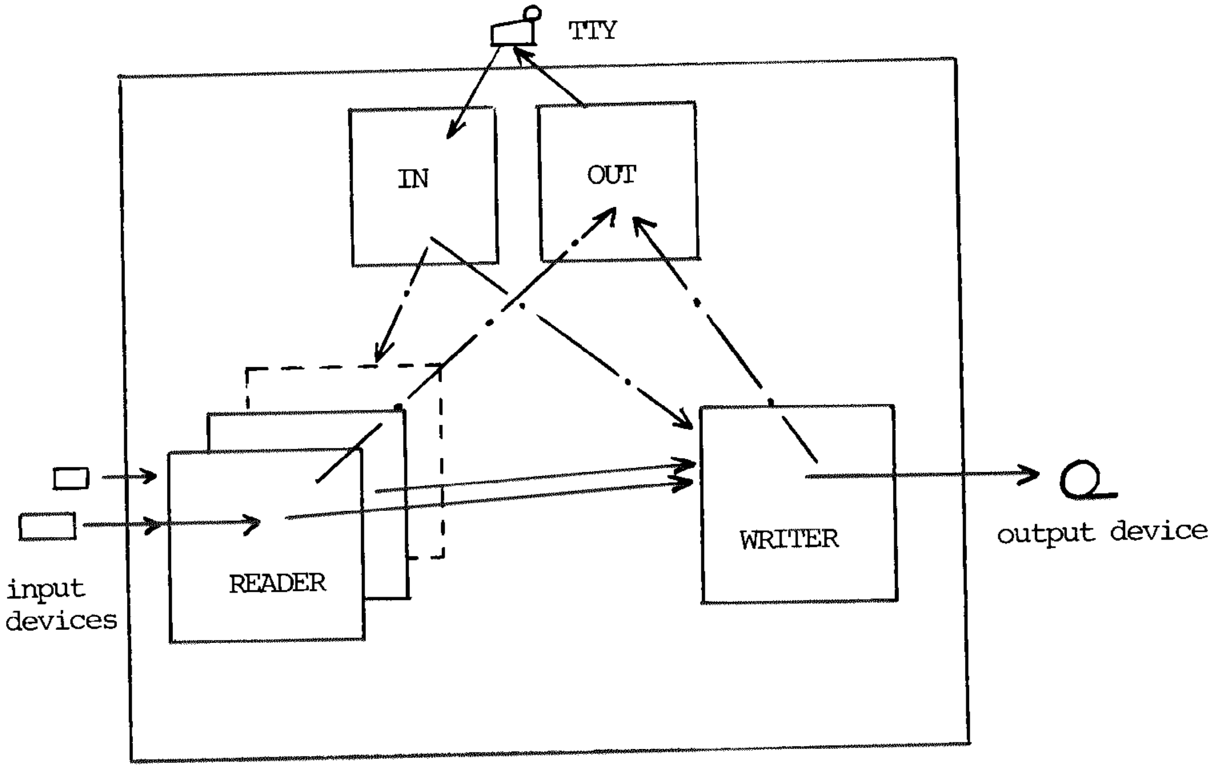
(6) The eventmasks used have the following significance:

8'000003	TIMER + ANSWER	(delay = 10 seconds)
8'000001	TIMER	
8'000002	ANSWER	(delay = infinite)

The answer will be forced home after 10 seconds, if it has not arrived before.

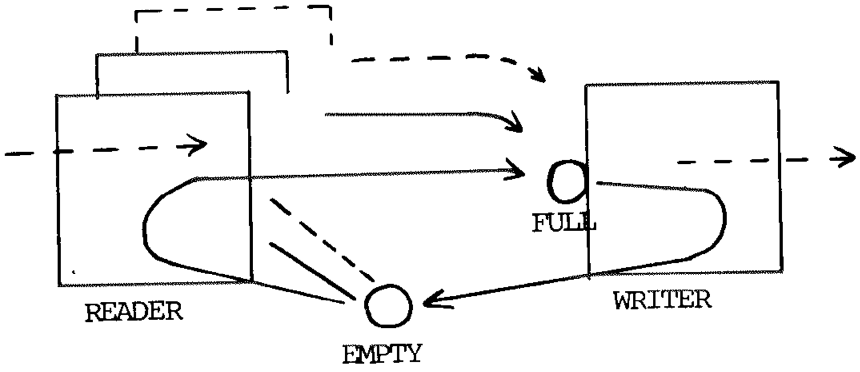
7.3 Data Concentration Example.

The example is a program consisting of a number of identical coroutines (here two) reading data from a number of devices and sending it to a writer coroutine which outputs these data in the order of arrival. Two additional coroutines take care of operator communication:



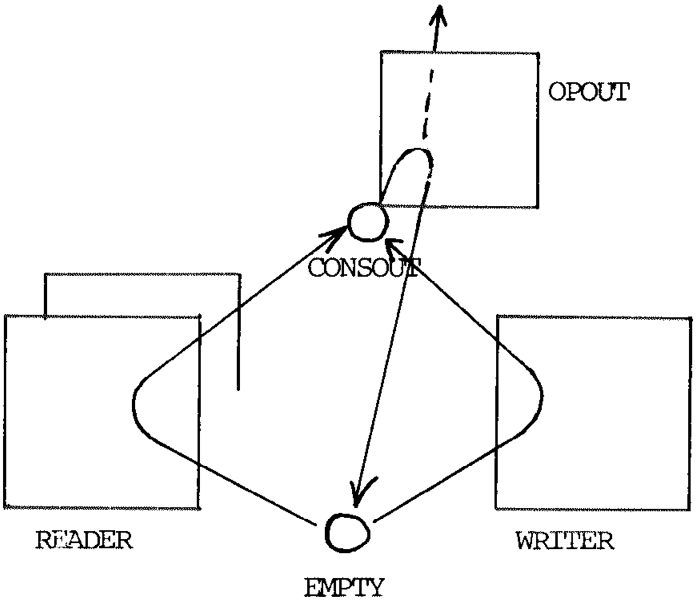
The coroutines use the I/O system to input and output data. To communicate with each other they make use of general semaphores:

Dataflow



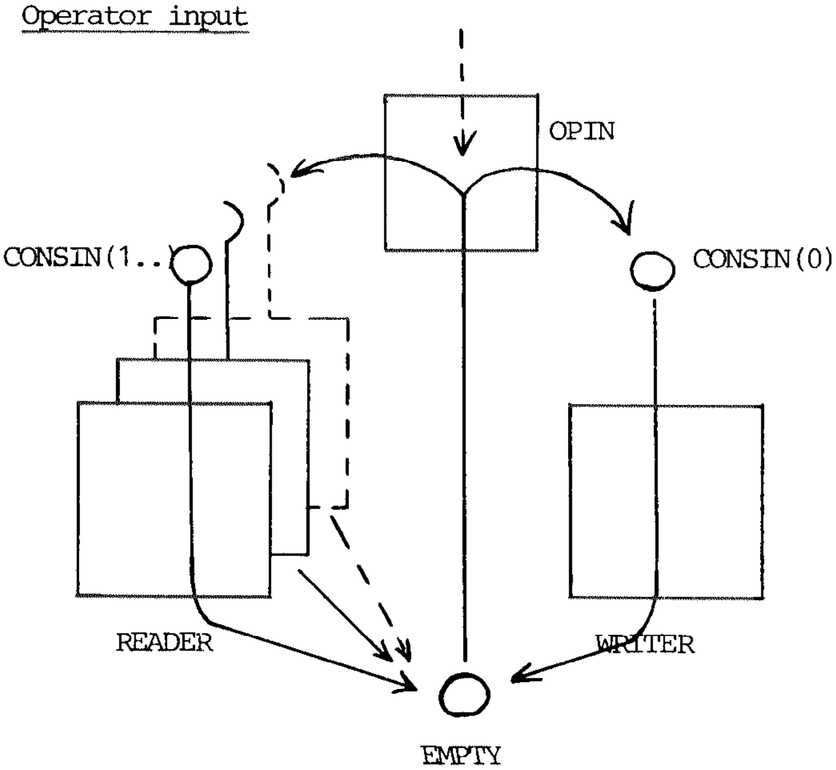
(maintains queue of empty buffers)

Operator Output



An empty operation of type CONSTYPE is fetched and sent to the output coroutine by means of the semaphore CONSOUT.

Operator input



An empty operation of type CONSTYPE is fetched and sent to the relevant READER/WRITER by means of the array of semaphores found in CONSIN.

```
| PROGRAM EXAMPLE / DATA CONCENTRATOR                    |

| CONSTANT SECTION |
 CONST

| configuration dependent constants |

maxreader           = 2,   |                                     |
semtablesize        = 6,   | 2x(maxreader+1)                     |
varlength           = 49,  | length of variable area for readers |
stasize             = 20,  | maxreader*stacksize                 |
semasize            = 30,  | semsize*(maxreader+1)               |
fbsize              = 188, | maxreader*(80+14)                   |
messpoolsize        = 14,  | (maxreader-1)*14                    |
syssize             = 40,  | 40+26xno of system operations       |
oppoolsize          = 3,   |                                     |
opareasize          = 36,  | oppoolsize*2*(conssize+2)           |
datpool             = 3,   |                                     |
datopsize           = 24,  | datpool*2*(dsize+2)                 |
intsize             = 246, | 3*writerlength                      |
| other reader constants   |
readersize          = 232  | 2xmaxreader*(varlength+9)           |
readmode            = 25,
maxsize             = 80,
rdnames             = " CDR<0><0><0>RDP<0><0><0>",
| writer constants |

writerbuf           = 410,| 5 record/block |
writerlength        = 82,
writemode           = 3,

| states |

neutral             = 0,
running             = 1,

| operating types used by e.g. waitgen |

timertype           = 8'000001,
constype            = 8'000100,  conssize = 4,
datatype            = 8'000200,  dsize    = 2,
| operator input constants |

cstart = "start", cstartlen = 6, startcom = 1,
cstop  = "stop",  cstoplen  = 5, stopcom  = 2,


| operator output/error recovery |

operatorintervention = 8'177777,
stopbits      = 8'001424,
delaybits     = 8'160000,
fatalerror    = 8'001004,

retryinterval = 10,    | 0.2 seconds |

headlength    = 7,
errtext       = "error",
finistext     = "finis", textl = 7,
nl = 10,  sp = 32,

| other constants |

coident      = 8'077400,
semsize      = 10,
stacksize    = 10;  ! 6 + 2 x nested calls of savelink !
```

```
| TYPE DECLARATIONS          |
 TYPE
coroutine = string(18);

gensemaphore = string(semsize);

opdescriptor =
        record
           opadr,
           optype,
           opsize  :  integer;

           | constype : |
           status : integer;
           name   : string(6);
           command: integer from 7;
           | datatype:|
           address : integer from 7;
           length  : integer from 9
        end;
```

```
VAR

infinite : integer;
empty, full, consout : integer;
emptysem, fullsem, conssem : gensemaphore;


| system coroutine - operator input    |

sysco : coroutine;
ttlength, clen, no, command : integer;
ciop : opdescriptor;
ttin : file "TTY", 8'100001, 1, 80
       of string(80);

| operator output                      |

opout : coroutine;
coop : opdescriptor;
ii, char : integer;
txt : string(7);
ttout : file "TTY", 8'100001, 1, 80, ub
        of string(80);

| writer                               |

writeco : coroutine;
wop, erop : opdescriptor;
wrtstate, wtime : integer;
writer : file "MT0", 8'100016, 2, writerbuf, fb;
         giveup writererror, 8'161777
         of string(writerlength);
```

```
| readers                          |

readerhead : coroutine;
commsem : integer;
rop  :opdescriptor;
rdstate, rdno, rdlength, rtime, endmark,
stack, result       : integer
reader : file "XXX", 8'100001, 1, 1;
         giveup readererror, 8'161777
         of string(1);

covars : string(readersize);

| common variables and system areas |

consin : integer; consvars : string(semtablesize);
sysarea       : string(syssize);
stackarea     : string(stasize);
semarea       : string(semsize);
opops         : string(opareasize);
dataops       : string(datopsize);
localbufs     : string(intsize);
filebufs      : string(fbsize);
messbufarea   : string(messpoolsize);
```

```
| CODEPROCEDURE DECLARATIONS |

procedure cdelay(const time : integer);
codebody p0080;

procedure createops(var   area          : string(1);
                    var   opadr         : integer;
                    const no            : integer;
                    var   sem           : integer);
codebody p0XXX;

procedure defcorout(var   corout        : string(18);
                    const no,
                          ident,
                          length        : integer);
codebody p0089;

procedure initcosys(var   area          : string(1);
                    var   sysco         : string(18);
                    const ident,
                          ops,
                          csbufs        : integer);
codebody p0088;

procedure initgensem(var  sem           : integer;
                     var  semarea       : string(1);
                     const disp         : integer;
codebody p0091;

procedure resetstack(var  stack         : integer;
                     const area         : string(1);
                     const disp,
                           depth        : integer);
codebody p0073;

procedure return(const stack  :  integer);
codebody p0097;
```

| CODEPROCEDURE DECLARATIONS - CONTINUED |

```
procedure savelink(const stack : integer);
codebody p0096;

procedure setuserexit;
codebody p0090;

procedure siggen(var sem     : integer;
                 var opadr   : integer);
codebody p0093;

procedure waitgen(var sem    : integer;
                  const mask : integer;
                  var  opadr,
                       delay : integer);
codebody p0092;

procedure binoct(const no    : integer;
                 var   str   : string(6);
codebody p0087;

procedure createmessbufs(var   bufarea : string(1);
                         const length  : integer);
codebody p0054;

procedure fill(const bytevalue : integer;
               const tostr     : string(1);
               const toindx,
                     count     : integer);
codebody;

procedure initzone(file  z;
                   const shares,
                         length,
                         area   : integer);
codebody p0155;
```

| CODEPROCEDURE DECLARATIONS - FINIS |

```
procedure invalue(const value
                        toaddr,
                        atype    : integer);
codebody  p0121;

procedure load(var   base  : integer;
               const index : integer);
codebody;

procedure movin(const fromstr    : string(1);
                const fromindx,
                      toaddr,
                      count      : integer);
codebody;

procedure movout(const fromaddr : integer;
                 var   tostr    : string(1);
                 const toindx,
                       count    : integer);
codebody;

procedure store(var                : integer;
                const index        : integer);
codebody;

procedure takeaddress(var   strvar : string(1);
                      var   addr   : integer);
codebody;
```

```
procedure readererror;
begin | giveupprocedure for readers |
   savelink(stack);
   rtime:= 0;
   if reader.z0 and operatorintervention <> 0 then
   rtime:= infinite;
   waitgen(empty, constype, rop.opadr, rtime);
   if rop.optype <> timertype then
   begin  | send status to operator |
        rop.status:= reader.z0;
        rop.name  := reader.zname;
        siggen(consout, rop.opadr);
   end;

   if reader.z0 and stopbits <> 0 then
   if reader.zrem = 0 then
   begin  | stop by signalling device end |
        reader.zrem:= 1;  | prevent inblock loop |
        endmark     := 1;
   end;

   if reader.z0 and delaybits <> 0 then cdelay(retryinterval);

   return(stack)
end;

procedure testoperator;
begin | test for operator input to reader |
      | result                            |
      | 0 - no message, <>0 message       |
   savelink(stack);
   rtime:= 0;  rop.opsize:= conssize;
   waitgen(commsem, constrype, rop.opadr, rtime);
   result:= 0;
   if rop.optype <> timertype then result:= rop.command;
   return(stack)
end;
```

```
procedure inoperator;
begin | acts on a result from testoperator |
   savelink(stack);
   testoperator;
   if result <> 0 then
   begin | message present |
        if result = startcom then rdstate:= running else
        if result = stopcom  then rdstate:= neutral;
   end;
   return(stack)
end;

procedure writererror;
begin | giveup procedure for writer|
   wtime:= 0;
   if writer.z0 and operatorintervention <> 0 then
   wtime:= infinite;
   waitgen(empty, constype, erop.opadr, wtime);
   if erop.optype <> timertype then
   begin | send status to operator |
        erop.status:= writer.z0;
        erop.name   := writer.zname;
        siggen(consout, erop.opadr);
   end;

   if writer.z0 and fatalerror <> 0 then
   begin | wait until operator acts|
        load(consin, 0);
        waitgen(consin, constype, erop.opadr, infinite);
        | regardless of contents, proceed|
        siggen(empty, erop.opadr);
   end;
   cdelay(retryinterval);
   repeatshare(writer);
end;
```

```
| INITIALIZATION OF VARIABLES AND START OF COROUTINES |

begin

      initcoosys(sysarea, sysco, coident, 0, 0);
      setuserexit;

      infinite:= -1;
      initgensem(empty, emptysem, 0);
      initgensem(full, fullsem,   0);
      initgensem(consout, conssem,0);
      createmessbufs(messbufarea, messpoolsize);

      | operator output coroutine |

      defcorout(opout, 0, coident+127, 0);
      goto 3000;| xxxx |

      coop.opsize:= conssize; coop.optype:= constype;
      coop.opadr := 0;
      createops(opops, coop.opadr, oppoolsize, empty);

      | writer coroutine           |

      wrtstate:= neutral;
      defcorout(writeco, 0, coident+126, 0);
      goto 2000;| xxxx |

      ciop.opadr:= 0; ciop.optype:= datatype;
      ciop.opsize:= dsize;
      takeaddress(localbufs,ciop.address);
      ii:= 1;
      repeat
        createops(dataops, ciop.opadr, 1, empty);
        ciop.address:= ciop.address+writerlength;
        ii:= ii+1
      until ii>datpool;
      initgensem(consin, semarea, 0); store(consin, 0);
```

```
| INITIALIZATION CONTINUED                 |

          | readers                  |

          no:= 1;  ii:= 0;
          repeat
             rdstate:= neutral; rdno:= no;
             resetstack(stack, stackarea, ii ,  stacksize);
             ii:= ii+stacksize;
             initgensem(consin, semarea, rdno*semsize);
             commsem:= consin; store(consin, rdno);
             defcorout(readerhead, rdno, coident+rdno, varlength);
             goto 1000; | xxxx |

             no:= no+1
          until no>maxreader;
```

```
| OPERATOR INPUT                             |

     open(ttin, 1);

400: getrec(ttin, ttlength);
     if ttlength>1 then
     begin | interpret command |
       if ttin↑ = cstart then
       begin
          clen:= cstartlen;
          command:= startcom;

415:      ttlength:= ttlength-clen;
          if ttlength>1 then
          begin | parameter present |
           move(ttin↑, clen, ttin↑, 0, ttlength);
           decbin(ttin↑, no);
           if no <= maxreader then
           begin | send command to the coroutine |
              load(consin, no);
              waitgen(empty, constype, ciop.opadr, infinite);
              ciop.command:= command;
              siggen(consin, ciop.opadr);
           end
       end
     end | start |
     else
     if ttin↑ = cstop then
     begin
       clen:= cstoplen;
       command:= stopcom;
       goto 415;
     end;
   end;
   goto 400;
```

```
     | READER COROUTINE STATEMENTS - REENTRANT     |

     1000:   while rdstate <> running do inoperator;


             | started - begin processing |
             takeaddress(filebufs, ii);
             initzone(reader, 1, 80, ii+94x(rdno-1));
             move(rdnames, (rdno-1)x6, reader.zname, 0, 6);
             open(reader, readmode);
             endmark:= 0;

             repeat
               getrec(reader, rdlength);
               if endmark = 0 then
               begin
                  rop.opsize:= dsize;
                  waitgen(empty, datatype, rop.opadr, infinite);

                  if rdlength>maxsize then rdlength:= maxsize;
                  movin(reader↑, 0, rop.address+2, rdlength);
                  invalue(8'60+rdno, rop.address,1); | identify reader |
                  invalue(sp,        rop.address+1,1);
                  rop.length:= rdlength+2;
                  siggen(full, rop.opadr);
               end
               else rdstate:= neutral;
               inoperator
             until rdstate<>running;

             close(reader, 1);

             rop.opsize:= conssize;
             waitgen(empty, constype, rop.opadr, infinite);
             rop.status:= 0;       | FINIS message  |
             rop.name   := reader.zname;
             siggen(consout, rop.opadr);| finis message to operator |

             goto 1000;
```

```
| WRITER COROUTINE                                     |


2000:   open(writer, writemode);
        setposition(writer, 1, 1);
        wrtstate:= running;

        repeat
           wop.opsize:= dsize;
           waitgen(full, datatype, wop.opadr, infinite);

        | data arrived - output it |

           putrec(writer, writerlength);
           fill(sp, writer↑, 0, writerlength);
           movout(wop.address, writer↑, 0, wop.length);

           siggen(empty, wop.opadr)
        until wrtstate<>running;

        close(writer, 1);

        wop.opsize:= conssize;
        waitgen(empty, constype, wop.opadr, infinite);
        wop.status:= 0; | FINIS |
        wop.name   := writer.zname;
        siggen(consout, wop.opadr); | finis message to operator |

        goto 2000;
```

```
| OPERATOR OUTPUT COROUTINE          |

3000:   open(ttout, 3);

3001:   coop.opsize:= conssize;
        waitgen(consout, constype, coop.opadr, infinite);

         | message arrived - output it  |

        putrec(ttout, headlength);
        fill(sp, ttout↑,0, headlength);
        ii:= 0;
        repeat
          move(coop.name, ii, coop.name, 0, 1);
          char:= byte coop.name;
          if char<>0 then
          begin
              insert(char, ttout↑, ii);
          end;
          ii:= ii+1
        until ii>=5;

        putrec(ttout, textl);
        if coop.status<>0 then
        begin | message about error |
          ttout↑:= errortext;
          putrec(ttout, 7);
          insert(sp, ttout↑, 0);
          binoct(coop.status, txt);
          move(txt, 0, ttout↑, 1, 6);
        end else ttout↑:= finistext;

        putrec(ttout, 1);
        insert(n1, ttout↑, 0);

        outblock(ttout);
        siggen(empty, coop.opadr);

        goto 3001;

end  | PROGRAM - DATA CONCENTRATOR |
```

8. COMPILING AND RUNNING.

A MUSIL compiler with version number 4, or compatible, should be used to compile programs using the facilities of the extended coroutine monitor CM002 or compatible.

The program shall be compiled with the modification parameter set to C:

```
..
MODIF C
..
```

When loading a coroutine system, the coroutine monitor should be loaded before any program using it, otherwise the program will jump to an undefined location (eg. zero) when the first coroutine function is invoked.

A program may be breaked with one of the following codes in addition to the system codes. Care should be taken if restarting after a break.

code	cause	explanation
-3	CSENDMESSAGE	No system operations available (size 13) to send messages.
6	SAVELINK RETURN	Stack over - or underflow.
7	WAITGEN	No system operation available for receiving message, although reserved.

This page is intentionally left blank.

APPENDIX A - RCSL NUMBERS.

Module	Name	Size (bytes)	RCSL43-GL...
Coroutine Monitor	CM002	1592	5089
Testoutput process	RC36-00367	9300	1537
Codeprocedures:			
TESTPOINT	P0072	24	3506
RESETSTACK	P0073	44	3509
CHANGEMASK	P0079	54	3527
CDELAY	P0080	18	4058
INITCOSYS	P0088	132	3304
DEFCOROUT	P0089	120	5362
SETUSEREXIT	P0090	78	3310
INITGENSEM	P0091	34	3313
WAITGENERAL	P0092	178	3316
SIGNAL GENERAL	P0093	56	3319
RETURN ANSWER	P0094	74	3322
CSENDMESSAGE	P0095	72	3325
SAVELINK	P0096	32	3328
RETURN	P0097	12	3331
RELEASE ANSWER	P0098	26	3334
PASS	P0126		
WAITSEM	P0127		
SIGNAL	P0128		
CREATEOPS	P0XXX		

APPENDIX B - PROCEDURE SUMMARY.

Declaration, parameters	Body	Waiting point?	Testoutput & Class
CDELAY (const TIME:integer)	P0080	yes	yes:6,5

CHANGE MASK (var COROUT:string(18);
const CONO,
MASK,
LENGTH:integer);
	P0079	no	no

CREATEOPS (var AREA:string(1);
var OPDESER:integer;
const NO:integer;
var SEM:integer);
	P00XXX	no	yes:1

CSENDMESSAGE(const NAME:string(6);
var SEM,
OPDESCR:integer);
	P0095	no	yes:2 (and 1)

DEFCOROUT (var COROUT:string(18);
const NO,
IDENT,
LENGTH:integer);
	P0089	no	no

INITCOSYS (var AREA:string(1);
var SYSCO:string(18);
const IDENT,
OPS,
CSBUFS:integer);
	P0088	no	no

INITGENSEM (var SEM:integer;
var SEMAREA:string(1);
const DISP:integer);
	P0091	no	no

PASS	P0126	yes	yes:6,5
RELEASEANSWER (var OPDESCR:integer);	P0098	no	no

RESETSTACK (var STACK:integer;
const AREA:string(1);
const DISP,
DEPTH:integer);
	P0073	no	no

RETURN (const STACK:integer);	P0097	no	no
RETURNANSWER (var OPDESCR:integer);	P0094	no	yes:2

Declaration, parameters	Body	Waiting point?	Testoutput & class
SAVELINK (const STACK:integer);	P0096	no	no
SETUSEREXIT	P0090	no	no

SIGGEN (var SEM:integer;
var OPDESCR:integer);
	P0093	no	yes:1

SIGNAL (var SEM:integer);	P0128	no	yes:2

TESTPOINT (const KIND:integer;
var DATA:integer);
	P0072	no	yes, any-preferably 4

WAITSEM (var SEM:integer);	P0127	yes	yes:3,5

WAITGEN (var SEM:integer;
const MASK:integer;
var OPDESCK,
DELAY:integer);
	P0092	Yes	Yes:3,2,5

APPENDIX C - DATA FORMAT SUMMARY.

1. PROCESSDESCRIPTOR. (COMPILER 4, MODIF C)

Procstart	+0	NEXT	QUEUE LINKS
	1	PREV	
	+2	CHAIN	PROCESS CHAIN LINK
(octal)	+3	SIZE	PROCESS DESCRIPTOR SIZE
	+4		PROCESS NAME
	5	NAME	
	6		
	+7	EVENT	HEAD OF EVENTQUEUE
	10		
	+11	BUFFE	CHAIN OF MESSAGE BUFFERS
	+12	PROG	ADDR. OF PROGRAM
	+13	STATE	PROCESS STATE
	+14	TIMER	
	+15	PRIOR	PRIORITY OF PROCESS
	+16	BREAD	BREAK ADDRESS
	+17	AC0	SAVED HARDWARE REGISTERS
	20	AC1	
	21	AC2	
	22	AC3	
	+23	PSW	CPU INSTRUCTION COUNTER (0:14)+CARRY(15)
	+24	SAVE	MUS WORKING LOCATION
	+25	SAVE1	MUSIL INTERPRETER AND CODEPROCEDURE WORKING LOCATIONS
	26	SAVE2	
	27	SAVE3	
	30	SAVE4	
	31	SAVE5	
	+32	R	MUSIL SAVED ARITHEMETIC REGISTER
	+33	PC	MUSIL PROGRAM COUNTER
	+34	OP	OPERATOR COMMUNICATION AREA
	+35	+	
	36	+	
	37	OPERNAME	
	40	+	
	+41	CCOROUT	CURRENT COROUTINE (=ACTIVE,USING CPU)
	+42	LATIME	LAST TIME DELAYS WERE ADJUSTED
	+43	HACTIV	HEAD OF ACTIVE QUEUE
	+44	HANSWER	ANSWER QUEUE
	+45	HDELAY	DELAY QUEUE
	+46	(TRETURN)	POSITION IN EVENT0 OF NEXT MESSAGE [1]
	+47	TRECORD	BIT 0=1, BIT(1:15)=ADDR. OF TESTRECORD ETC.
	+50	CDEVICE	NOT USED, VALUE =0
	+51	MSEM	SEMAPHORE USED FOR MESSAGES
	+52	MCOROUT	COROUTINE WAITING FOR MESSAGES, OR 0
	+53	CUDEX	ADDRESS OF USER EXIT
	+54	CBUFFER	HEAD OF SIZE 13 BUFFER POOL
	+55	ZONE *1	ADDRESSES OF USER DEFINED ZONES.
	56	ZONE *2	
		...	

[1] TRETURN is also used by CSENDMESSAGE to save return.

The save locations are used as follows:

code procedure	SAVE2	SAVE3	SAVE4	SAVE5
CHAININ	addr.OPDESCR	addr.HEAD	-	-
CHAINOUT	addr.HEAD	addr.OPDESCR	-	-
CSENDMESSAGE	addr.NAME	SEM	-	-
DEFCOROUT	addr.COROUT	CONO	COIDENT	CODLENGTH
INITAREA	addr.AREA	-	-	-
INITCOSYS	addr.TRECORD	addr.SYSCO	SYSOPS	-
INITGENSEM	addr.SEM	addr.SEMAREA	-	-
RESETSTACK	addr.STACKREF	addr.SEMAREA	-	-
RETURNANSWER	addr.OPDESCR	-	-	-
SIGGEN	SEM	OPADR	-	-
SWAPVARS	addr.AREA	NEWINDEX	-	-
WAITGEN	SEM	addr.RESULT	-	EVENTMASK

Location SAVE1 is reserved for the MUSIL interpreter.

The testrecord has the following structure:

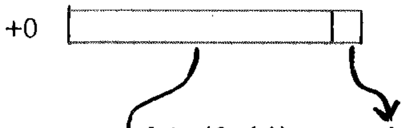
```
           -5  | ZCNT        |  NUMBER OF WAITS ON BSEM RESULTING IN DELAY
           -4  |             |
           -3  |             |
           -2  | EVNO        |  As procstart +46, USED IN WAITGEN CALL
           -1  | BSEM        |  SIZE 13 POOL SEMAPHORE (SIMPLE)
TRECORD    +0  |             |
   (1:15)+1    |             |
  (OCTAL)  2   | TESTRECORD  |
           3   |             |
           4   |             |
           5   |             |
           6   |             |

          +16  |_____________|
```

2. COROUTINE DESCRIPTOR. (SYSTEM PART)

	-2	OPMASK	MASK FOR OPERATION TYPES
	-1	CIDENT	COROUTINE IDENT (<> -1)
address	+0	NEXT	CHAIN IN QUEUES
(OCTAL)	+1	CEXIT	SAVED RETURN ADDRESS
	+2	CLATOP	0 OR CURRENT REMAINING DELAY OR BUF
	+3	CRETURN	POINTER TO HEAD OF 'COROUTINE ARRAY'
	+4	CAC1SAVE	SAVED AC1
	+5	CSPC	SAVED PC (process + 33)
	+6	CPARM	SAVED PARAMETER

3. SIMPLE SEMAPHORE.



state:	bit(0:14)	bit(15)
OPEN	COUNT > 0	1
NEUTRAL	0	1
CLOSED	HEAD OF QUEUE OF WAITING	0

4. GENERAL SEMAPHORE.

	-1	-1	PSEUDO IDENT, ALL ONES.
SEMADR	+0	NEXT	LINK IN DELAY QUEUE
	+1	NXTOP	QUEUE OF OPERATIONS SIGNALLED
	+2	CLATOP	0 OR MIN.DELAY FOR WAITING COROUTINES
	+3	NXTCO	QUEUE OF WAITING COROUTINES

5. OPERATIONS.

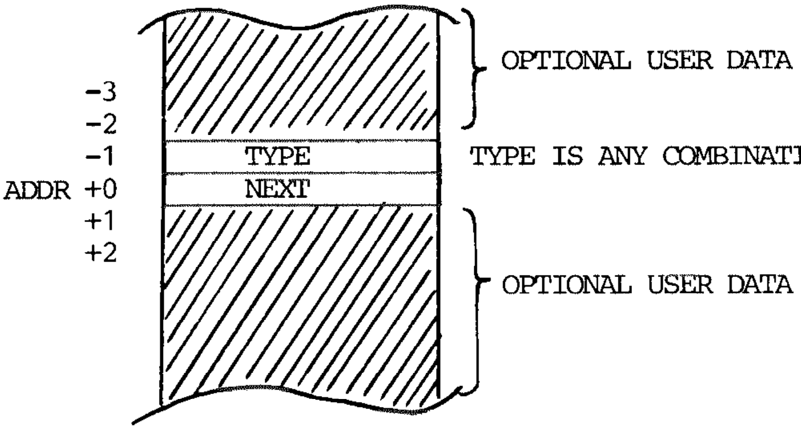
MESSAGE/ANSWER - FROM SIZE-13 POOL.

```
          -12   NEXT        }
          -11   PREV        }
          -10   CHAIN       }
           -9   SIZE=13     }  MUS Message/Answer Fields
           -8   SENDER      }
           -7   RECEIVER    }  RECEIVER:0 FREE  >0 MESSAGE <0 ANSWER
           -6   MESS0       }
           -5   MESS1       }
           -4   MESS2       }
           -3   MESS3       }
           -2   SPECIAL      (ANSWER):ANSWER SEMAPHORE (MESSAGE):ORIG.MESS
           -1   TYPE         1b14:ANSWER     1b13:MESSAGE             BUF
   ADDR  +  0   NEXT
```

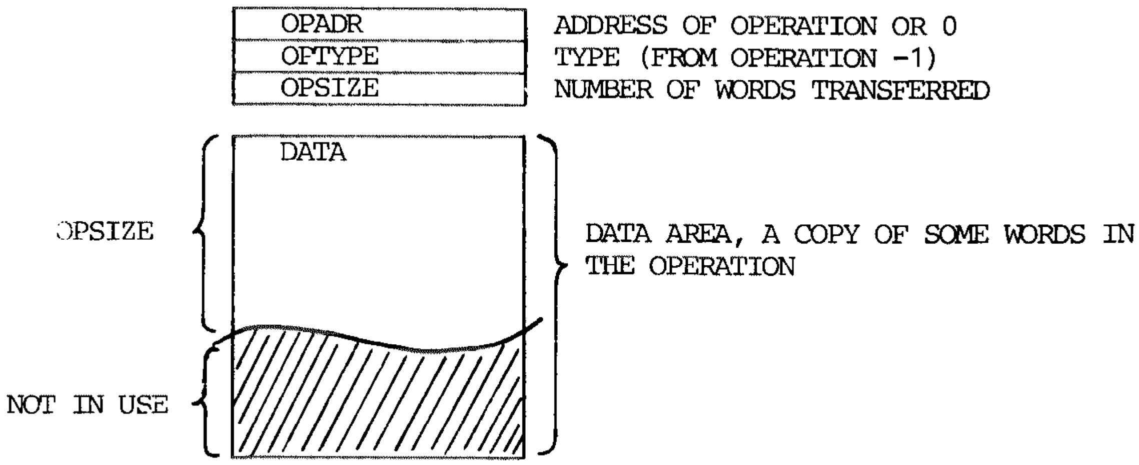
TIMER

```
           -1   TYPE=1b15
     ADDR  +0   NEXT         NOT USED
```

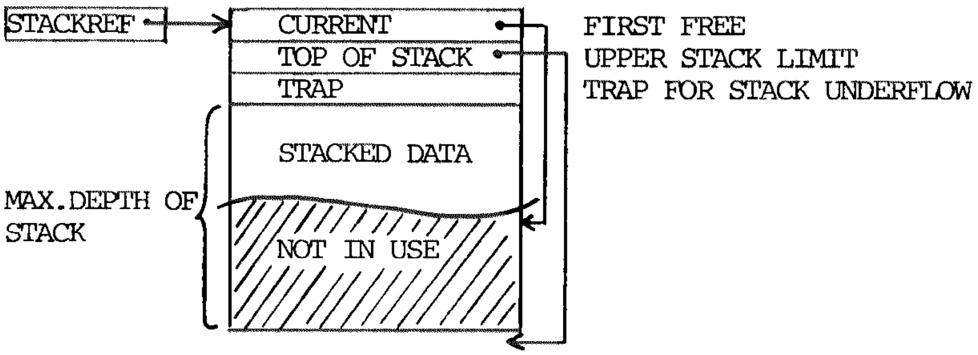
INTERNAL OPERATION



6. OPERATIONS DESCRIPTOR.

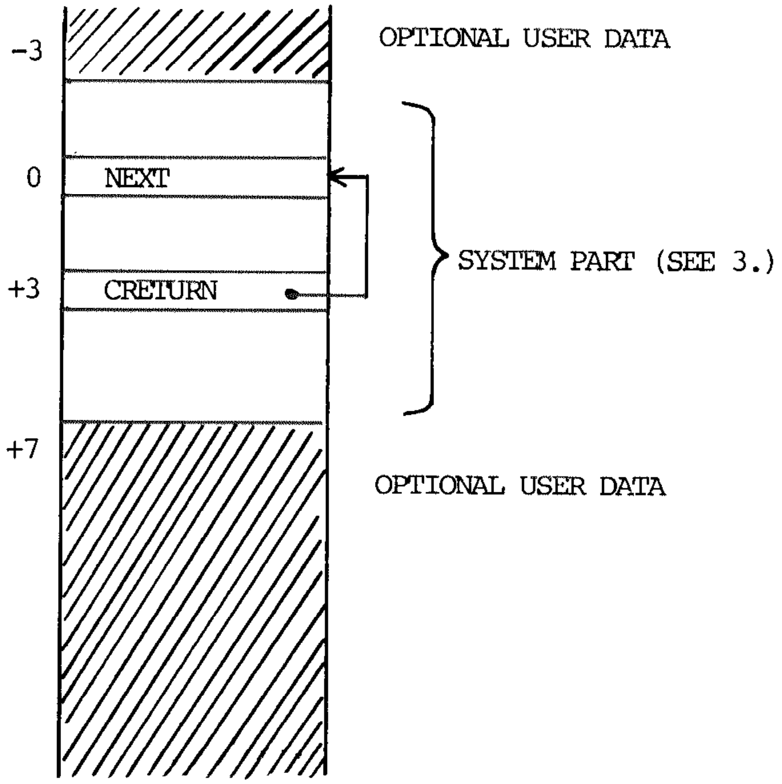

OPADR
ADDRESS OF OPERATION OR 0
OPTYPE
TYPE (FROM OPERATION -1)
OPSIZE
NUMBER OF WORDS TRANSFERRED
DATA
OPSIZE
DATA AREA, A COPY OF SOME WORDS IN THE OPERATION
NOT IN USE

7. STACK.

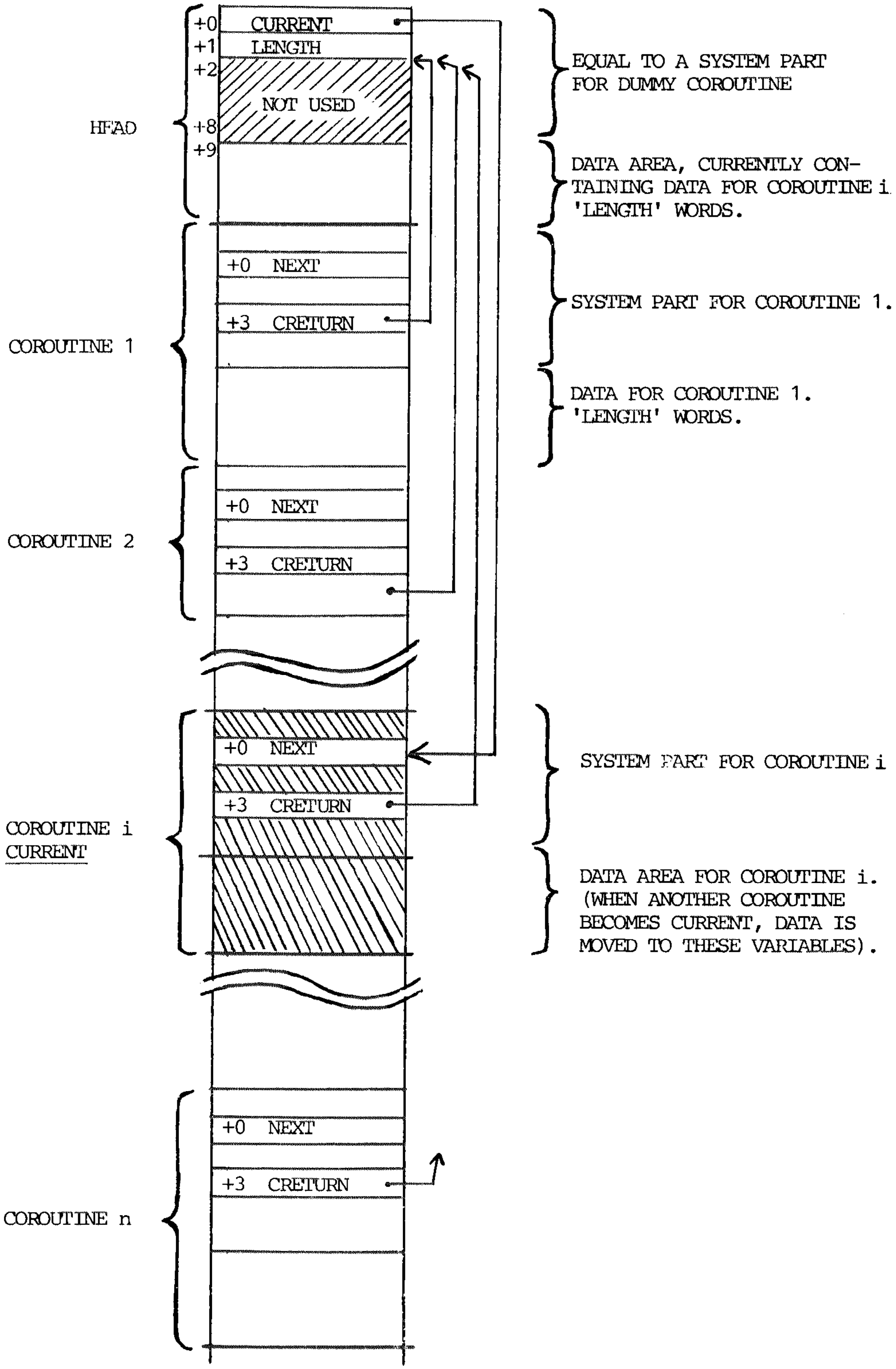


8. COROUTINE SYSTEMS.

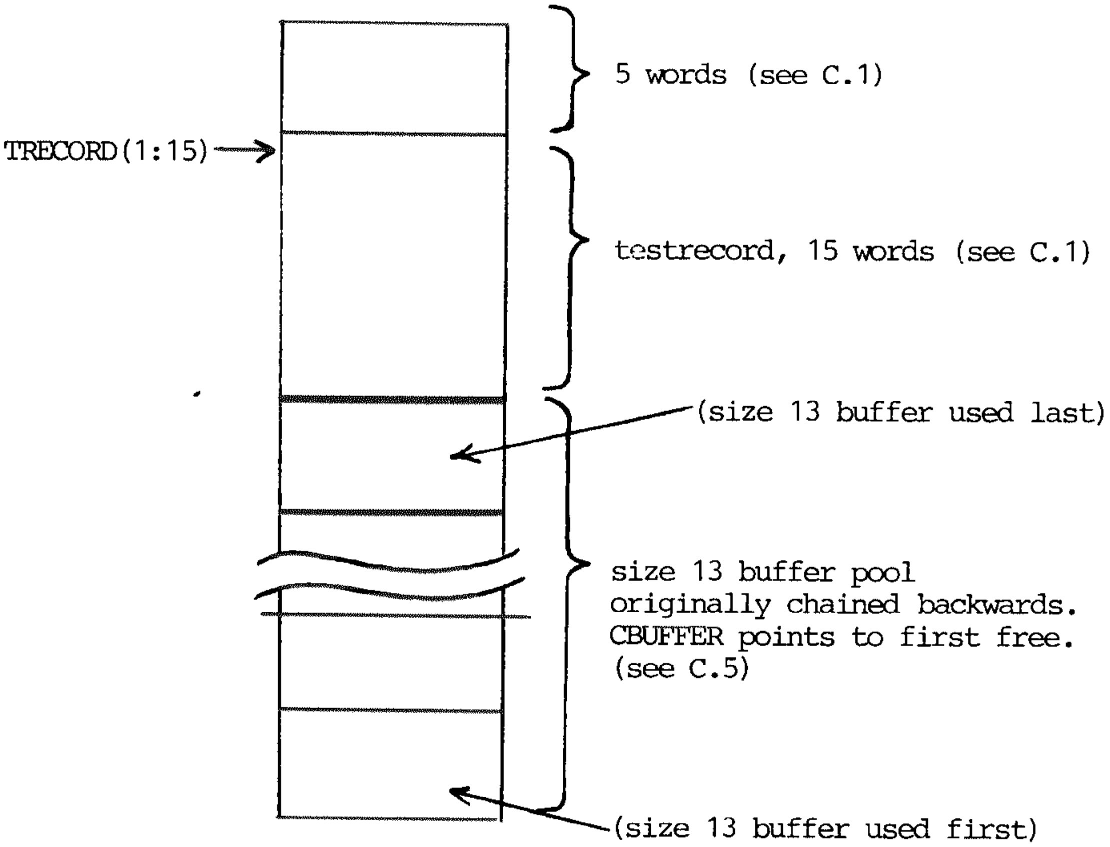
SINGLE COROUTINE



MULTIPLE INCARNATION COROUTINES - WITH IDENTICAL DATA AREA STRUCTURE.



9. SYSTEM AREA.


5 words (see C.1)
TRECORD(1:15)
testrecord, 15 words (see C.1)
(size 13 buffer used last)
size 13 buffer pool
originally chained backwards.
CBUFFER points to first free.
(see C.5)
(size 13 buffer used first)

APPENDIX D - SOME ADDITIONAL CODEPROCEDURES OF INTEREST.

The library of codeprocedures contains some codeprocedures in addition to the coroutine procedures, which may be of interest when coding e.g. driver-like modules. As these procedures at the moment are very poorly documented, this appendix gives a short description in order to make them available for general use. The selection of which procedures from the 154 existing should be described has been done somewhat arbitrarily.

The procedures have been divided into 4 groups (the section numbers refer to the description found in the remaining part of this appendix):

1. Array simulation:

LOAD (base, index)	integer arrays	(1.1)	-	43-GL 631
STORE (base, index)			-	43-GL 655
INITAREA (area, vars)	general arrays	(1.2)	P0074	43-GL 3512
SWAPVARS (area, index)			P0075	43-GL 3515

2. Queue administration:

CHAININ (oper, head)	(2)	P0076	43-GL 3518
CHAINOUT (head, oper)	(2)	P0077	43-GL 3521
EXAMINE (oper)	(2)	P0078	43-GL 3524

3. MUSIL addressing extension:

TAKEADDRESS (stringvar, addr)	(3.1)	-	43-GL 661
INITZONE (zone, shares, length, area)	(3.4)	P0055	43-GL 5161
CREATEMESS (area, length)	(3.3)	P0054	43-GL 2350
SPRIO (name, addr, prio)	(3.2)	-	-

4. Data handling:

ACONVERT (fromaddr, toaddr, table, count);	(4.4)	P0131	43-GL 5695
AFILL (value, toaddr, count);	(4.3)	P0119	43-GL 5674
AMOVE (fromaddr, toaddr, count);	(4.1)	P0120	43-GL 5677
CONVIN (source, disp, addr, count, table);	(4.4)	-	43-GL 613
CONVOUT (addr, dest, disp, count, table);	(4.4)	-	43-GL 616
FILL (value, dest, disp, reps)	(4.3)	-	43-GL 622
IEXTRACT (result, area, disp)	(4.2)	P0123	43-GL 5686
IINSERT (value, area, disp)	(4.2)	P0124	43-GL 5689
INVALUE (value, addr, type)	(4.2)	P0121	43-GL 5680
MOVIN (fromstr, disp, toaddr, length)	(4.1)	-	43-GL 640
MOVOUT (fromaddr, tostr, disp, length)	(4.1)	-	43-GL 643
OUTVALUE (result, addr, type)	(4.2)	P0122	43-GL 5683

Parameter conventions and data structures used by these procedures are described in the following sections.

1. Array Simulation.

1.1 Integer arrays.

The procedures LOAD and STORE are implemented in a way using knowledge about the specific storage allocation scheme used by the MUSIL compiler. They make use of the following 'pseudo array':

```
base  : integer;
area  : string(l);
```

The 'base' serves two purposes. It contains the value just loaded, or to be stored, and it is used to address the array structure. The 'area' contains actual values. For a given length l, which shall be even, valid values of the index are 0 to l/2-1. No checking is, however, done.

Procedure declarations and descriptions:

```
procedure LOAD (var   BASE  : integer;
                const INDEX : integer);
codebody;
```

The word at relative location INDEX (0-(l/2-1)) is moved into BASE.

```
procedure STORE (var   BASE  : integer;
                 const INDEX : integer);
codebody;
```

The opposite of LOAD. The value of BASE is moved to relative location INDEX (0- (l/2-1)).

Note: The operations LOAD (A, -1) and STORE (A, -1) has no effect, as the value of A is moved to itself.

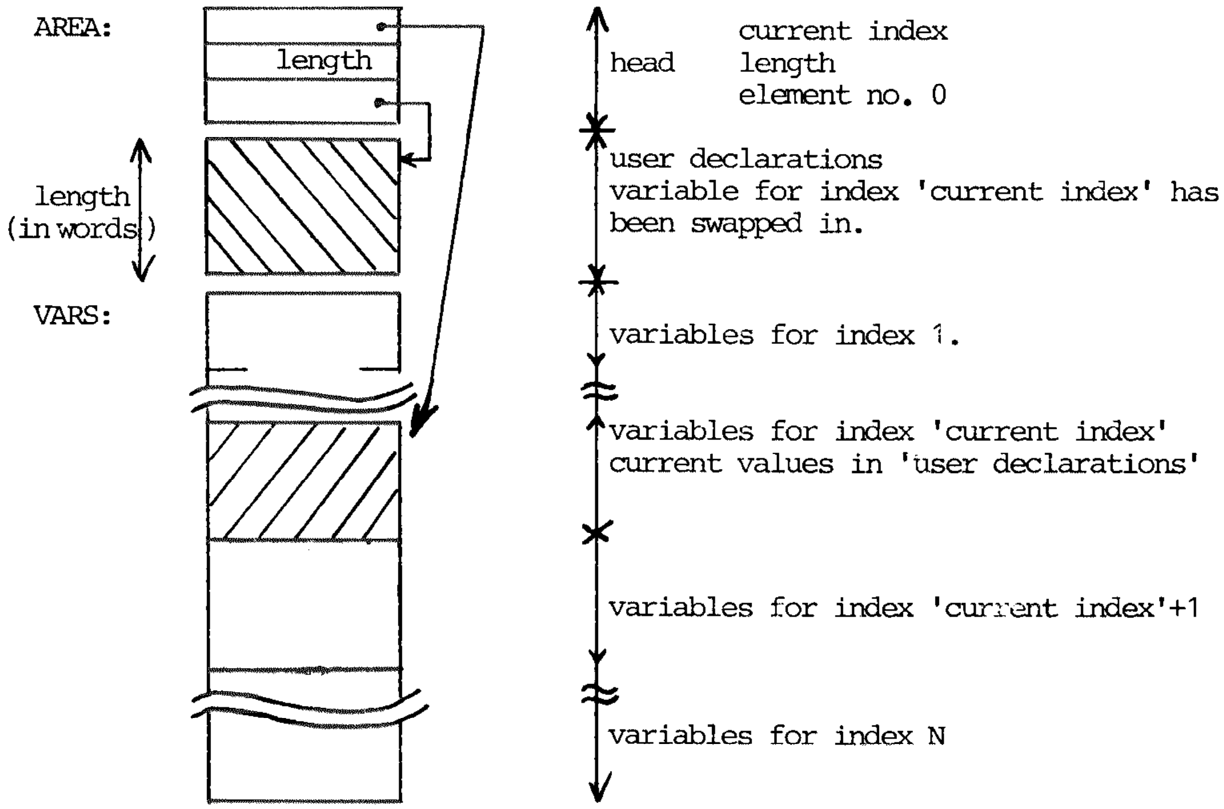
1.2 General arrays.

The implementation of INITAREA and SWAPVARS uses knowledge about the specific storage allocation scheme used by the MUSIL compiler. The procedures use a pseudo-array with the following general structure:

```
AREA : integer;             ! 3 words header:             !
head : string(4);           !    +0 : current index       !
                            !    +1 : length              !
                            !    +2 : address of element no.0!
! here comes user variables !

VARS : string(1) ;          ! elements 1 ...              !
```

The length of the string containing the elements (here VARS) should be computed as the length of the area containing the user variable declarations times the maximum number of elements, N. The total length of the variables can be found using the method described in chapter 4.2.2. The index may vary between 1 and N. No checking is, however, done to verify this.


AREA:
length
length (in words)
VARS:
head
current index
length
element no. 0
user declarations
variable for index 'current index' has been swapped in.
variables for index 1.
variables for index 'current index'
current values in 'user declarations'
variables for index 'current index'+1
variables for index N

Procedure declarations and descriptions:

```
procedure INITAREA (var AREA : integer;
                    var VARS : string(1));
codebody P0074;
```

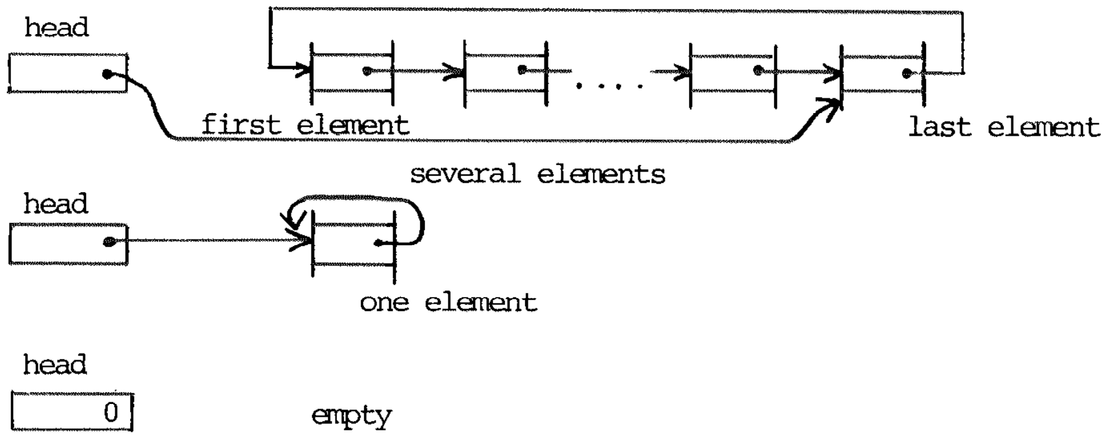
The procedure initializes the 3 word long head. Current index is set to 1. The length of the user variables declarations is computed as the difference between the addresses of the head and the VARS area.

```
procedure SWAPVARS (var   AREA     : integer;
                    const NEWINDEX : integer);
codebody P0075;
```

If NEWINDEX is different from current index, then the variables in the user area is written back where they belong, and the variables belonging to NEWINDEX are fetched, and current index is updated. No checking is done that NEWINDEX has a valid value. The area should have been initialized by INITAREA before use of SWAPVARS. A whole number of words are always moved.

2. Queue Administration.

The procedures CHAININ, CHAINOUT and EXAMINE works on a first in/first out queue structured like this:



The head and the link fields are one word long. The links are absolute storage word addresses. The queue elements may be e.g. operations.

Procedure declarations and descriptions:

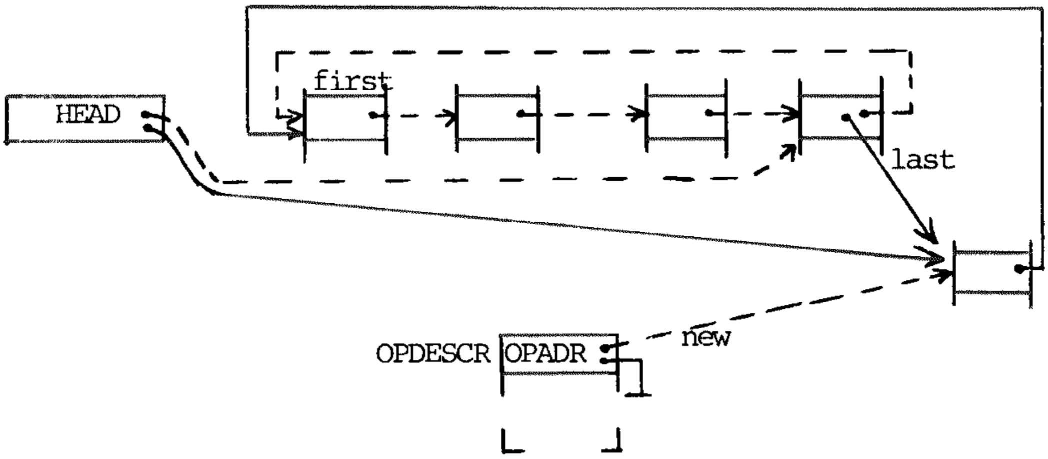
```
procedure CHAININ (var OPADR : integer;
                   var HEAD  : integer);
codebody P0076;

procedure CHAINOUT(var HEAD  : integer;
                   var OPADR : integer);
codebody P0077;

procedure EXAMINE (var OPADR : integer);
codebody P0078;
```

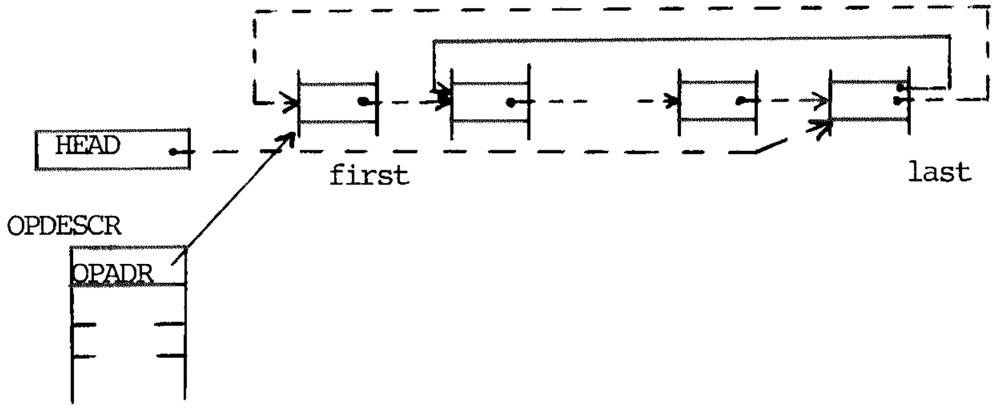
The OPADR is the address field of an operations descriptor, pointing to the operation. HEAD is the head of the queue initially NIL = zero.

CHAININ puts the element in OPADR into the queue as the last element:



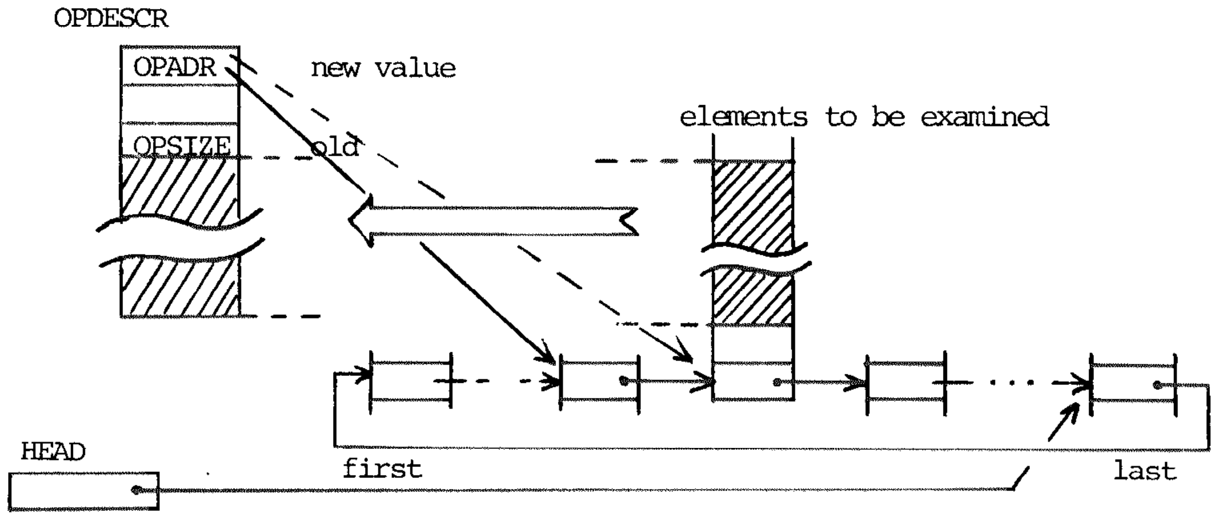
OPADR is set to zero. If OPADR was initially zero, then the procedure has no effect.

CHAINOUT takes out the first element in the queue and sets OPADR to point to the element. If the queue was empty, OPADR is set to zero.



No other fields in OPDESCR are touched.

The procedure EXAMINE is used to make the contents of the elements in a queue sequentially available for inspection. OPADR points to an element in the queue, and OPSIZE should have an appropriate value. A call of EXAMINE will then take the next element in the queue and move the contents into the OPDESCR fields in the same way as WAITGEN (see 3.1.2). OPADR is updated to point to this element. OPTYPE is not changed. If OPADR was zero the procedure has no effect.



A queue maintained by CHAININ/CHAINOUT is examined in this way:

```
(1) first element:      opdescr.opadr:=  qhead;
                        opdescr.opsize:= length;
                        EXAMINE(opdescr.opadr);

(2) following elements: EXAMINE(opdescr.opadr);
```

3. MUSIL addressing extensions.

The procedures TAKEADDRESS, SETSHARES, SPRIO and CREATEMESS are a sort of extension to the facilities in the MUSIL compiler with respect to addressing and ressource allocation.

3.1 Find absolute address of a string.

Declaration:

```
procedure TAKEADDRESS (var STRVAR: string(1);
                       var ADDR  : integer);
codebody;
```

The absolute storage address (a byte address) of the first byte of STRVAR is returned in ADDR.

3.2 Set multiprogramming priority.

```
procedure SPRIO (const PROCNAME : string(6);
                 var   PROCADDR : integer;
                 const PRIORITY : integer);
codebody;
```

The procedure searches the processes for a process with name PROCNAME. If it does not exist, a value of zero is returned in PROCADDR. If it is found, its process descriptor address is returned in PROCADDR, and its priority is set to PRIORITY, if this is nonzero. Typical priorities are

MUSIL program:	1b8	(=128)
Drivers :	1b0, or 1b0 + some value.	

3.3 Create additional message buffers.

```
procedure CREATEMESSBUFS (var BUFAREA : string(1);
                          const LENGTH: integer);
codebody P0054;
```

The procedure has only effect the first time it is called in a program after load. LENGTH is given in bytes. The BUFAREA (0)... BUFAREA (LENGTH -1) is divided into LENGTH//20 message buffers. The fields mess0, ..., mess3 are initialized with the 8 byte text 'NOT USED' which may be convenient in connection with later core dumps.

3.4 Set fields in zone and share descriptors.

```
procedure INITZONE (file Z;
                    const SHARES : integer;
                    const LENGTH : integer;
                    const AREA   : integer);
codebody P0155;
```

The file Z should be declared like

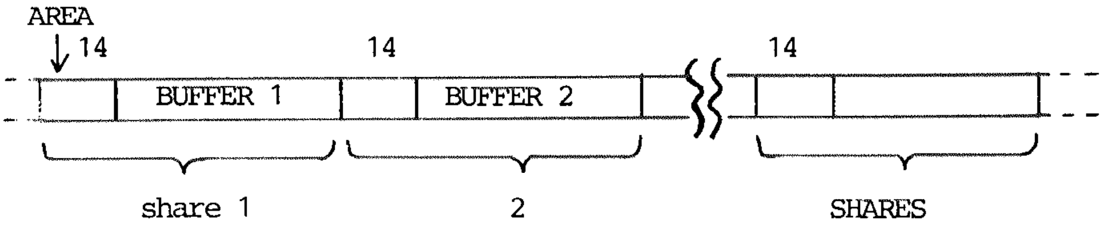
```
    Z : file ..., ..., 1, 1...
        ...
        of...  ;
```

i.e. with 1 share and 1 byte long buffer.

The value of AREA is an absolute byte storage address obtained by a call of e.g. TAKEADDRESS, and should be even. The parameter SHARES gives the number of shares to be created. The parameter LENGTH is the buffer length in bytes. The procedure uses

SHARES * (7 + (LENGTH+1)//2) * 2 bytes

in AREA, which is structured as follows:



The share states are set to free. The zone variables Z.zlength and Z.zshared are both set to LENGTH, and Z.zused is initialized.

Note: (SHARES -1) extra message buffers have to be created, e.g. by CREATEMESSBUFS.

4. Data handling.

These procedure extend the possibilities in the standard MUSIL procedures MOVE, INSERT and CONVERT, and the operators BYTE and WORD.

4.1 Data movement.

Declarations:

```
procedure AMOVE    (const FROMADDR : integer;
                    const TOADDR   : integer;
                    const COUNT    : integer);
codebody P0120;

procedure MOVIN    (const FROMSTR  : string(1);
                    const FROMINDX : integer;
                    const TOADDR   : integer;
                    const COUNT    : integer);
codebody;

procedure MOVOUT   (const FROMADDR : integer;
                    var   TOSTR    : string(1);
                    const TOINDX   : integer;
                    const COUNT    : integer);
codebody;
```

These procedures are extensions of the MUSIL standard MOVE. The parameters designated as FROMADDR and TOADDR are absolute storage byte addresses. The number of bytes to be moved is given in COUNT. The actual move is done by the MUS utility MOVE.

The procedure AMOVE transfer COUNT bytes from FROMADDR to TOADDR.

The procedure MOVIN moves COUNT bytes from FROMSTR (FROMINDX) to TOADDR and on. The procedure MOVEOUT moves COUNT bytes from FROMADDR to TOSTR (TOINDX) and on. The FROMINDX and TOINDX are displacements, and a value of zero indicates first byte in the string.

4.2 Insertion and extraction.

```
procedure IEXTRACT  (var    RESULT    : integer;
                     const FROMSTR   : string(1);
                     const FROMINDX : integer);
codebody P0123;

procedure IINSERT   (const VALUE     : integer;
                     var    TOSTR     : string(1);
                     const TOINDX    : integer);
codebody P0124;

procedure INVALUE   (const VALUE     : integer;
                     const TOADDR    : integer;
                     const ATYPE     : integer);
codebody P0121;

procedure OUTVALUE  (var    RESULT    : integer;
                     const FROMADDR : integer;
                     const ATYPE     : integer);
codebody P0122;
```

These procedures are extensions of the MUSIL BYTE and WORD operators, and the standard procedure INSERT.

The parameters TOADDR and FROMADDR are absolute storage addresses. TOSTR and FROMSTR are MUSIL strings.

The procedures IEXTRACT and IINSERT respectively extracts an integer from a string and inserts an integer into a string. The TOINDX and FROMINDX are the displacements in the string for the place of most significant 8 bits of the integer, the first byte having a displacement of zero. IEXTRACT works like the WORD operator, but the two bytes may be displaced to any position.

The procedures INVALUE and OUTVALUE works on one or two bytes according to ATYPE:

ATYPE	INVALUE	OUTVALUE
1 (byte)	insert VALUE(8:15)	fetch one byte.
2 (word)	insert VALUE(0:15) in two bytes.	fetch two bytes.

INVALUE inserts one or two bytes at address TOADDR. OUTVALUE fetches one or two bytes at address FROMADDR, like the BYTE or WORD operator.

4.3 Duplicate bytes.

```
procedure AFILL  (const BYTEVALUE :integer;
                  const TOADDR    :integer;
                  CONST COUNT     :integer);
codebody P0119;

procedure FILL   (const BYTEVALUE :integer;
                  const TOSTR     :string(1);
                  const TOINDX    :integer;
                  const COUNT     :integer);
codebody;
```

The procedure inserts the value given in BYTEVALUE extract 8 in the destination strings COUNT times. FILL inserts the value in TOSTR (TOINDX) ... TOSTR (TOINDX + COUNT -1). AFILL uses the address TOADDR which is an absolute storage byte address.

4.4 Move with conversion.

```
procedure ACONVERT (const FROMADDR  : integer;
                    const TOADDR    : integer;
                    const TABLEADDR : integer;
                    const COUNT     : integer);
codebody P0131;

procedure CONVIN   (var   FROMSTR   : string(1);
                    const FROMINDX  : integer;
                    const TOADDR    : integer;
                    const COUNT     : integer;
                    const TABLEADDR : integer);
codebody;

procedure CONVOUT  (const FROMADDR  : integer;
                    var   TOSTR     : string(1);
                    const TOINDX    : integer;
                    const COUNT     : integer;
                    const TABLEADDR : integer);
codebody;
```

These procedures work like AMOVE, MOVIN and MOVOUT, except that they convert the bytes moved by means of the table specified in the parameter TABLEADDR, which is an absolute byte storage address.