

RS-M

RCSL: 52-AA187

pp: 1 : 54

Author: Bo Bagger
Laursen

Edited: August, 1974

RC 3500 Monitor 1

Keywords: RC 3500, Software, Monitor.

Abstract: This paper is a description of the RC 3500 Monitor 1, which is based on the concepts of parallel processes, coroutines, and semaphores.

A/S REGNECENTRALEN

3-9, Guldsmødegade

DK-8000 Aarhus C.

CONTENTS

Section

INDEX	0
GENERAL DESCRIPTION	1
Purpose	1.1
FUNCTION	2
Processes, Monitor	2.1
Synchronization, Common Variables, Queues	2.2
Mutual Exclusion, Critical Regions	2.3
Priorities and Reentrancy	2.4
Priorities, Levels	2.4.1
Stretches, Indivisibility	2.4.2
Reentrancy, Private Variables	2.4.3
Semaphores	2.5
Chained Semaphores	2.5.1
Simple Semaphores	2.5.2
Mutual Exclusion	2.5.3
Driver Rules	2.6
SOFTWARE TIMER	3
TESTOUTPUT	4
Testoutputrecord	4.1
Testoutput Driver	4.2
Testoutput Options	4.3
REFERENCE DOCUMENTS	5
IMPLEMENTATION	6
Names, Functions, Options, Formats	6.1
h-Names, Monitor Functions	6.1.1
i-Names, Options	6.1.2
j-Names and Formats	6.1.3
SYSTEM INITIALIZATION	7

CONTENTS (continued)

Section

SIZE, EXECUTION TIMES	8
Size	8.1
Execution Times	8.2
TESTOUTPUT SYSTEM 1	Appendix A
COROUTINE EXAMPLE AND DRIVER EXAMPLE	Appendix B

pp. 1 : 54.

The section numbers are specified.

Activate	2.2 - 2.4.1
Active	2.2 - 2.4.1
Active queue	2.4.1 - 6.1.1 - 6.1.3
Chained semaphore	2.5.1 - 6.1.3
Common variables	2.2
Coroutine	2.1
Coroutine description	2.4.3 - 6.1.3
Coroutine mask	4.3
Critical region	2.3
Current coroutine	6.1.1
Driver	2.1 - 2.6
Dummy coroutine	2.4.1 - 4.3
Executing times	8.2
First of free store	6.1.1
Formats	6.1.3
H-names	6.1.1
I-names	6.1.2
Incarnation	2.4.3
Indivisibility	2.3 - 2.4.2
Interrupt level	2.1
J-names	6.1.3
Monitor	2.1
Monitor functions	6.1.1
Monitor queue	2.2 - 6.1.3
Mutual exclusion	2.3 - 2.5.3

Operation	2.5.1 - 6.1.3
Options	6.1.2
Pass	6.1.1
Priority	2.4.1
Priority level	2.4.1
Private variables	2.4.3
Process	2.1
Queues	6.1.3
Reentrancy	2.4.3
Regret	6.1.1
Regret action	6.1.3
Regret next	6.1.1
Reset	7
Return from regret action	6.1.1
Return from start action	6.1.1
Running coroutine	2.2 - 2.4.1
Scheduling	2.4.1
Semaphore	2.2 - 2.5 - 6.1.3
Semaphore functions	2.2
Signal	2.5.2 - 6.1.1
Signal chained	2.5.1 - 6.1.1
Simple semaphore	2.5.2 - 6.1.3
Simple regret	6.1.1
Size	8.1
Software timer	3
Start action	6.1.3
Stretch	2.4.2
Synchronization	2.2

Testbits	4.3	
Testoutput	4	- 6.1.1
Testoutput driver	4.2	
Testoutput mask	4.3	
Testoutput options	4.3	
Testoutput record	4	- 4.1
Timer operation	3	
Timer semaphore	6.1.1	
Top of program system	6.1.1	
Type 2 limit	4.3	
Wait	2.5.2	- 6.1.1
Wait chained	2.5.1	- 6.1.1
Waiting	2.2	

The Monitor 1 is designed, programmed, and implemented by Per Mondrup in the period 1972/73.

Purpose.

1.1

The primary design goal for the RC 3500 Monitor 1 is to implement multiprogramming by means of semaphores. The implementation incorporates the following essential features:

1. Queued communication (FIFO) and mutual exclusion by means of semaphores.
2. Possibilities for reentrant parallel processes (coroutines) with levels of priority.
3. Built-in testoutput for debugging and maintenance.
4. Software timer.

Processes, Monitor.

2.1

At run time a program system in the RC 3500 consists of a set of processes running in parallel. A process is defined as the sequential execution of an ordinary program, possibly containing loops, conditional branches, procedure calls, etc.

Because of the special architecture of the RC 3500, the processes are divided in two categories, namely coroutines and drivers.

Drivers are processes which run on an interrupt level > 0 and are therefore assigned their own set of working registers.

Coroutines are processes which run on the same interrupt level as the monitor, interrupt level 0. Coroutines and the monitor therefore share one common set of working registers.

The monitor is a collection of functions which make the computer appear as if it were running all the processes at the same time.

The monitor is responsible for the scheduling of the coroutines and the communication between all processes.

The scheduling of the coroutines is accomplished by means of a set of tables describing all coroutines in the computer.

The communication between processes is accomplished by means of chained semaphores.

Synchronization, Common Variables, Queues.

2.2

We will consider three kinds of synchronizing:

1. Synchronizing to a device:

A driver will be synchronized to a peripheral device by means of hardware interrupts. This means that the driver will execute a few

instructions and may decide to wait for a new interrupt from the device before it proceeds. This is accomplished by clearing the interrupt bit of the associated interrupt level.

2. Synchronizing a coroutine to another process:

A coroutine A synchronizes to another process B by introducing a queue. Normally, the queue is empty and coroutine A may wait for process B putting something in the queue. Coroutine A may perform the waiting by calling the monitor function "wait chained (queue)". If there is something in the queue, coroutine A proceeds immediately and may access the common variables in the queue element. If the queue is empty, coroutine A starts waiting for the queue, and the monitor may then let another coroutine utilize the CPU-time.

The other end of this synchronization is caused by process B when it puts a queue element into the queue by means of the monitor function "signal chained (queue, element)" after having accessed the common variables in the queue element. If coroutine A at this moment is waiting for the queue, the monitor will activate coroutine A, which means that it will get CPU-time, when convenient.

3. Synchronizing a driver to another process:

A process which directly handles a peripheral device and waits for its interrupts is called a driver. Coroutines on the monitor level synchronize only by means of "wait chained" and "signal chained".

A driver runs on an interrupt level > 0 and possesses its own working register set. Therefore a driver synchronizes with other processes in a special way.

A driver communicates a queue element to another process by starting a subprocess in the monitor called "the interrupter" which

runs on an interrupt level greater than all processes in the program system, thereby ensuring indivisibility. The interrupter places the queue element in a special queue called the "monitor queue", belonging to the monitor. The queue element contains a semaphore address, where the monitor places the queue element in due time.

When a driver wants a queue element from a queue, it must find out itself if there are any queue elements in the queue. If the queue is not empty, the driver may process the first queue element. If the queue is empty, the driver stops the queue (see further in section 2.6), and brings the driver in a state so that nothing will happen unless a new queue element arrives at the queue.

In this way two processes A and B may synchronize to each other and communicate by accessing common variables in queue elements in a controlled manner.

The monitor functions "wait chained" and "signal chained" are called semaphore functions. They are described in detail in sections 2.5.1 and 6.1.1.

At any given moment most coroutines will be waiting for a queue if the system is well-balanced. The remaining few coroutines are active, which means that they want to use the CPU. One of the active coroutines is running which means that it actually uses the CPU. The monitor selects the running coroutine among the active coroutines according to a priority scheme explained in section 2.4.1.

Mutual Exclusion, Critical Regions.

2.3

Suppose two coroutines want to share one common resource in such a way that the two coroutines are guaranteed exclusive access to the common resource, e.g. updating a common table, common variables, or calling a procedure which is not reentrant (see page 2-6).

We must find a solution which mutually excludes the two coroutines access to the resource.

The part of the program where the table is updated or where the procedure is executed is called a critical region, and it should be performed indivisibly with respect to the other coroutine.

One solution is to use the monitor functions "wait chained" and "signal chained" as follows:

Introduce a queue "sema" initialized with one dummy queue element. When one of the coroutines wants to access the common resource, it proceeds like this:

```
loop:      .
           .
           .
           wait chained (sema)
           <critical region>
           signal chained (sema, queue element)
           .
           .
           .
           goto loop
```

If during <critical region> the other coroutine wants to access the common resource it will also execute "wait chained (sema)". But now the queue is empty, and the coroutine will wait until the first one activates it as a result of "signal chained (sema, queue element)".

For another solution see section 2.5.3.

Priorities and Reentrancy.

2.4

In this section we will consider a few concepts which may give an efficient system with fast response time (priorities) and a modest storage requirement (reentrancy).

The active coroutines are those not waiting for the moment, but wanting CPU-time. The active coroutines are organized in queues according to their priority. All active coroutines with the same priority form a priority level. There exists one queue for each priority level.

The active coroutines are scheduled every time the set of active coroutines changes. That may happen in two situations. Firstly, when a coroutine performs a call of a monitor function. Secondly, when the monitor processes a queue element (an operation) in the monitor queue originating from a driver.

The monitor always selects the running coroutine as the first coroutine in the highest priority queue which is not empty.

When a coroutine is activated, it is placed as the last element in the priority queue associated its priority.

The system has a dummy coroutine with lowest priority which is always active. This dummy coroutine only runs when no other coroutines are active. See further in section 4.3 about the functions performed by the dummy coroutine.

Stretches, Indivisibility.

A coroutine can be divided into stretches. A stretch starts when the coroutine is activated and it terminates when the coroutine calls a wait function (e.g. "wait chained").

Coroutines within one priority level are scheduled so that each coroutine always completes its stretch before another coroutine from the set becomes running. In other words, they mutually exclude each other from having CPU-time and the critical regions are the stretches.

A coroutine may be interrupted at any time between two instructions by a driver, because of the interrupt system, or by a coroutine on a higher priority level, because of the scheduling philosophy.

A set of coroutines on the same priority level can therefore easily update common variables if the entire updating takes place within a stretch.

A set of coroutines on different priority levels would have to use a semaphore to guarantee the mutual exclusion and the consistency of the common variables.

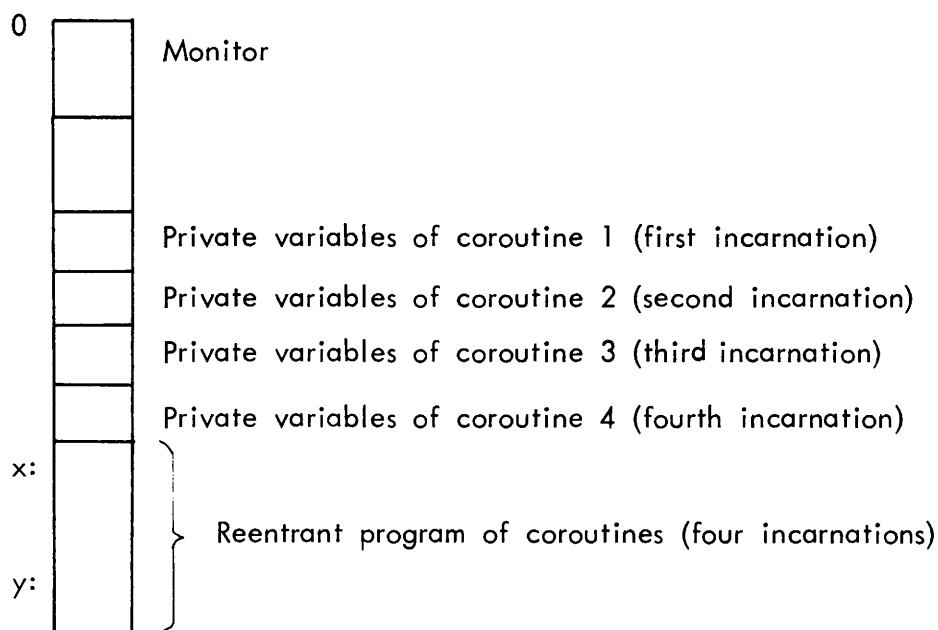
A driver may be interrupted at any time between two instructions by a driver on a higher interrupt level, unless the interrupt system is disabled (which may cause loss of interrupts if the system is disabled too long).

Reentrancy, Private Variables.

2.4.3

Consider an example where we have four coroutines, where all coroutines follow exactly the same program, but each coroutine has its own set of private variables. The common program need only be stored once, and it is called reentrant in four incarnations.

The storage layout for this system might be:



Notice that the four incarnations are four different processes. When cor. 1 runs, an index register (w3) normally points to the private variables of cor. 1 and the program will address the variables relative to w3. When cor. 2 runs, w3 points to the private variables of cor. 2, and the same program then works on these variables. If the coroutine loses this pointer, it can be reestablished via the monitor variable h10.

A special private variable is the program counter w0. At a given moment, cor. 1 may have its program counter point at x, while cor. 2 has its program counter point at y, etc. Thus, although the program is the same, the coroutines may find themselves executing different parts of it at a given moment.

The program counter and the registers w1, w2, and w3 are stored in the coroutine description. The coroutine description consists of 7 words of the private variables, and it is maintained by the monitor (see section 6.1.3).

From the monitor point of view all coroutines are reentrant in principle, because the monitor is concerned with the coroutine description only.

The instruction counter may point anywhere. At system initialization (reset) all coroutines become active (their coroutine descriptions are linked together). The monitor makes them run one at a time as normally, and the program they execute is determined by the initial program counter in the process description.

The monitor functions are not reentrant procedures.

This is the reason why the monitor queue exists. If a driver on interrupt level > 0 starts the interrupter, while the monitor is executing a monitor function on interrupt level 0, the processing of the monitor queue is delayed until the monitor has finished the current monitor function.

Semaphores.

2.5

In this section we give the exact explanation of semaphore functions like "wait chained" and "signal chained".

A chained semaphore is a double word in the store representing a queue, plus 1 or 2 words more (the exact format is found in section 6.1.3).

If the queue is empty, one or more coroutines may wait for an element (hereafter called operation) to be put into the queue. In this situation the chained semaphore links together the coroutine descriptions for the waiting coroutines. The situations can be described as the semaphore being in the following four states:

1. open: The queue contains some operations. The semaphore contains the head and tail of a chain linking the operations together.
2. passive: The queue is empty, but no process waits for it.
3. closed: One or more coroutines wait for the queue. The semaphore contains the head and tail of a chain linking the coroutine descriptions together.
4. stopped: A process is stopped and waits for becoming started, when an operation is placed in the queue. Normally, a semaphore is only stopped by a driver.

The semaphore functions "wait chained" and "signal chained" work as follows:

wait chained (semaphore addr.):

If the semaphore is open, the first operation is removed from the chain. The coroutine is activated and the address of the operation is passed to the coroutine. If the semaphore is passive or closed, the calling coroutine waits for the semaphore.

signal chained (semaphore addr., operation addr.):

If the semaphore is open or passive, the operation pointed to by "operation addr." will be added at the tail of the chain.

If the semaphore is closed, the first coroutine waiting for the semaphore is activated and it receives the operation. The calling coroutine stays active, although it can lose the CPU for some time in case the activated coroutine has a higher priority.

If the semaphore is stopped, the semaphore must contain an address of a piece of code in the receiving process, which is executed. This piece of code must start the receiving process and returns to the monitor.

The operations can be anywhere in the store, for instance in the private variables or in a common table. The monitor is not concerned with the length of the operations, as it only links together the first word of the operations.

It is essential that the semaphore functions are executed indivisibly (with the interrupt system disabled), so that "signal chained" and "wait chained" update the common chains properly. Outside the monitor disabling is never necessary in coroutines as mutual exclusion can be handled by semaphores.

Simple Semaphores.

2.5.2

A simple semaphore consists only of a double word in the store and is handled much like a chained semaphore. The only differences are that a simple semaphore cannot be stopped and that the queue of operations is replaced by a counter symbolizing the length of an abstract queue (multiplied by two). In the passive or closed state the simple semaphore is exactly like a chained semaphore.

The associated two semaphore functions work as follows:

`wait (semaphore addr.):`

If the semaphore is open, it is decreased by 2, and the calling coroutine is activated.

If the semaphore is passive or closed, the calling coroutine waits for the semaphore.

`signal (semaphore addr.):`

If the semaphore is open or passive, it is increased by two. If the semaphore is closed, the first coroutine waiting for the semaphore is activated.

The calling coroutine stays active, although it can lose the CPU for some time in case the activated coroutine has a higher priority.

Simple semaphores are used in cases where the operations do not contain essential information or in cases where the operations are handled and chained in a user defined way.

Mutual Exclusion.

2.5.3

In section 2.3 we saw an example of mutual exclusion guaranteed by means of a chained semaphore. However, the operation is superfluous and a simple semaphore can be used.

A resource or a critical region can be "protected" by means of a simple semaphore as follows:

`mutex` `simple semaphore initially 2`

Access to critical regions must always be performed like this:

```
wait (mutex);  
critical region;  
signal (mutex);
```

If long independent critical regions exist, one "protecting" semaphore should be used for each region. This means that if one coroutine is inside a long critical region, another coroutine will not be prevented from entering an independent critical region.

Driver Rules.

2.6

Most drivers are controlled by operations sent to them from other processes. As a standard the operation could contain: Chain word, pointer to answer semaphore, pointers describing a buffer, operation code, status, etc. The standard format is given in section 6.1.3.

The driver initiates the device registers and starts the device as needed.

A general outline of a driver might be:

```
driver:  if semaphore passive then
          begin stop semaphore
              loop on device interrupt level
          end else
          begin
              element:= address of first operation in semaphore queue
              action:= element. code
              case action of
              begin action 1: ---;
                  action 2: ---;
              end
              element. status:= status after action
              remove operation from semaphore
              start interrupter
              goto driver
          end
```

The following SLAM-code might perform the essential parts of the driver code:

```

a0:  ; w3 = semaphore addr
      ldr  w2    x3 j20      ; w2:= semaphore. state
      bew  w2 in 0    ,a2    ; if sem.state = 0 then
      ldr  w1 i 1          ; begin stop semaphore
      str  w1 x3 j20        ;
a1:  lce  w0 i          a1    ; loop
a2:                                ; end else
                                ; begin
      .
      .
      .
      .
      .
      ; w3 = semaphore addr ; remove operation from semaphore
      ldi  w2    x3 j20      ; w2:= next operation
      xsr  w2    x3 0        ; w2:= semaphore. state;
                                ; semaphore. state:= w2
      ist  w2 s x1 i1        ; start interrupter
      ldr  w0 i          a0    ; goto driver

```

The name conventions are explained in section 6.1.

The software timer is a coroutine, which is commissioned to return operations after a specified time interval.

The timer coroutine maintains internally the absolute time since the system was last reset. When the timer coroutine receives an operation from another coroutine, the absolute time for returning the operation is calculated from the actual absolute time and the specified time interval in the operation. The calculated absolute time for return is placed in the operation and the operation is chained on an internal queue according to increasing times for return.

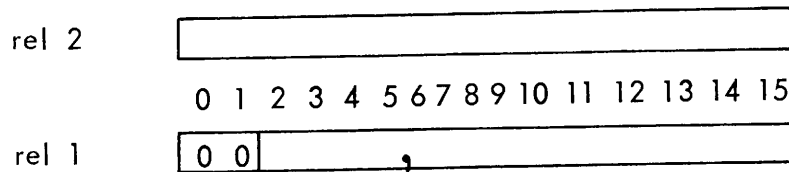
An operation in the internal queue is returned to the sender as a normal answer from a driver, when the internal absolute timer reaches the calculated time for return in the operation.

A timer operation has the format:

operation	+ j10	chain
	+ j11	answer semaphore
	+ j12	rel 2
	+ j14	rel 1
	+ j16	abs 2
	+ j13	abs 1

where rel 2 is time * 2^{-4}
 rel 1 is time * 2^{10}

which together makes a doubleword indicating the interval, when the operation is requested to be returned from the timer. If the time is measured in seconds, the comma is placed between bit 5 and bit 6 in rel 1.



Obs.: Bit 0 and bit 1 in rel 1 must be zero!

The shortest interval, which may be specified is:

$$2^{-10} \text{ sec.} \sim 1 \text{ mSec.}$$

The longest interval, which may be specified is:

$$(2^{19} - 1) \text{ sec.} \sim 58 \text{ days.}$$

Abs 2 and abs 1 constitute a doubleword where the timer stores the calculated absolute time for return.

Restrictions:

1. A coroutine must not call the functions "signal chained" and "regret" within one stretch, because in that case "regret" only functions as a normal "wait chained".
2. Both "simple regret" and "regret" to the timer must not be called by a coroutine running on a higher priority level than the timer coroutine.

A testoutput record may be produced in the following situations:

- a coroutine calls one of the monitor functions
- a driver starts the "interrupter"
- the system exits the monitor and enters a coroutine after having performed a monitor function
- the monitor processes an operation in the monitor queue
- the system is reset by means of the switches

Testoutputrecord.

4.1

The produced testoutputrecords have the format:

```
Record:    record size < 8 + type (header)
          + 2:    record word 1
          + 4:    record word 2
          .
          .
          .
          + record size
```

where record size is the number of bytes in the testoutputrecord exclusive the header

and type indicates the type of the record-contents.

The following survey gives the connection between type and contents:

<u>TYPE</u>	<u>CONTENTS</u>
<u>type 0 - interruptservice</u>	empty
<u>type 1 - exit from monitor</u>	testoutput descriptor for the running coroutine return address coroutine. w2

exit length	{	coroutine. w2.2
		coroutine. w2.4
		coroutine. w2.6
		.
		.

type 2 - processing of monitor queue

		operation address
signal chained length	{	operation. answersemaphore
		. top transferred
		. first of buffer
		. top of buffer
		.
		.

type 3 - signal chained

		operation address
signal chained length	{	operation. answersemaphore
		. top transferred
		. first of buffer
		. top of buffer
		.
		.

type 4 - signal

semaphore address

type 5 - wait chained

semaphore address

type 6 - wait

semaphore address

type 7 - pass

empty

type 10 - regret

semaphore address

address of information to regret action

type 11 - testoutput lost

empty

type 12 - reset

empty

type XX - <user defined type>

<user defined contents>

Testoutput Driver.

4.2

The production of testoutput is supervised by a testoutput driver in the monitor. Some external process (coroutine) must supply the testoutput driver with buffer operations with the format:

		.
		.
		.
		<buffer>
		.
		.
		.
operation	+ j10:	chain
	+ j11:	answer semaphore
	+ j12:	top transferred
	+ j14:	first of buffer

- first of buffer must be the address of the first free byte + offset (= j35).
- top transferred is updated by the testoutput driver to the address of the first free byte + j35, when the operation is returned to the sender.

When the testoutput records are produced they are placed in the buffer, and the testoutput driver returns the buffer operation when the buffer is filled.

If the testoutput driver semaphore queue runs empty, the testoutput driver stops the semaphore. The testoutput records produced until a new buffer operation arrives are placed in a dummy buffer, and they are lost.

When a new buffer operation arrives, a testoutputrecord of type 11 is placed in the buffer as the first record in the buffer.

In order to utilize the testoutput system some processes must be implemented which supply the testoutput driver with buffer operation and transform the contents of the buffer to some output device. For an actual implementation see appendix A.

Testoutput Options.

4.3

The testoutput record production is determined by the following monitor options (see also section 6.1.2).

i2 - the testoutput mask is a bitpattern which controls the testoutput at assemble time in the following way:

- 1 < 0 testoutput at interrupt service (type 0)
- 1 < 1 - - exit from monitor (type 1)
- 1 < 2 - - processing of monitor queue (type 2)
- 1 < 3 - - signal chained (type 3)
- 1 < 4 - - signal (type 4)
- 1 < 5 - - wait chained (type 5)
- 1 < 6 - - wait (type 6)
- 1 < 7 - - pass (type 7)
- 1 < 8 - - regret (type 10)
- 1 < 11 - - simple regret (type 10)

i4 - is the maximal number of bytes in a testoutput record exclusive the header

i5 - is the standard testbits with the same interpretation as the bits in the testoutput mask. i5 is used as the first testbits until changed by the switches (see page 4-6).

i6 - signal chained < 5 + exit.

Signal chained: is the number of bytes in an operation to be output when a testoutputrecord of type 2 or type 3 is produced.

Exit: is the number of bytes to be output when a testoutputrecord of type 1 is produced, starting with coroutine. w2.2 (see page 4-2). This is normally the words in an operation.

i7 - the coroutine mask is a bitpattern consisting of 8 bits which makes it possible to divide the coroutines in 8 classes. A coroutine in a class produces testoutput if the corresponding bit in the coroutine mask is on.

If the coroutines in a class produce testoutput, this is controlled by the testbits (i5) and the testoutput mask (i2).

The options i2, i5, i6, i7 must be defined at assemble time. i5, i6, and i7 may be redefined at run time by the following procedure:

When the running coroutine is the dummy coroutine, the following algorithm is performed:

```
a0:  while switch [0 : 15] <> 0 do goto a0
a1:  while switch [0] = 0 do goto a1
a2:  if switch [1] = 0 then
      begin update type 2 limit
          goto a0
      end
a3:  if switch [2] = 0 then
      begin if switch [3] = 1 then
          begin update testbits
              goto a0
          end
          if switch [4 : 15] <> 0 then goto a0
              else goto a5
      end
      end
```

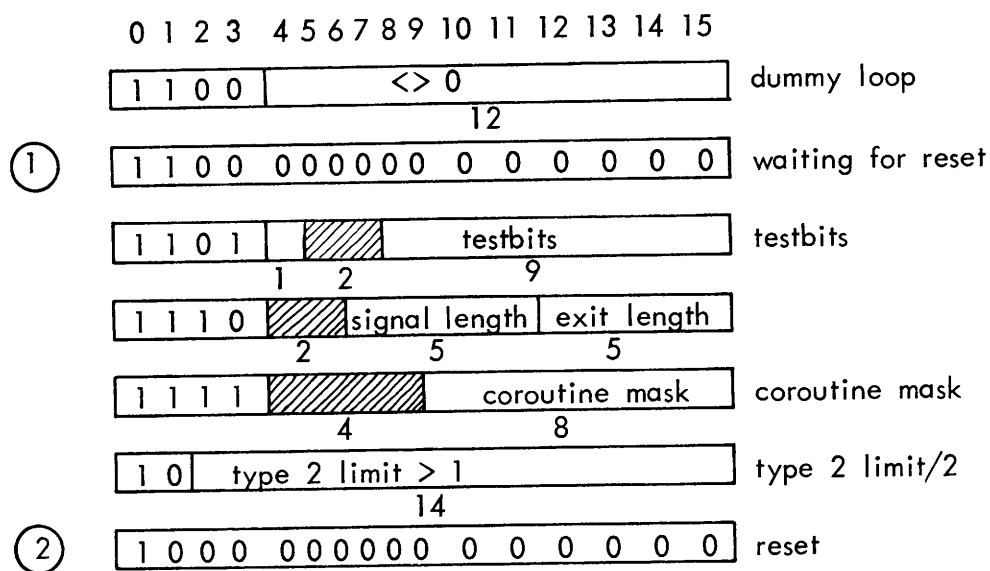
```

a4:  if switch [3] = 0 then
      update signal length and exit length
      else
      update coroutine mask
      goto a0
a5:  generate reset record (type 10) and signal (reset semaphore)
a6:  while switch [1] = 1 do goto a6
      reset

```

Type 2 limit defines an interval $[0 : \text{type 2 limit}]$ where a testoutput record of type 2 is suppressed if the address of the answer semaphore lies in this interval.

The following figure gives a survey how the switches may be used to redefine the wanted options:



In order to reset the system ① must be performed before ② .

The wait situation at label a6 may be utilized to delay the testoutput recordproduction just before the whole system is actually reset. See appendix A for a system which uses this facility.

How a coroutine is mapped on a class is described in section 6.1.3.

- a. RCSL: 52-AA316 RC 3500 Reference Manual
- b. RCSL: 52-AA059 RC 3500 General Purpose Controller,
General Information
- c. RCSL: 061 RC 3500 Assembler Manual

Names, Functions, Options, Formats.

6.1

A process in a program system using MONITOR 1 couples to the monitor via the SLAM-names *h*, *i*, and *j*. The *h*-names are used for entries in the monitor, the *i*-names are used for monitor options, and the *j*-names are used for offsets in the standard data structures.

h-Names, Monitor Functions.

6.1.1

The *h*-names include entries to the monitor functions and give access to other system variables.

The monitor function may be called from coroutines in the following general way:

1. load *w1* and *w2* as required by the monitor function
2. ldr *w3* *i* *a0* ; *w3*:= return address
ldr *w0* *i* *hi* ; goto monitor function *hi*

Upon return to *a0*, *w3* contains the address of the coroutine description, which facilitates reentrant programming.

h0: signal chained

Call: *w1* = address of chained semaphore
 w2 = address of operation
 w3 = return address

Return: *w1* and *w2* undefined
 w3 = coroutine address

If the semaphore is open, passive, or stopped the operation is placed in the semaphore queue. If the semaphore is stopped, the monitor jumps to the start action.

If the semaphore is closed, the first coroutine in the semaphore queue is removed from the semaphore queue and activated, according to its priority with $w2$ = operation address.

h1 = signal

Call: $w1$ = address of simple semaphore
 $w3$ = return address

Return: $w1$ and $w2$ undefined
 $w3$ = coroutine address

If the semaphore is open or passive, two is added to semaphore. state.

If the semaphore is closed, the first coroutine in the semaphore queue is removed from the semaphore queue and activated according to its priority.

h2: wait chained

Call: $w1$ = address of chained semaphore
 $w3$ = return address

Return: $w1$ undefined
 $w2$ = address of operation
 $w3$ = coroutine address

The calling coroutine is removed from the active queue.

If the semaphore is open, the first operation in the semaphore queue is removed from the semaphore. The operation address is placed in $w2$ and the coroutine is activated according to its priority.

If the semaphore is passive or closed, the coroutine is placed in the semaphore queue as the last element.

It is forbidden to call "wait chained" if the semaphore is stopped!

h3: wait

Call: w1 = address of simple semaphore
 w3 = return address

Return: w1 and w2 undefined
 w3 = coroutine address

The calling coroutine is removed from the active queue.

If the semaphore is open, two is subtracted from semaphore. state, and the coroutine is activated according to its priority.

If the semaphore is passive or closed, the coroutine is placed in the semaphore queue as the last element.

h4: pass

Call: w3 = return address

Return: w1 and w2 undefined
 w3 = coroutine address

The calling coroutine is removed from the active queue and activated again according to its priority.

h5: return from start action

When an operation arrives at a semaphore, which is stopped, the operation is chained on the semaphore and the monitor jumps to the start action as described in the semaphore (semaphore. start action) with

 w1 = semaphore address
 w2 = operation address

The startaction code in the receiving driver must start the driver and return to h5 in the monitor.

h6: testoutput

Call: w1 = address of first word
 w2 = number of words * 2 < 8 + type
 w3 = return address

Return: w1 and w2 undefined
 w3 = coroutine address

A testoutput record identified by type is produced. Type must be different from the types described in section 4.1.

Number of words * 2 must be less than the option i4 (see page 6-8).

The content of the testoutputrecord will be the record defined by "address of first word" and "number of words".

h7: simple regret

Call: w1 = information to regret action
 w2 = semaphore address
 w3 = return address

Return: w1 and w2 undefined
 w3 = coroutine address

A coroutine has regretted a "signal chained" to the semaphore described in w2 and wants the operation back.

When the coroutine makes the call "simple regret", the coroutine is removed from the active queue, and the monitor jumps to the regret action as described in the semaphore (semaphore. regret action) with

 w1 = information to regret action
 w2 = semaphore address

The regret action must secure that one or more operations are released and returned to a semaphore according to an agreement between the coroutine which calls regret and the process which has received the operation.

If the process which has received the operation has passed the operation on to another process, the regret action must call the monitor function regret next (h13), see page 6-7.

The regret action returns to h9 with

w1 = semaphore address.

This semaphore is the semaphore where the operations are returned according to the agreement between sender and receiver. Information used in this agreement may be passed as the "information to regret action" in the call of the regret function.

The calling coroutine is activated again according to its priority.

h8: regret

Call: w1 = information to regret action
 w2 = semaphore address
 w3 = return address

Return: w1 undefined
 w2 = address of an operation
 w3 = coroutine address

Regret works like "simple regret" except that the calling coroutine waits for the semaphore as specified in w1 when the regret action returns to h9 (see below).

h9: return from regret action

The regret action must return to h9 with

w1 = semaphore address.

See further under simple regret (h7).

h10: current coroutine

The running coroutine may reestablish its coroutine description address through this variable:

```
ldr w3 h10; w3:= coroutine address
```

This facilitates addressing of private variables.

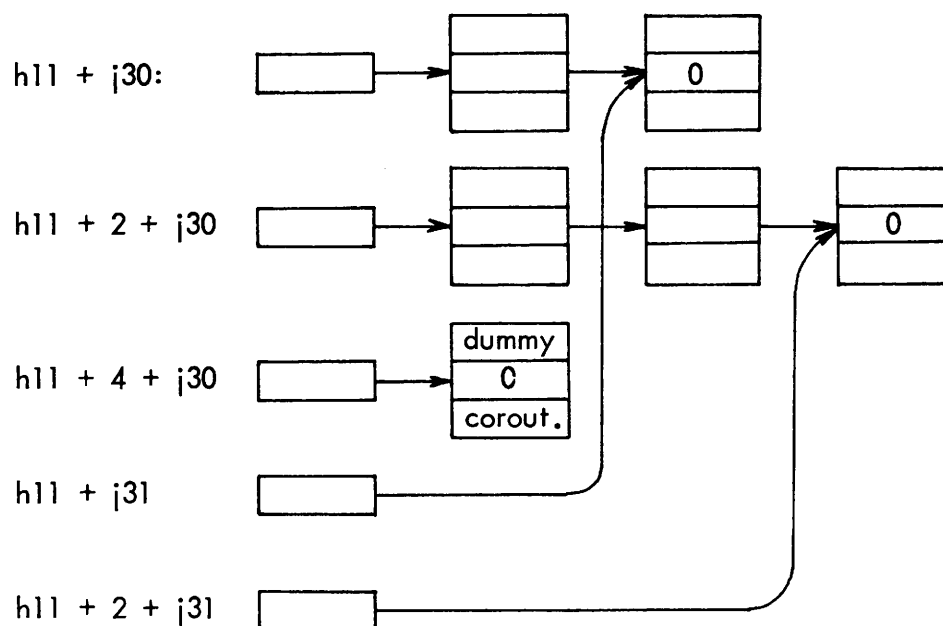
h11: active queue

is the address of the highest priority level in the active queue.

Suppose the system has two priority levels. When the system resets, the coroutine description must be linked together on the priority level as defined in the coroutine description by the following reset code in the coroutine:

```
; w3 = coroutine address  
ldr w1 x3 j1 ; w1:= priority  
sti w3 x1 j31 ; priority. last. chain:= coroutine  
str w3 x1 j31 ; priority. last:= coroutine
```

After reset the situation might be:



(See further under system initialization, section 7).

h12: first of free store

is the address of the first free storage word which may be used for buffers etc.

h13: regret next

Call: w1 = information to regret action
 w2 = semaphore address 1
 w3 = semaphore address 2

Return: to h9 with
 w1 = semaphore address 2

Regret next are called from a regret action associated a process which has passed on the requested operation(s).

Semaphore address 1 is the semaphore from which one or more operations are wanted back. Semaphore address 2 is the semaphore where the operations are returned according to the agreement between sender and receiver.

h14: timersemaphore address

is the address of the semaphore where timer operations are placed by a call of

signal chained (h14, timerop, return)

h15: top of program system

is a label which must be appended all program systems using the Monitor 1. Thus a program system has the structure:

```

b.   f127, g127                      ; begin outermost block
      .
      .
      .
b.   i12                              ; begin options
      .
      .
```

```

      .
b.   h15, i50                      ; begin monitor entries
      .
      .
      .                      ; and formats
      <monitor>
      <testoutput system>
      <timer>
      <process 1>
      <process 2>
      .
      .
      <process n>
h15:
      e.
      e.
      e.

```

i-Names, Options.

6.1.2

- i0 - is the interrupt level for the monitor and must equal 0.
- i1 - is the interrupt level for the monitor subprocess "the interrupter".
i1 = 31 is normally used.
- i2 - the testoutput mask (see section 4.3).
- i3 - indicates the number of priority levels in the scheduling system.
- i4 - maximal number of words * 2 in a testoutput record exclusive
the header (see section 4.3).
- i5 - standard testbits (see section 4.3).
- i6 - signal chained < 5 + exit.

Number of bytes in each operation to be output during testoutput
record production.

i7 - standard coroutine mask.

i7 = 127 is normally used (see section 4.3).

i8 - the monitor is loaded to i8.

i8 = 256 is used as a standard.

i9 - the monitor is moved to i9 before reset.

i9 = 0 is used as a standard.

The monitor must be situated in an area of the store, where the monitor addresses have the same sign.

i10 - interrupt level for testoutput device.

i11 - the timerinterruptlevel.

i11 = 1 is used as a standard.

If i11 = 0 the timer is excluded.

i12 - timerinterval.

i12 = 0 is used as a standard.

The absolute time is increased with $1 < i12$ every time the timer oscillator gives an interrupt.

i13 - is the number of buffer operations in the testoutput-output system and must be greater than 2.

i14 - is the number of bytes the buffer in a bufferoperation sent to the testoutput device exceeds the size of the biggest legal testoutput-record. This option is used by the coroutine in the testoutput-output system (see appendix A) to control the buffer size.

i15 - version number of the monitor.

Coroutine Description

A coroutine is described in a table with the following format:

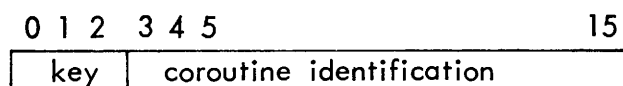
```

"coroutine" + j6:  testoutput descriptor
                + j1:  priority
                + j0:  chain
                + j2:  w0
                + j3:  w1
                + j4:  w2
                + j5:  w3
                + j5+2 ff:  further private variables as needed

```

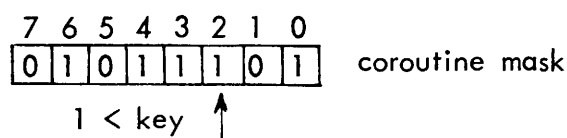
"Coroutine" is the address of the second byte in chain, so all coroutine addresses are odd numbers.

The testoutput descriptor word has the format:



key - may take the values 0-7 and defines the class to which the coroutine belongs in the testoutput system (see section 4.3).

The connection between the key and the coroutine mask is:



In this example the coroutine will produce testoutput when running. The testoutput produced is determined by the test-bits (i5).

If the corresponding bit in the coroutine mask was off, no testoutput would be produced.

status - may be used to return the status of a device, when a device returns the operation.

code - may be used to define to the receiver which action should take place when the operation is processed (see f.ex. section 2.6).

Semaphores

A chained semaphore has the format:

```
semaphore + j23:  regret action
           + j20:  state
           + j21:  last
           + j22:  start action
```

semaphore - is the address of state.

regret action - is the address of a piece of code which is executed if a process calls the functions "regret", "simple regret", or "regret next".

The regret action address is only required if there is an agreement between the receiver and senders associated the semaphore.

start action - is the address of a piece of code which is executed if the semaphore is stopped. The start action may be omitted if the semaphore is never stopped.

state, last - defines the semaphore and takes the values:

chained semaphore	semaphore. state	semaphore. last
open	first operation in queue	last operation in queue
passive	0	semaphore address
closed	first coroutine in queue	last coroutine in queue
stopped	1	irrelevant

A simple semaphore has the format:

```
semaphore + j20:  state
           + j21:  last
```

state, last - defines the semaphore and takes the values:

simple semaphore	semaphore. state	semaphore. last
open	number of signals + 2	irrelevant
passive	0	semaphore address
closed	first coroutine in queue	last coroutine in queue

Active queue

A queue associated a priority level is controlled by:

priority + j30: first
+ j31: last

first, last take the values:

Priority level	priority. first	priority. last
empty	0	priority address
not empty	first coroutine in queue	last coroutine in queue

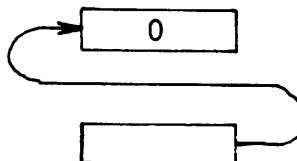
As an example of the active queue see page 6-6.

Queues

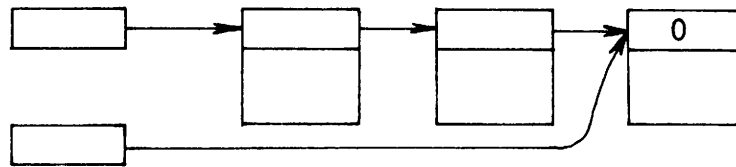
All queues are linked with one queue element containing the absolute address of the next. The chain word in the last element contains a zero.

A queue is defined by two words in the store as illustrated on the figure:

1. Empty queue:

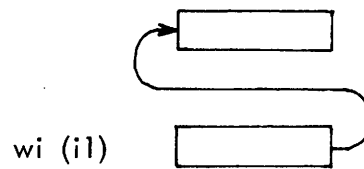


2. Not empty queue:

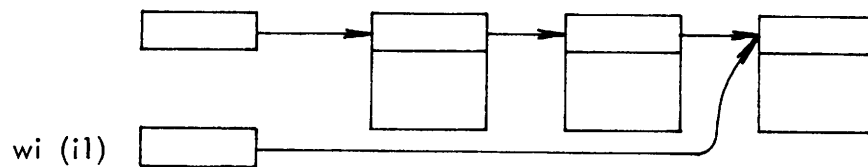


The monitor queue is an exception, because the second "word" is a working register on interrupt level $i1$:

1. Empty monitor queue:



2. Not empty monitor queue:



After autoloading of a program, the system initializes in the following steps:

1. The program is moved to cell i9 and the system starts the interruptlevel i1.
2. Different testpoints in the monitor are updated according to the monitor standard options.
3. The monitor variables are reset, whereafter the system chains through the processes included in the system and executes the reset code in every process. This reset code in a process must reset all private variables because it shall be possible to reset the entire system at any moment independent of the current state of the processes. E.g. the processes must reset their semaphores, and the coroutines must link on their priority queue (see examples in appendix B).
4. When the system reaches h15, it returns to the monitor.
(ldr w0 i <address> is placed in h15 and h15 + 2).
5. The chain word in the last element in all priority queues is reset to zero and the monitor level is started in scanning the priority queues.
6. The "interrupter" on interruptlevel i1 is initialized.
7. Interruptlevel i1 is cleared by an lce-instruction, which implies that all lower interrupt levels are enabled.
8. Now the system runs!

The driver on the highest interrupt level is started and runs until the driver in question clears the interruptbit of the associated interruptlevel. The system "falls" down on the next interruptlevel which is not cleared.

When no drivers are running, the coroutines on interrupt 0 are scheduled by the monitor, which makes the coroutines running according to the start addresses in the coroutine description (coroutine. w0).

Size.

8.1

The size of the monitor exclusive the testoutput and the timer process is

650 bytes

The size of the monitor inclusive all testoutput and exclusive the timer process is

1638 bytes

The size of the monitor inclusive testoutput and the timer process is

2052 bytes

See appendix A for the size of a system utilizing the testoutput system incorporated in the monitor.

Execution Times.

8.2

Time in μS . of the monitor functions:

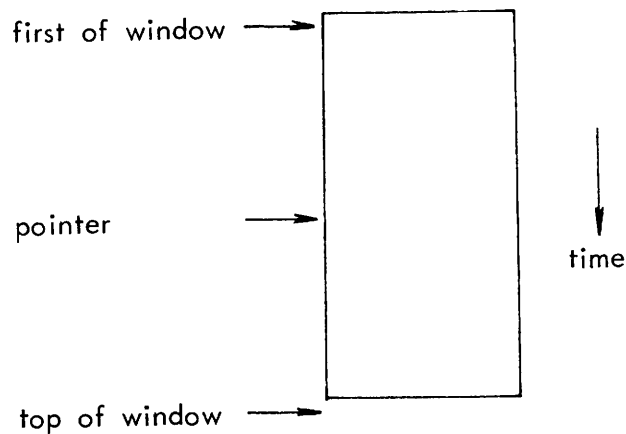
	State of involved semaphore:			
	<u>passive</u>	<u>open</u>	<u>closed</u>	<u>stopped</u>
interrupt	$297 + P$	$297 + P$	$344 + P$	$328 + P + \text{startaction}$
signal chained	$185 + P$	$185 + P$	$231 + P$	$216 + P + \text{startaction}$
signal	$139 + P$	$139 + P$	$193 + P$	
wait chained	$217 + P$	$261 + P$	$220 + P$	
wait	$208 + P$	$252 + P$	$212 + P$	
pass		$236 + P$		
regret		$314 + P + \text{regret action}$		
simple regret		$327 + P$		
nested regret		$66 + (\text{nested}) \text{ regret action}$		
$P = 9 \times (\text{pri} - \text{h11})$ pri is priority of current coroutine at call				
signal - wait chained: 447 μS .				

This appendix is a description of a coroutine and a device, which utilizes the testoutput system incorporated in the Monitor 1.

The coroutine supplies the testoutput driver in the monitor with buffer-operations and outputs the content of the received testoutput records line by line on an RC 810 display unit as controlled by the operator.

User Interface

As seen from the user the produced testoutput records constitute a "window" of events together with a pointer as illustrated on the figure:



The pointer indicates the last testoutput record shown on the display unit. The position of the pointer may be controlled by the user in the following way:

1. Press the line-interrupt button (SND):

The next testoutput record will be displayed on the screen following the last displayed record.

The pointer in the window is advanced one record.

2. Press the page-interrupt button (SND):

=====

A full screen will be displayed starting on the first line on the screen. The pointer in the window is advanced to the last record displayed on the screen.

3. Type $\left\{ \begin{matrix} + \\ - \end{matrix} \right\}_1 \left\{ \text{digit} \right\}_0^4$ (in the leftmost corner) and press the page-interrupt button (SND):

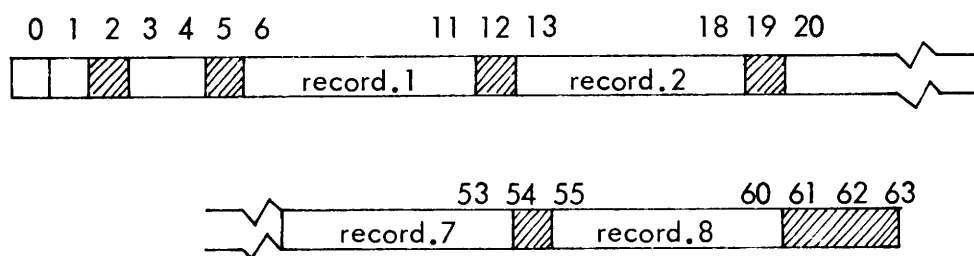
=====

The pointer in the window is moved forward or backward a number of records relative to the current position as indicated by the typed signed integer.

From the new position a full screen will be displayed starting on the first line on the screen.

The pointer in the window is advanced to the last record displayed on the screen.

A testoutput record will be output as a line on the screen with the format:



The hatched fields are always spaces.

[0] is alternately SP and * in order to distinguish between the screen pages.

[1] : $\left\{ \begin{matrix} \text{SP} \\ + \\ - \end{matrix} \right\}$ has the following interpretation:

+ - the user has requested a screen display starting at a test-output record which was produced "before" the first record in the window.

The record displayed with the + is the first record in the current window.

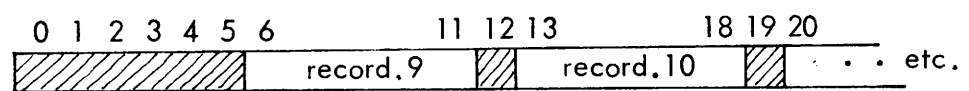
- - some testoutput records between this record and the preceding record displayed are lost.

SP - this testoutput record follows the preceding record in time. No testoutput records are lost between the two records displayed.

[3 : 4] - is the type of the testoutput record displayed as explained in section 4.1.

[6 : 11] , [13-18] etc. is the content of the testoutput record displayed as unsigned octal integers.

If the record may not be displayed on one line, the display is continued on the next lines with the format:



If the dummy coroutine is brought in the state "waiting for reset" by means of the switches (see section 4.3), the window is "stopped" with a testoutput record of type 12_8 as the last displayed record. The rest of the screen is filled with spaces and the user may request display of records before the testoutput record of type 12_8 , before actually resetting the system.

Monitor Interface

The buffer operations sent to the testoutput driver have the format:

•
•
•

<buffer>

operation + j10: chain
+ j11: answer semaphore
+ j12: top transferred
+ j14: first of buffer
+ j16: number of records
+ j13: forward pointer
+ j15: backward pointer

buffer - contains the testoutput records as placed by the monitor.

answer semaphore - is the address of the coroutine semaphore.

top transferred - is the address + j35 of the first free byte in the full buffer as updated by the monitor.

first of buffer - is the address + j35 of the first byte in the buffer.

number of records - is the number of testoutput records in the buffer.

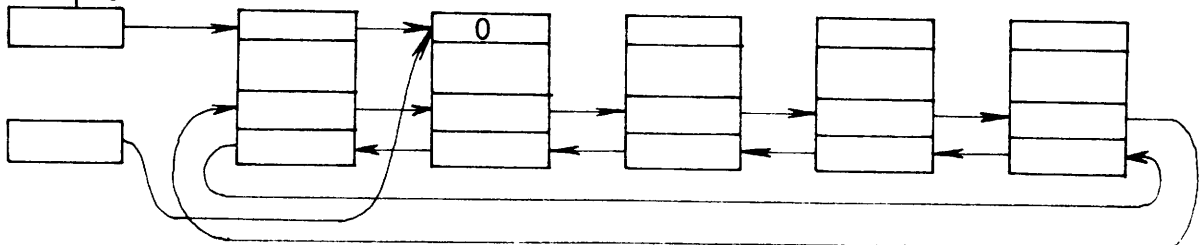
The coroutine calculates this variable, when the operation is received from the monitor and it is used when the coroutine moves the pointer in the window.

forward, backward pointer - is used to link the available buffer operations in a cyclic way.

Function

During reset two buffer operations are placed in the testoutput driver semaphore queue and the situation is:

testoutput driver
semaphore



The window is empty and

first of window = top of window.

Every time a full buffer operation is received from the monitor, the buffer is appended the window and "top of window" is updated, where- after the next buffer operation in the list is signalled to the testoutput driver. If the monitor reaches "first of window", which may happen because the list is cyclic, the testoutput records in this buffer are removed from the window and "first of window" is moved to the next buffer. If the window pointer lies in the buffer which was removed, the pointer is moved to the next buffer like "first of window" and a - (minus sign) appears in front of the next testoutput record displayed on the RC 810 screen.

The RC 810 Driver

has one input semaphore and one output semaphore.

The driver outputs one line at a time and fills the line with spaces, if the line is not complete.

The first position on each line is used to indicate the change from one screen picture to the next.

See page A-2 for the format of the lines.

The driver works according to the following algorithm:

```
a0:  open keyboard and wait interrupt
      disable keyboard
      read status
      if pageinterrupt then
      begin reset address count
          input to input buffer
          signal inputoperation
          if inputsemaphore passive then
```

```

startentry:  begin stop semaphore
              loop
            end

inputentry:  linecount:= - 16
            globalcount:= - 16
            end else
            begin
              linecount:= - 1
            end

a1:  if outputop home then
      signal output operation
      if outputsemaphore passive then
        begin stop semaphore
          loop
        end

outputentry:
      output one line
      globalcount:= globalcount + 1
      linecount:= linecount + 1
      if globalcount = 0 then
        begin reset address count
          globalcount:= - 16
          exchange start characters
        end
      goto if linecount = 0 then a0 else a1

```

Size

The size of the testoutput coroutine and the
RC 810 driver is

1324 bytes

The following is some examples how a coroutine in several incarnations and a driver may be structured using the assembler language SLAM.

; the coroutine in this example is reentrant in two incarnations
; and may be structured like this

```

b. e29                                ; coroutine e-block
    ldr w0 1                          e0 ; goto reset

; e-names definitions and dataarea for first incarnation
e1=3<13+8.17                        ; testoutputdescriptor1
e2=h11+2                            ; priority1
e3=k-j6                             ; first coroutine address
e1,e2,0,r.5                         ; coroutine description1
f12: 0,0                            ; semaphore1
k-j6+2                              ; reset link

; e-names definitions and dataarea for second incarnation
e1=4<13+8.20                        ; testoutputdescriptor2
e2=h11                              ; priority2
e1,e2,0,r.5                         ; coroutinedescription2
f16: 0,0                            ; semaphore2
0                                   ; reset link

b. b10,c3,d1                        ; begin bcd-block
; d-names definitions
d0=9                                ; semaphore offset
d1=d0+4                            ; reset link offset

.
.
.
b. a2                                ; begin coroutine code
.
.
b0:
.
.
e.                                    ; end coroutine code

b. a0                                ; reset
e0: ldr w3 1      e3                ; w3:=first coroutine
a0: ldr w1 1      b0                ; coroutine.w0:=start
    str w1      x3 j2
    ldr w1      x3 j1                ; place coroutine in active queue
    sti w3      x1 j31
    str w3      x1 j31
    ldr w1 1      0
    str w1      x3 d0+j20            ; semaphore.state:=0
    ldr w2 1      x3 d0
    str w2      x3 d0+j21            ; semaphore.last:=semaphore
    ldr w3      x3 d1                ; w3:=next incarnation
    bew w3 in    0                  ,a0

e.                                    ; end reset
e.                                    ; end bcd-block
e.                                    ; end e-block

```

; the driver in this example has one semaphore and may
; be structured like this

```

b. e10                                ; driver e-block

f23=e0                                ; driver semaphore address
e1=g45                                ; driver interruptlevel
e2=g46                                ; device number

b. b10,c5,d3                          ; driver bcd-block
    ldr w0 1                          b10; goto reset

    b0                                ; regret action
e0:  0                                ; semaphore.state
    0                                ; semaphore.last
    b1                                ; start action

b. a10                                ; driver code
    .
    .
    .
b2:  .                                ; start:
    .
    .
    .
e.                                     ; end driver code

b. a3                                  ; regret action
b0:; w1=information w2=driversemaphore
    < regret action code >
    ; w1=return semaphore
    ldr w0 1                          h9 ; goto return from regretaction
e.                                     ; end regret action

b. a2                                  ; start action
b1: ; w1=driversemaphore w2=operation
    ldr w3 1      b2
    1st w1 d  x3 e1                    ; w3(e1):=driversemaphore
    1st w2 s  x2 e1                    ; w2(e1):=operation
    1st w3 e  x0 e1                    ; w0(e1):=start
    ldr w0 1                          h5 ; goto return from start action
e.                                     ; end start action

b10: ldr w3 1      e0                  ; reset
    ldr w1 1      1                    ; semaphore.state:=stopped
    str w1      x3 j20
e.                                     ; end bcd-block
e.                                     ; end e-block

```