TITLE: CR80 AMOS KERNEL
PRODUCT SPECIFICATION

DOCUMENT NO: CSS/302/PSP/0008

PREPARED BY: Erik Kliim Hansen
Jørgen Høg

APPROVED BY: Jørgen Høg

AUTHORIZED BY: Jørgen Høg

DISTRIBUTION:

ISSUE:	1	2	3					
DATE:	790827	810303	820601					

400-571-2

CR80 AMOS KERNEL PRODUCT SPECIFICATION	sign/date EKH/820601	page i
	repl EKH/810303	project

PAGE RECORD AND ISSUE LOG.

PAGE	ISSUE 1	2	3	4	5	6	7	8
01								
02								
03								
04								
05								
06								
07								
08								
09								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								
22								
23								
24								
25								
26								
27								
28								
29								
30								
31								
32								
33								
34								
35								
36								
37								
38								
39								
40								
41								
42								
43								
44								
45								
46								
47								
48								
49								
50								
51								
52								
53								
54								
55								
56								
57								
58								
59								
60								
61								
62								
63								
64								
65								
66								
67								
68								
69								
70								
71								
72								
73								
74								
75								
76								
77								
78								
79								
80								
81								
82								
83								
84								
85								
86								
87								
88								
89								
90								
91								
92								
93								
94								
95								
96								
97								
98								
99								
100								

ISSUE	DATE	PREPARED BY	APPROVED BY	AUTHORIZED BY
1	790827	JHØ	JHØ	JHØ
2	810303	JHØ	JHØ	JHØ
3	820601	EKH	JHØ	JHØ

PAGE RECORD AND ISSUE LOG.

PAGE	ISSUE 1	2	3	4	5	6	7	8
101								
102								
103								
104								
105								
106								
107								
108								
109								
110								
111								
112								
113								
114								
115								
116								
117								
118								
119								
120								
121								
122								
123								
124								
125								
126								
127								
128								
129								
130								
131								
132								
133								

PAGE	ISSUE 1	2	3	4	5	6	7	8
134								
135								
136								
137								
138								
139								
140								
141								
142								
143								
144								
145								
146								
147								
148								
149								
150								
151								
152								
153								
154								
155								
156								
157								
158								
159								
160								
161								
162								
163								
164								
165								
166								

PAGE	ISSUE 1	2	3	4	5	6	7	8
167								
168								
169								
170								
171								
172								
173								
174								
175								
176								
177								
178								
179								
180								
181								
182								
183								
184								
185								
186								
187								
188								
189								
190								
191								
192								
193								
194								
195								
196								
197								
198								
199								
200								

ISSUE	DATE	PREPARED BY	APPROVED BY	AUTHORIZED BY
1	790827	JHØ	JHØ	JHØ
2	810303	JHØ	JHØ	JHØ
3	820601	EKH	JHØ	JHØ

LIST OF CONTENTS

Page

K

Page

K

LIST OF FIGURES

K

1. SCOPE

The purpose of this document is to describe the CR80 AMOS MONITOR KERNEL.

The AMOS computer program configuration items described in this document are

- CSS/302, CSS/303 — Kernel
- CSS/360 — Root including RTC and memory manager
- CSS/306 — Idle process
- CSS/308 — Init program
- CSS/361 — Buffer allocation procedures
- CSS/316 — Double precision mul/div.

The KERNEL is the lowest level of CR80 AMOS system software layers. The KERNEL implements processes, CPU management, inter process communication and the lowest level of I/O device handling: Interrupt handling.

1.1 Organization of Document

The document contains in section 3 a description of the concepts used in the Kernel, the functions performed by the Kernel and the general structure of the Kernel. In section 4 a concise interface description is given of all Kernel functions. Section 5 lists the limitations pertinent to the Kernel. Section 6 and 7 contains practical information concerning compilation and system generation.

In section 8 key performance figures are given for the Kernel.

Finally, appendixes A, B and C exhibit listings of source files which contain definitions pertinent to the Kernel. These files should be used as part of the source text for CR80 assembler programs which make use of the Kernel functions.

2. APPLICABLE DOCUMENTS

2.1 CR80 MINI COMPUTER HANDBOOK
CSD/HDBK/0082

2.2 P. Brinch Hansen
Operating Systems Principles
Prentice Hall, N.J.

2.3 European Purdue Workshop - TC8
Real Time Operating System Guidelines.

2.4 CR80 AMOS, I/O SYSTEM
PRODUCT SPECIFICATION
CSS/006/PSP/0006

2.5 CR80 AMOS, SYSGEN
USER'S MANUAL
CSS/121/USM/0023

2.6 CPU-SCM, CR8002 M Product Specification
CSD/005/PSP/0049

2.7 CPU-SCM, LR8002 M /011P-/00
XAMOS/CR801 Application Product Specification
CSD/005/PSP/0091

3. KERNEL REQUIREMENTS

The purpose of the AMOS Kernel is to implement multiprogramming on the CR80 multiprocessor.

The AMOS Kernel fulfils the following requirements:

- implementation of software processes
- communication between processes
- synchronization of processes
- CPU management
- I/O interrupt handling
- dedication of processes to specific CPUs.
- support of CR80 computers with up to 512 kbyte of main memory and 8 CPUs.

The second last requirement arises from the CR80 architecture (see ref. 2.1) which allows CPUs to have private 'subbusses' connecting the CPU to a part of the main memory. CPUs having such a subbus should primarily execute programs and operate on data accessible via its subbus.

Although a given process is dedicated to execute on a single processor, the existence of more than a single CPU is shielded from the programmer using the Kernel. There is no difference between the communication taking place between two processes executing on the same CPU and that taking place between two processes executing on different CPUs.

The primitives for communication between processes are based on the concept of messages and answers described in ref. 2.2.

Three different types of messages/answers have been implemented:

messages - answers,
system messages - system answers,
path messages - path answers.

The mechanics for these three types are similar. Each type, however, has its own eventqueue, with the advantage of efficient separation of messages/answers used for different purposes.

The intended use of system messages/answers is communication with peripheral device drivers (via the AMOS I/O system).

The Kernel consists of a Kernel program, a Kernel context*) and an I/O context. The Kernel context and the I/O context share a number of variables. The most important of these are:

- interrupt tables
- process control blocks
- CPU control blocks
- Critical region control blocks

---

*) The word context is used to mean a set of registers (CPU resident or saved). This is the CR80 HW process concept.

The Kernel program is designed to be modular. It is structured as a nucleus part which contains basic procedures for handling process control blocks and CPU control blocks, and a number of submodules each containing procedures for a separate class of eventtypes.

The AMOS Kernel supports un-mapped CR80 CPU's with basic instruction set as defined in ref. 2.6, and CPU's with extended instruction set to execute programs in more than 64 K word of memory (XAMOS).
The CPU type is invisible to the programmer.

3.1 Invokation of the KERNEL

The Kernel is invoked

(a) when a MON instruction with proper argument is executed,

(b) when an I/O interrupt is received by a CPU,

(c) when a CPU interrupt is received

(d) when a local interrupt is generated (timer action, trap, timeout during addressing, parity error and bound violation),

The action taken when causes (a) or (d) occur are similar. A branch to a proper monitor procedure is taken. This may or may not generate a programmed context switch (saving of current registers, and loading of a new set of registers) to the Kernel context. (This always happens in case (d).)
The context switch is performed as follows:

1. The execution level is incremented, and program memory section 0 is selected by firmware (XAMOS only).
2. The current registers are saved at the normal context save area (relative data locations - 2 through 13) and thereby automatically disabling interrupt handling in the current CPU.
3. A function code is loaded into register 3. (Register 3 never holds a user defined call parameter.)
4. The PCB index (rel. loc.-3) is loaded into register 5.

5. The memory section (page) bits in the Process Status Word are set to Ø.

6. A hardware semaphore (the Kernel Semaphore) is reserved, or a busy waiting is performed until it can be reserved.

7. The current registers Ø through 6 are transferred to a Kernel parameter area.

8. The Kernel context is loaded.

9. It is checked that the current level is not greater than 16. If it is, the process is terminated.

10. The proper action is taken according to the function code loaded in step 2.

Steps 3 through 7 are called 'enter Kernel'.

The alternate possibility is that no context switch occurs. In the former case the Kernel subroutine invoked is called a <u>Function</u>, in the latter it is called a <u>Procedure</u>.

When events (b) and (c) occur, the CPU firmware will perform a context switch to the I/O context. The further processing is described in section 3.10.

Events of type (c) are reserved for exclusive use by the Kernel. CPU interrupts are used to transfer I/O interrupts from one CPU to another CPU.

3.2 Parameter Checking

The parameters used when calling the Kernel are primarily of two kinds:

- indices to be used in Kernel tables
- addresses relative to the calling process.

The first kind of parameters are checked to be within their appropriate boundaries, typically ranging from Ø to a maximum value.

The second kind of parameters are checked to lie within the memory area allocated to the process (more specifically the addresses are checked to be lower than the SIZE of the process).

In connection with creation of processes, however, absolute addresses are sent to the Kernel for use in connection with initialization of a context area.

As there is no simple way of validating these, the access to calling Create process should be restricted (refer to sections 3.3.4 and 4.9.).

3.3 Processes

A process is defined as an incarnation of the data transformations obtained by execution of a program. A program is defined as a collection of machine instructions, which can be executed within a single context (i.e. without change of BASE and PROG registers (see ref. 2.1)). This definition of a program makes a monitor procedure (a subroutine to which transfer is performed by execution of the MON instruction) potential part of many different programs (this also emphasises the rule, that the result of execution of a monitor procedure must be independent of the exact value of PROG).

3.3.1 Process Control Blocks

For management of processes, the Kernel has a pool of process control blocks (PCB). This pool is created at system initialization time. All processes but two (the KERNEL PROCESS and the I/O PROCESS) are associated with a PCB.

The pool of PCBs resides in memory section Ø (addresses lower than 64K) or in section 1 (addresses from 64 K to 128 K).

The exact layout of a process control block is shown in fig. 3.3.1.a.

Addressing of PCBs is performed indirectly through a PCB index table (fig. 3.3.1.b).

The PCBs are kept on a linked list (PCB item SCHAIN).

LOCATION	NAME	CONTAINS
0	SCHAIN	Link to next PCB
1	SNAME	Process name
2		Process name
3		Process name
4	SACCESS	Capabilities (3.3.4)
5	SLOGPCB	PCB index value
6	SPARENT	Link to PCB of parent process
7	SCHILD	link to PCB of child process
8	SNEXT	link to PCB of sister process
9	SFWLNK	link to next PCB in ready list
10	SRVLNK	link to previous PCB in ready list
11	SSTATE	process state (3.3.4-6)
12	SAWAIT	Awaited event types (3.6)
13	SERROR	error code (3.3.6)
14		error location (3.3.6)
15	SCPU	ref. to CPU control block
16	SRDYQ	ref. to head of ready list (3.4)
17	SPRIO	process priority (3.4)
18	SPROGR	absolute ref. to program (PROG)
19	SMICRO	program page (XAMOS) or ref. to micro program load module (3.4.3)

Figure 3.3.1.a-1 Process Control Block
The use of PCB parameters is explained in the sections indicated in parantheses.

LOCATION	NAME	CONTAINS
20	SBASE	ref. to context save area
21	SABASE	absolute ref. to context save area (BASE)
22	SSECT	process memory section (PSW encoded)
23	SSIZE	size of area belonging to process
24	SEXECT	accumulated
25		execution time
26		in units of TIMER interrupt increments
27	SCREAT	process creation time
28		(same format as used -
29		by procedure READRTC (3.12)
30	RLINK	PCB link for critical region chains
31	SSIGNAL	signal boolean (3.8)
32	SWORK	temporary save location
33	SMSGLIM	max. numb. of msg buffers allocatable by this process (3.7)
34	SMSGUSD	nmb. of msg. buffers allocated (3.7)
35	SMSGQH	message event queue head
36		message event queue head (3.7)
37	SANSQH	answer event queue head
38		answer event queue head (3.7)
39	SSYMQH	system message event queue head
40		system message event queue head (3.7)

Figure 3.3.1.a-2: Process Control Block
The use of PCB parameters is explained in the sections indicated in the parantheses.

LOCATION	NAME	CONTAINS
41	SSYAQH	system answer event queue head
42		system answer event queue head (3.7)
43	SPMQH	path message event queue head
44		path message event queue head (3.7)
45	SPAQH	path answer event queue head
46		path answer event queue head (3.7)
47	SANSWR	ref. to buffer of spefically awaited answer (3.7)
48	SINTRPT	currently awaited interrupt (3.10)
49	SDELAY	current delay (3.9)
50	SCYCLE	cycle value (3.9)
51	SPHASE	current phase (3.9)
52	SPARSIG	parent signal counter (3.8)
53-60	SSAVE Ø - 7	save locations
61	SMSGSLH	list of saved messages (3.7)
62		list of saved messages
63	SANSSLH	list of saved answers (3.7)
64		list of saved answers
65	SSYMSLH	list of saved system messages (3.7)
66		list of saved system messages
67	SSYASLH	list of saved system answers (3.7)
68		list of saved system answers
69	SPTMSLH	list of saved path messages (3.7)
70		list of saved path messages
71	SPTASLH	list of saved path answers (3.7)
72		list of saved path answers
73	SMEMORY	memory allocation parameter

Figure 3.3.1.a-3: Process Control Block
The use of PCB parameters is
explained in the sections
indicated in parantheses.

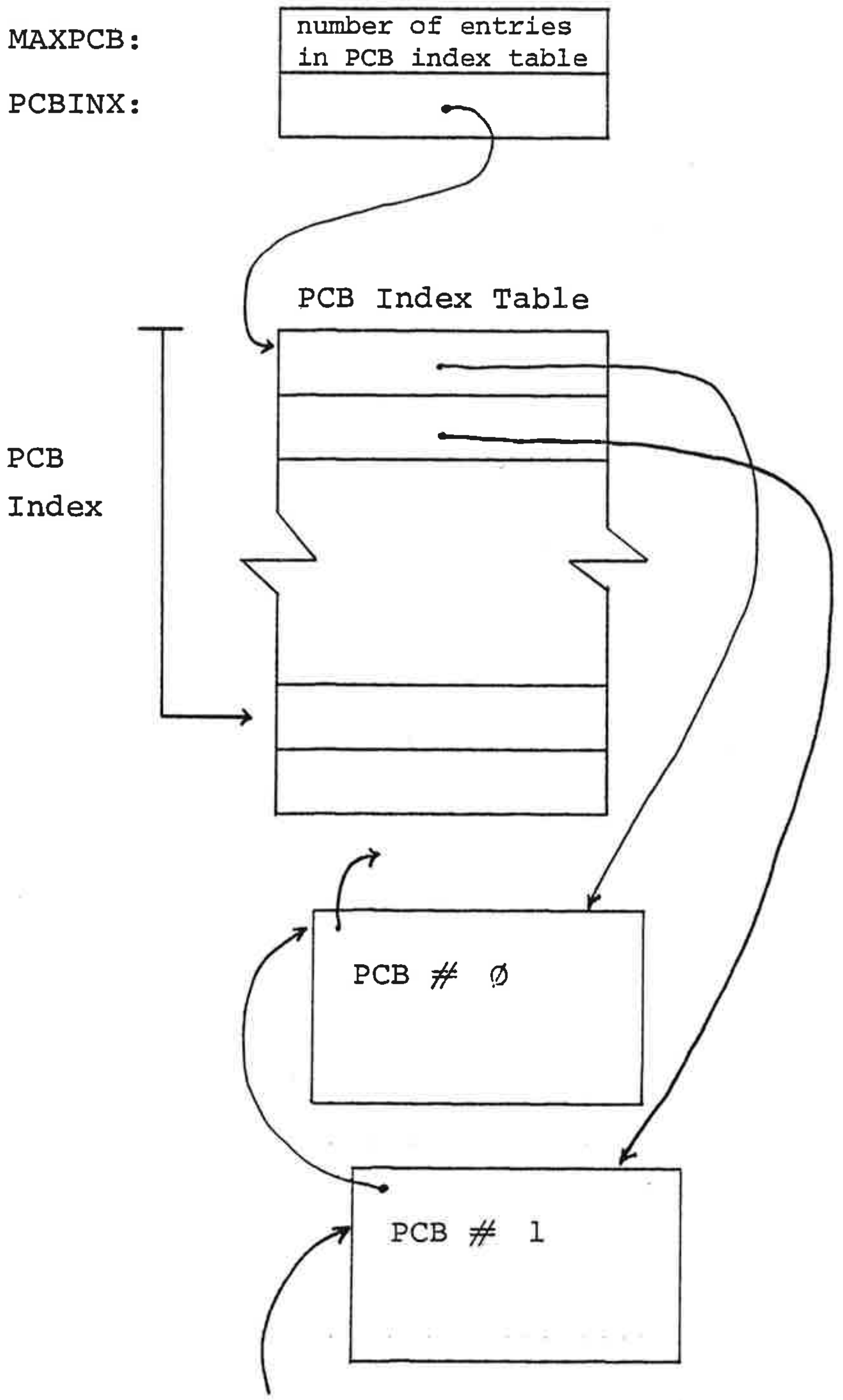
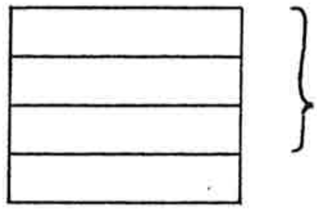


Fig. 3.3.1.b PCB Index Table

Reference to a process is performed by use of a process-name. A process-name contains a 6 letter symbolic part and an index value called name-ident.

process
-name



6 letter symbolic name

name-ident.

When a process is addressed using a process name, the name-ident is in a first attempt used as an index in the PCB index table.

If the name stored in the PCB obtained in this way matches the symbolic part of the process-name, the process is found, else the list of PCBs is scanned until a match is found or until all PCBs have been inspected. If the PCB is found by scanning, the name-ident is updated to contain the proper index.

The same manner of addressing is also used for CPUs (see section 3.4) and critical regions.

The PCB contains references to the contiguous memory area in which the local data of the process associated with the PCB reside.

The lowest addresses of this data area are used by the CPU HW and by the Kernel, as shown in fig. 3.3.1.c.

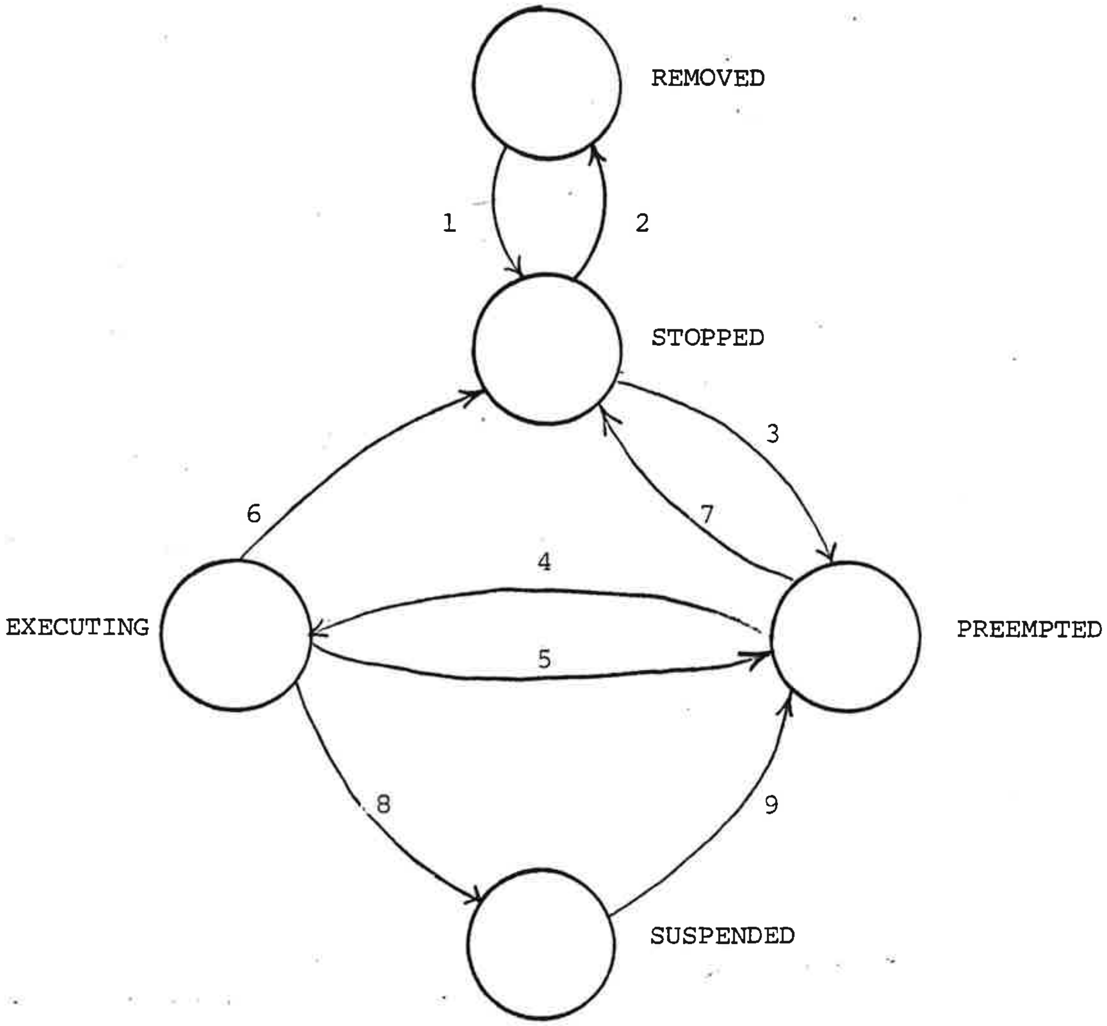
The PCBs are used by the KERNEL process, the IO process and by the RTC process.

LOCATION	NAME	CONTAINS
BASE -6	XUSERIDØ	User-id
-5	XUSERID1	User-id
-4	XCBASE	a copy of the BASE register
-3	XPCB	the PCB index value
-2	XLEVEL	monitor call nesting level
-1	XBOUND	reset value of BOUND register
+0 → +7	XRØ - XR7	save location for register Ø → 7
+8	XBASE	save location for BASE register
+9	XMOD	save location for MODIFY register
+1Ø	XPROG	save location for PROG register
+11	XPRPC	save location for Program Counter
+12	XTIMER	save location for TIMER register
+13	XPSW	save location for PSW (PP SW in XAMOS)
+14	XOLDPRC	BASE of preempted context
+15	XLOCACT	relative address of local interrupt routine
+16	XLOCRET	saved return link at local interrupt
+17	XCAUSE	local interrupt cause code
+18	XDEVICE	device address of interrupting device
+19	XTIMRS	TIMER register reset value
+20	XMONRET	Kernel save location
+21	XTLINK	Kernel save location

Fig. 3.3.1.c BASE relative locations used by Kernel and by CPU firmware

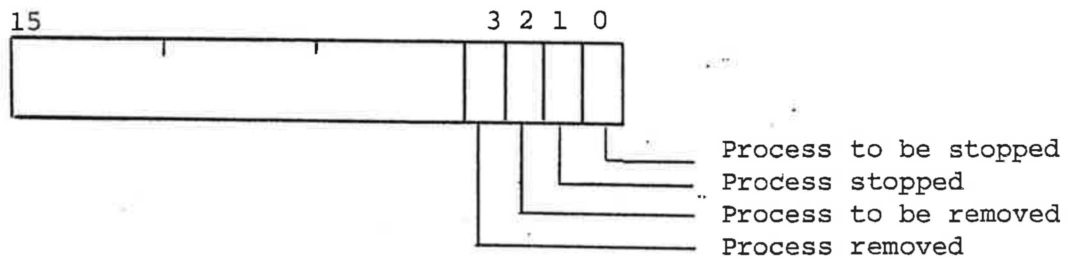
3.3.2 Process States

A process may be in one of five state as shown below:



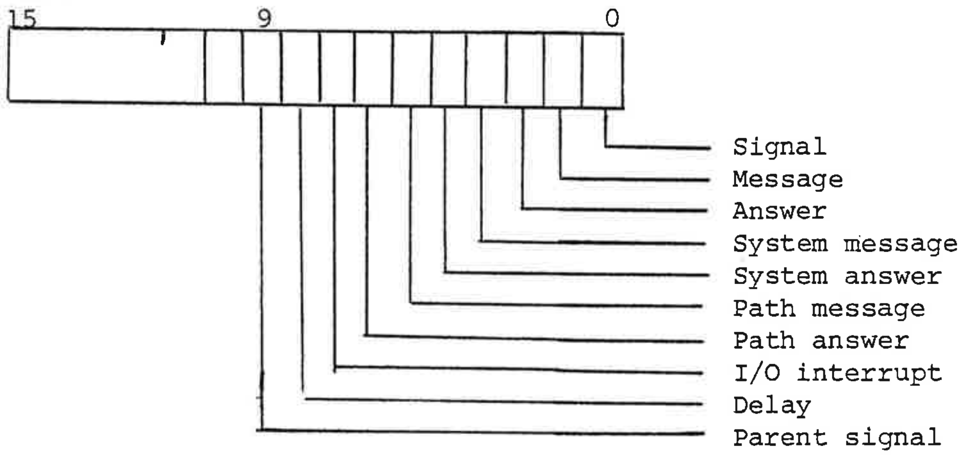
The state of a process is recorded in its PCB in the two parameters SSTATE and SAWAIT.
SSTATE contains a combination of state flags and state transition flags:

SSTATE:



SAWAIT contains a bit mask for awaited events:

SAWAIT:



The states REMOVED and STOPPED are explicitly indicated in SSTATE.

If the process is not in either of these two states, it will be in the SUSPENDED state if SAWAIT is nonzero. If SAWAIT is zero, the process will be EXECUTING or PREEMPTED. Which of these two states it is in, can only be determined by its position in its ready list (see 3.4).

The transitions 1-9 between the states are caused by the following events:

1: The process is subject to creation.

2: The process is subject to removal.

3: The process is subject to a call of START-PROCESS executed by its parent process.

4: The process is loaded by the scheduling algorithm.

5: The process is preempted by the scheduling algorithm or by a call of WAIT EVENT with a zero event mask (3.6).

6: The process is subject to a call of STOP-PROCESS by its parent process. If the parent executes on a different CPU, the transition to STOPPED may be delayed until the process calls a Kernel FUNCTION or until the scheduling algorithm preempts it.

7: The process is subject to a call of STOP-PROCESS by its parent process.

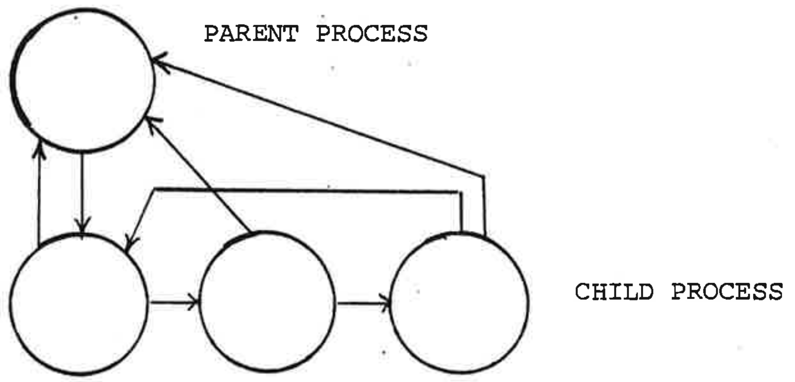
8: The process calls WAIT EVENT with a non zero event mask, and none of the specified event types have an occurred event. An alternate possibility is that the process calls SUSPEND.

9: An awaited event occurs, or the process is subject to a call of READY.

(SUSPEND and READY are only called from the CRITICAL REGION procedures (ref. 2.5)).

3.3.3 Process Hierarchy

Process are organized in a hierarchical manner as shown below:



A process may create subordinate processes. These are called child processes in relation to the former process, which in turn is called their parent process.

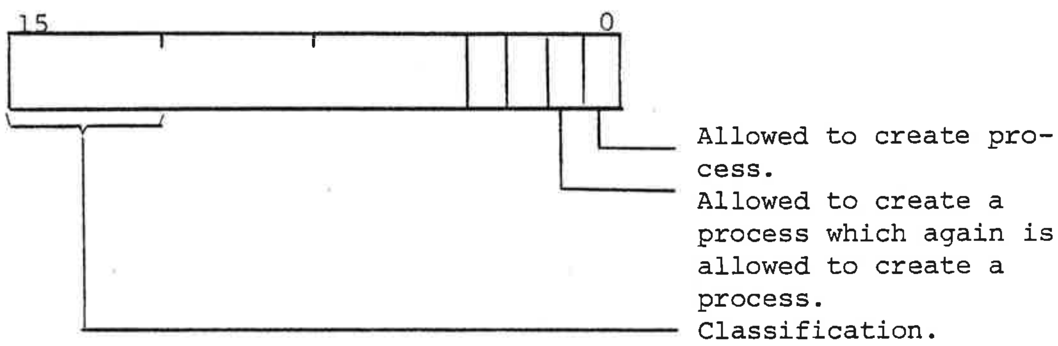
The child processes are kept on a circular list (ref. fig. 3.3.1.a-1, parameter SNEXT).

The parent process has a reference to this list in SCHILD. The children all have a reference to their common parent in SPARENT.

3.3.4 Creation and Removal of Processes

The creation of a process is performed by a call of create process (see 4.9).
The process created becomes a child of the calling process.
The calling process must have the capability to create processes. The process capabilities are defined in its PCB parameter SACCESS.

SACCESS:



The capabilities of a process are defined at the time the process is created. A process cannot create a child with a classification higher than its own.
Neither can a process create a child with the capability to create a child of its own if the former process does not have the capability "allowed to create a process which again is allowed to create a process".
Creation of a process involves allocation and initialization of a PCB.
The initialization is performed according to parameters specified in a parameter block (ref. 4.9).

Child processes can only be removed by their parent process. When a parent process removes a child by calling Remove process (refer to 4.12) the child process is forced to execute a "clean up program" which performs the following tasks:

- The child removes all its own children one by one.
- The child calls CLNIO (refer to 2.4) for cancelling all I/O activities it might have invoked.
- The child calls CLNMEM for release of all memory it might have allocated.
- The child calls the kernel function CLNMESSAGE for cleaning up message communications it might be involved in:

    - Messages received but not yet answered are redirected to ROOT for answering them.

    - Messages sent for which an answer has not yet been received are modified to look as if they were orginated by ROOT.

- The child calls the Kernel function CLNINTRT which releases all interrupts reserved by the child.

3.3.5 Starting and Stopping of Processes

A parent process has the capability to start and stop its child processes by calling start process or stop process.

These functions may be used to build a long term scheduling facility in which the parent is the scheduler.

Stop process will not in general cause an immediate stop of the child process. The child process which may execute on another CPU will however be stopped the first time it enters the kernel. This will eventually happen when its time slice elapses (refer to 3.5).

3.3.6 Other Process Management Functions

For management of processes five other functions are implemented.

Get child enables a parent process to inspect its child processes one by one.

Get attributes delivers an extract of the PCB parameters for a given process.

Lookup process returns the PCB index (name-ident) of a process if its symbolic name is known.

Identify process returns the symbolic name of a process if its PCB index is known.

Adopt process allows a parent process to hand over a child to the grandparent of the child.

3.4 CPU's

CPU's are handled by the Kernel as separately identifiable objects.

Each CPU has its own ready list(s) of processes and is scheduled separately. When a process is created, it is determined which CPU is shall execute on.

Dynamic creation of CPU's is not supported. It is a system generation task to define the number of CPU's in a system (section 7).

CPU's are identified by CPU-names which are constructed like process names (see section 3.3).

3.4.1 CPU Control Blocks

For each CPU in a system there exists a CPU Control Block (CB). The CPUCB consists of one part which occurs once and another part which occurs as many times as there are software priorities (refer to 3.5).

The CPUCB is shown in fig. 3.4.1.a. and b.

The kernel holds a CPUCB index table which contains pointers to the existing CPUCB's. The CPUCB index table is indexed by a CPUCB index and constructed similarly to the PCB index table (fig. 3.3.1.b).

Most of the CPUCB parameters are used by the scheduling algorithm.

LOCATION	NAME	CONTAINS
0	SCHAIN	link to next CPU CB
1	SNAME	symbolic
2		name of
3		the CPU
4	not used	
5	CCPUID	physical CPU number
6	CLOGCPU	CPUCB index for this CB
7	CCPUMS	address of CPU message location(ref. 2.1)
8	CCPUIP	BASE of CPU service process
9	CIMASK	CPU interrupt mask (PSW)
10	CMICRO	ref. to currently loaded micro program module (initially zero)
11	CIDLEP	ref.to PCB of CPU idle process
12	CRUNPR	ref. to PCB of currently executing process.

Fig. 3.4.1a CPU Control Block
This part occurs once.

LOCATION	NAME	CONTAINS
X + Ø	CCURPR	ref. to first PCB in ready list
X + 1	CSCHCN	schedule count (3.6)
X + 2	CSCHRS	schedule reset count
X + 3	CSLISZ	slice size (TIME register increments)
X + 4	CACTIM	accumulated exec. time
X + 5	CHWPRI	HW priority (Ø,1,2, or 3)

Fig.: 3.4.1.b CPU Control Block
This part occurs CPRIOS times.
(assembly time parameter)

3.4.2 CPU Procedures

Some of the CPUCB parameters may be inspected and modified by using the functions Get CPU parameter and Set CPU parameter respectively.

The parameters which are accessible by these functions are CCPUID, CIMASK, CSCHRS, CSLISZ, CACTIM, and CHWPRI.

CPU's are identified by CPU names which are constructed like process names (refer to 3.3.1).
However, Get and Set CPU parameter use the CPUCB index to identify the CPU. It is also the CPUCB index which is used in connection with create process.

The function look-up CPU may be used to deliver the CPUCB index for a CPU.

3.4.3 Scheduling

The scheduling algorith implements a prioritized multiplexing of a CPU among the preempted processes waiting to execute on it.

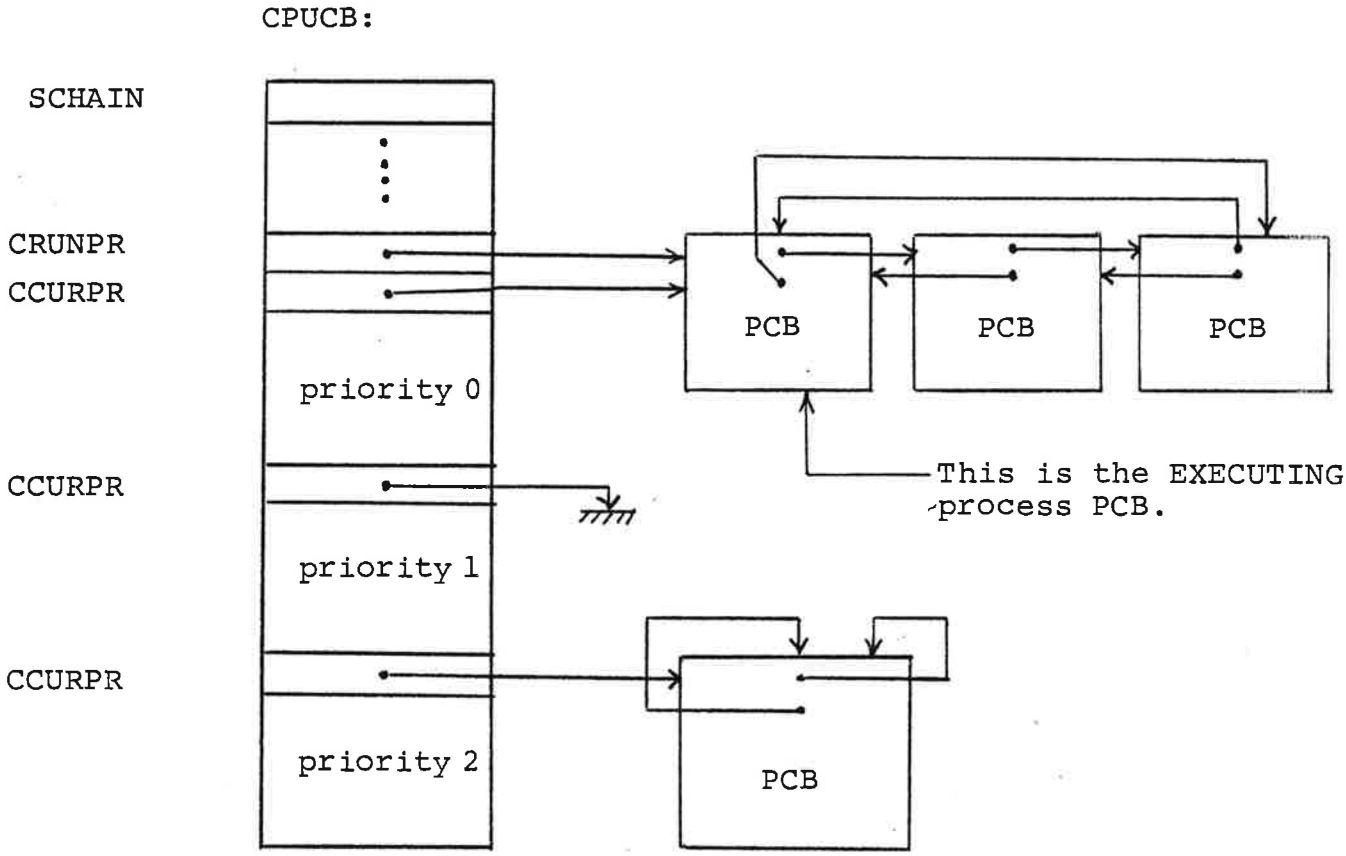
The scheduling algorith works independently for each CPU. The scheduling algorith is invoked

- When a process calls wait event, await answer, await system answer, or await path answer to receive a not yet occurred event type.

- When a process encounters a timer action (a decrement of the TIMER register resulting in a negative value) or when it calls wait event with a zero event mask.

In the former case the process is suspended until an awaited event occurs, in the latter it is preempted and its timer register is incremented by the time slice size defined for the software priority level (CPUCB item CSLISZ). It will enter the executing state again controlled by the scheduling algorithm.

For a given CPU, the executing process and the preempted processes are kept in circularly organized ready lists. There is a ready list for each software priority (assembly parameter CPRIOS) (refer to fig. 3.4.3.a).



In this example there are 2 preempted processes and 1 executing process at priority level Ø and 1 preempted process at priority level 2.

Fig. 3.4.3.a CPU Ready Lists.

The algorithm for selecting a process for execution is shown in the flowchart fig. 3.4.3.b.

It may be noted that there has to be at least one process which is ready to execute. To ensure this there is initially created an Idle process for each CPU (refer to 3.15).

When a process has been selected for execution, it is checked whether the process requires a micro program module to be loaded into the CPU loadable control store. If this is the case (PCB item SMICRO is greater than 3) and if the module is not already loaded (SMICRO different from CPUCB item CMICRO), a procedure is called which loads it.

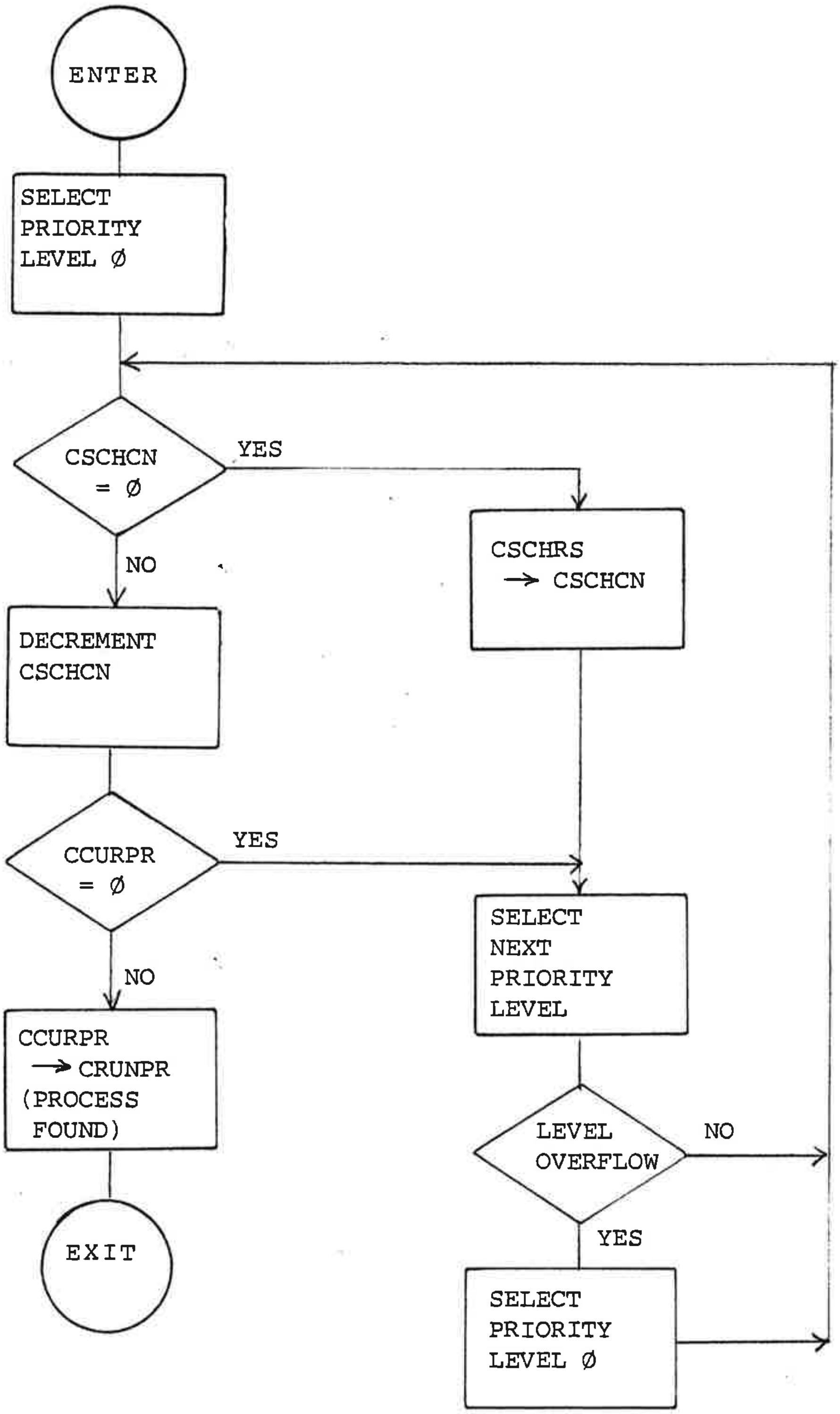


Fig. 3.4.3.b SCHEDULING ALGORITHM
(SELECTION OF THE NEXT PROCESS TO EXECUTE)

3.5 Critical Regions

Critical regions are used for sharing variables between different processes, and for synchronization.

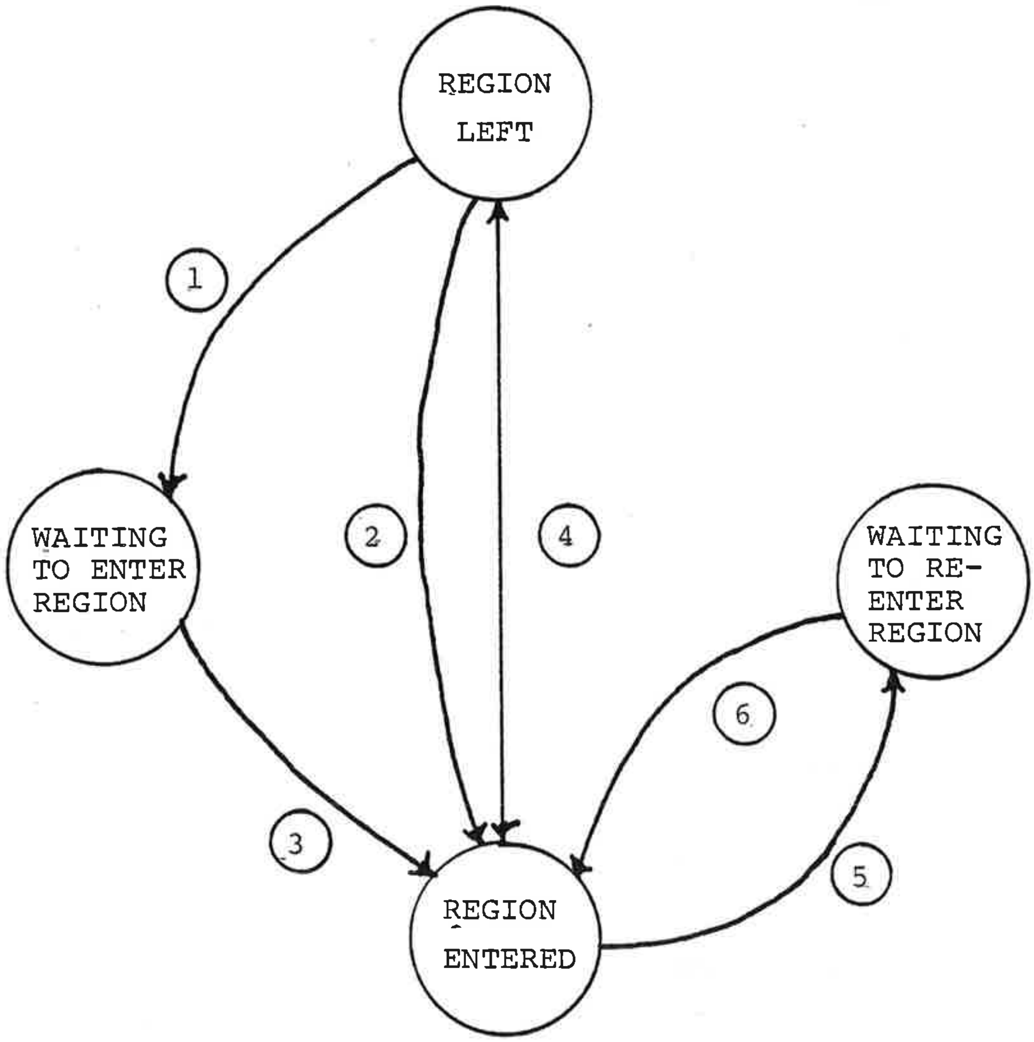
The critical region primitives are designed to solve two problems with shared variables:

- that of addressing, and
- that of contention.

A critical region consists of a control block (CRCB) which is allocated from a system pool of CRCB's and an associated contiguous memory area which holds the common variables. This memory area is called the Variable Space (VS). The allocation of VS is not part of the critical region primitives.

Addressing of variables in the VS is relative to the origin of the VS. A user process should not know the absolute address of the VS. Addressing of critical regions is symbolic. A critical region is addressed by name. The name of a critical region is constructed in the same manner as process names (ref. to section 3.3.1).

In connection with a specific region a process will be in one of the following states:


REGION LEFT
WAITING TO ENTER REGION
WAITING TO RE-ENTER REGION
REGION ENTERED
1
2
3
4
5
6

Note that these states only apply to the relation between a single region and a process. The process may interact with several other regions at the same time.

The meaning of the states are:

Region left:

In this state the process has no access to the VS of the region. A process will initially be in this state.

Region entered:

In this state the process has access to all the variables of the VS. Only a single process can be in this state (in relation to a specific region) at any one time.

Waiting to enter region

The process is suspended until no other process is in the 'region entered' state.

Waiting to re-enter region

The process is suspended until a process leaves the region.

The purpose of this state is to be able to wait until the variables of the VS fullfills a wanted condition.

The transitions between the states occur at the following events:

1: The current process calls ENTER-REGION and the region already contains a process in the 'region entered' state.

2: The current process calls ENTER-REGION and no process is in the 'region - entered' state.

3: Another process (which was in the 'region entered' state) calls LEAVE-REGION or WAIT-REGION, and the current process is at the head of the queue of processes waiting to enter the region and no processes were in the state 'waiting to re-enter region'.

4: The process calls LEAVE-REGION.

5: The current process calls WAIT-REGION.

6: Another process calls LEAVE-REGION or WAIT-REGION, after having modified the contents of the region variable space and the current process is at the head of the queue of processes waiting to re-enter the region.

The normal use of critical regions is

- to enter a region
- modify and/or inspect the variables in VS
- if the variables inspected must fullfill a certain condition (which they do not) before processing can continue, the process may call WAIT-REGION. This causes the process to be delayed until at least one other process has been in the 'region entered' state, and has modified the contents of the region variable space.
- and finally to leave the region.

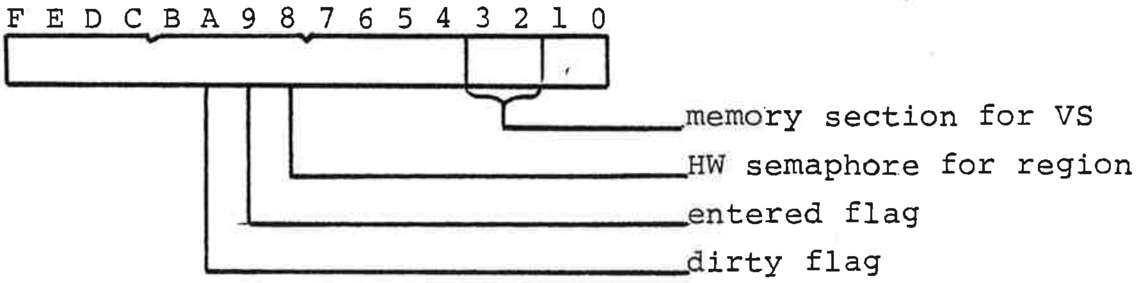
A region need not control a VS. If it does not, the critical region serves as a simple synchronization element.

3.5.1 Region Control Block

For each critical region a region control block (RCB) must exist. RCB's are allocated from a pool of free RCBs which is set up at system initialization time. The kernel has a RCB index table which contains pointers to the RCBs.

The structure of a RCB is shown in figure 3.5.1.a.

The word CRSTA in the RCB needs a further explanation; CRSTA contains the following fields:



The HW semaphore is used to synchronize about the use of the region control block itself. The entered flag defines whether a process is in the entered state or not. The dirty flag is set when a write operation is performed on the variable space and cleared when the wait queue is transferred to the entered queue.

LOCATION	NAME	CONTAINS
0	SCHAIN	link to next RCB
1	SNAME	symbolic name of region
2		
3		
4	SLOGRCB	RCB index
5	CRADDR	absolute word address of variable space
6	CRSTA	status word. Refer to the text
7	CRSIZE	size of variable space in words
8	CREQP	pointer to PCB of first process waiting to enter region
9	CREQL	pointer to PCB of last process waiting to enter region
10	CRWQF	pointer to PCB of first process waiting to reenter region
11	CRWQL	pointer to PCB of last process waiting to reenter region
12	CRCPCB	PCB index of entered process (-1 if none entered)

Fig. 3.5.1.a Region Control Block.

3.5.2 Critical Region Procedures.

Procedures are provided for creating critical regions, for entering, leaving, waiting to re-enter regions, to get items from the variable space and to put items into the variable space.

For a detailed description of the procedures, refer to section 4.

3.6 Events

An event is defined as an incident which transfers synchronization and/or data information from a process or a peripheral device to another process. The following event types are defined and supported by the Kernel:

(a) . Messages
(b) . Answers
(c) . System messages
(d) . System answers
(e) . Path messages
(f) . Path answers
(g) . Signals
(h) . Parent signals
(i) . I/O interrupts
(j) . Delays

Event types (a) through (f) are described in section 3.7, (g) and (h) in section 3.8, (j) in section 3.9 and (i) in section (3.10).

3.6.1 Receiving Events

The primary Kernel function to call for receiving an event is wait event (section 4.2). Wait event allows a process to wait for and receive the first occurring event of a number of event types.
If no events of the types specified in calling wait event have yet occurred, the process is suspended until one occurs.
If an event has been sent but not yet received, the process will receive it and continue processing.

Receiving an event may imply receiving data (as in the case of messages and answers). Other event types are not associated with data.

If wait event is called to receive e.g. a system answers, the first occurred system answer will be received. Sometimes it is preferable to wait for a specific system answer. This is possible by calling await system answer. Similar functions exist for answers and path answers.

It is sometimes advantageous for receive and process events in an order different from the first sent - first delivered order implemented by the kernel.

For this purpose three functions are available:

inspect events
save event
restore events.

Inspect events is similar to wait event with the only difference that events are not removed from the kernel when received by the receiver, i.e. they may be received again.

Save event is used to temporarily save an event which may be an answer or a message which has been received by a call of inspect events or a message which has been received by wait event.

The event is removed from the corresponding event queue and inserted in a save queue for the event type in question.

Restore events, which is called with an event type as parameter, transfers the saved events of the defined type to the corresponding event queue. The events may then be received again by wait or inspect events.

3.6.2 Sending Events

There are a number of functions for sending events - one for each event type (except I/O interrupts where hardware/firmware is used to do this).

When an event is sent, it is checked if the receiver process is waiting for this event (possibly among other events and/or event types). If this is the case, the state of the receiver process is changed from SUSPENDED to PREEMPTED - unless the receiver process is in the STOPPED state - and the receiver process is linked to its ready list at the second position in the list. If the list was empty, it is placed at the head of the list.

If the receiver process is not awaiting the event, the event is queued. The method of queuing is different for each event type and is described in the appropriate of sections 3.7 through 3.10.

K

3.7 Message Type Events

This section describes messages, answers, system messages, system answers, path messages, and path answers.

A message is 5 words of user defined information. The transmission of a message is always performed in two steps:

- the message data is copied from the sender process to a system supplied message buffer,

- the message data is copied from the message buffer to the receiver process.

The first step is accomplished when the sender calls the appropriate send function (refer to sections 4.18, 4.19, 4.21, 4.22, 4.26, and 4.27). The second step is performed when the receiver process is ready to receive the message (or answer). This happens after a call of the appropriate wait function (refer to sections 4.2, 4.20, 4.23, and 4.28).

The message buffer is used to identify the event (when sending an answer it is necessary to specify the message to which it is a reply). The message buffer is allocated from a pool of message buffers, which is defined at system generation time (refer to section 7) and initialized at system initialization time (refer to section 3.12). The allocation of a message buffer is performed when

- a message is sent
- a system message is sent, or
- a path is opened (refer to 4.24)

The message buffer is deallocated when

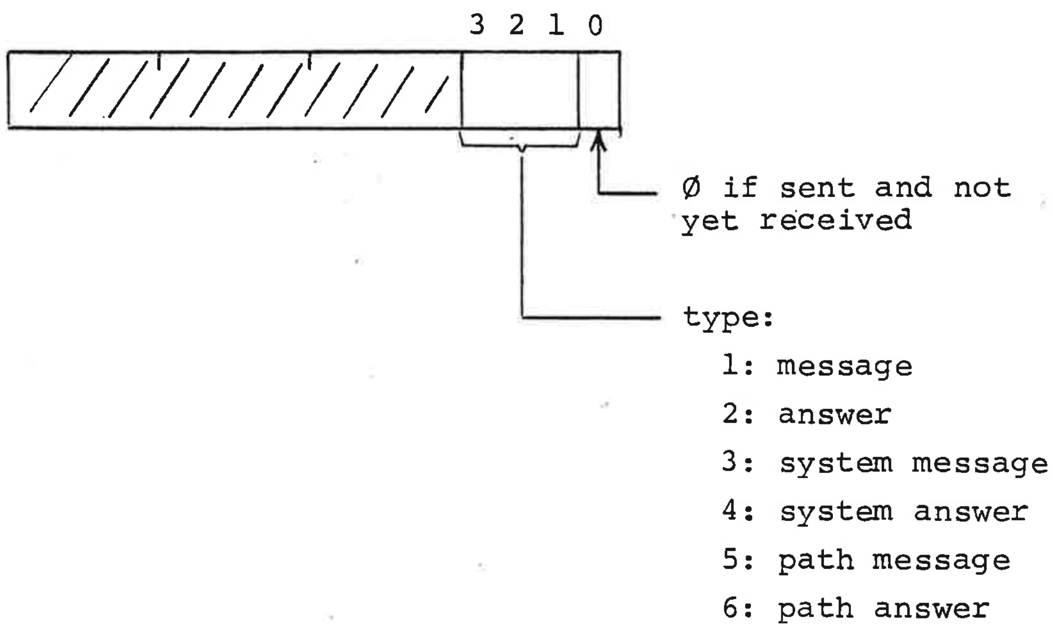
- the answer is received
- the system answer is received
- the path is closed

The number of message buffers which a process has in use (allocated) at any one time cannot exceed the value of the process creation parameter VMSGS (refer to 4.9).

The format of a message buffer is

0	LINK TO NEXT MESSAGE
1	MESSAGE BUFFER INDEX(EVENT)
2	PCB INDEX OF PROCESS SENDING MESSAGE
3	PCB INDEX OF PROCESS RECEIVING MESSAGE
4	MESSAGE STATE
5	CONTENTS OF MESSAGE/ANSWER
6	
7	
8	
9	

The message state parameter has the following layout:



3.7.1 Path Messages/Answers

Path messages and answers are different from ordinary and system messages/answers in the following respects:

- The message buffer is allocated by a special call (open path) which also identifies the receiver process.

- The message buffer stays allocated until a special function (close path) is called.

- When a path message is sent the message buffer must be identified (EVENT), but the receiver is not explicitly identified.

A path can only be closed by the process which opened it.

3.8 Signal Type Events

The signal type events supported by AMOS are signals and parent signals.

Parent signals are reserved for use by the Kernel. There is no separate function for sending of parent signals. Parent signals are automatically sent when a process calls Error (or Terminate) (refer to 4.3) or when it encounters a local interrupt which is not a timer action (refer to 4.1). Sending a parent signal consists of incrementing the parent signal counter (PCB item SPARSIG) of the parent process. If the parent process awaits a parent signal, the parent signal is received by it. Receiving a parent signal implies decrementing the parent signal counter.

Signals can be sent to any process. The function for sending signals is described in 4.17. Sending a signal means setting the signal boolean (PCB item SSIGNAL) to true (=1). Receiving a signal involves setting the signal boolean to false (=0).

As no resources are involved in sending signals, signals may be used unrestrictedly.

(The standard AMOS Teletype writer driver uses signals for calling the attention of processes identified by the teletype operator).

3.9 Delays

Delays are primarily used for two purposes:

- to generate a long term scheduling,
- to timeout waiting for events which do not occur.

Delays are defined in units of 0.100 secs. Delays are implemented by the Real Time Clock (RTC) driver (refer to 3.14) which receives an I/O interrupt from a hardware clock every 10 milliseconds.

The RTC maintains a phase for every process in the system (PCB item SPHASE). The phase is originally set to zero.
Every 100th millisecond the RTC scans through the chain of PCB's and every non zero phase found is decremented. Every phase which is zero is reset to the cycle value (PCB item SCYCLE). The cycle is also initially zero, but may be changed by a call of Set cycle (refer to 4.33).
When a process calls wait event, it may specify a delay. When await answer, await system answer, or await path answer is called, a delay must be specified. The process wil regain control (enter the EXECUTING/PREEMPTED state) at the latest when a timespan equal to the total of the specified delay and the phase value at the time of call has elapsed.

A cyclic behaviour of a process can be implemented by setting the cycle to the required period and include in the program the following sequence of code:

```
MAINLOOP:
    MOVC  BMDELAY        R2;
    MOVC  Ø              RØ;
    MON WAITEVENT          ;
        .                  ;
        .                  ;
        .                  ;
    JMP  MAINLOOP          ;
```

3.10 I/O Interrupt

The Kernel provides the following functions for handling of I/O interrupts:

- Reserve interrupt
- Release interrupt
- Clear interrupt
- Set interrupt

In order to avoid confusion the term interruption is used for the event that an I/O device transmits its I/O address and device priority code to the CR80 Main Bus Controller and thereby causes an interruption of a CPU.
The term interrupt is taken to mean all interruptions generated by a specific device.

Interrupts are resources which must be reserved by the process before an interruption can be awaited and received.
Reserve interrupt establishes a connection between an I/O device and a process. This connection lasts until the process is removed or it calls release interrupt with the corresponding interrupt as argument.

Any interruption generated by a peripheral device are received by the Kernel. The Kernel maintains an interrupt occurrence table with 64 entries, one for each possible interrupting device.

When an interruption is received by the Kernel, it is checked if it was awaited by a process. If this is the case, the interruption is delivered to the process. Otherwise, the interrupt occurrence table entry is incremented.

When a process calls wait event specifying interrupts as an eventtype, it is checked if the proper occurrence table entry has a non zero value. If so it is decremented, and the process continues immediately.

A process may reserve more than one interrupt. It may however only await interruptions from a single device. If a process has reserved more than one interrupt, it must define the currently awaited by a call of set interrupt. This is not necessary if only one is reserved.

Clear interrupt sets the occurrence table entry to zero.

Release interrupt breaks the connection between a process and an I/O device. The process will not be able to await and receive interruptions from the device after a call of release interrupt with the corresponding interrupt as argument.

3.10.1 Processing of I/O interrupt in the Kernel

In a CR80 multiprocessor, one and only one CPU may execute with the I/O interrupts enabled. That means that it is always the same CPU which is interrupted.

When the interruption occurs, the CPU performs a context switch to the I/O context.

The I/O process thus loaded immediately enters the Kernel by reserving the Kernel Hardware semaphore. It checks to see if any process awaits the current interruption. If not, the proper occurrence table entry is incremented, the I/O process leaves the Kernel by releasing the Kernel hardware semaphore and performs a programmed context switch back to the preempted context.

If the interruption was awaited, there are two cases to consider:

1. The waiting process must execute on the same CPU as does the I/O process.

   In this case the I/O process switches to the Kernel context which puts the interrupted process in the PREEMPTED state and sets the awaiting process in the EXECUTING state, leaves the Kernel and performs a context switch to that of the waiting process.

2. The waiting process must execute on another CPU.

In this case the I/O process prepares itself to execute in a second incarnation on the other CPU. While still being in the Kernel, it sends a CPU interrupt to the other CPU and then releases the Kernel semaphore; this causes the other CPU to load the second incarnation of the I/O process. This twin reserves the Kernel semaphore, and subsequently sets a hand-shake signal to cause the original I/O process to perform a context switch back to the interrupted process.

The situation in the second CPU is now similar to 1. above.

3.11 Handling of Errors

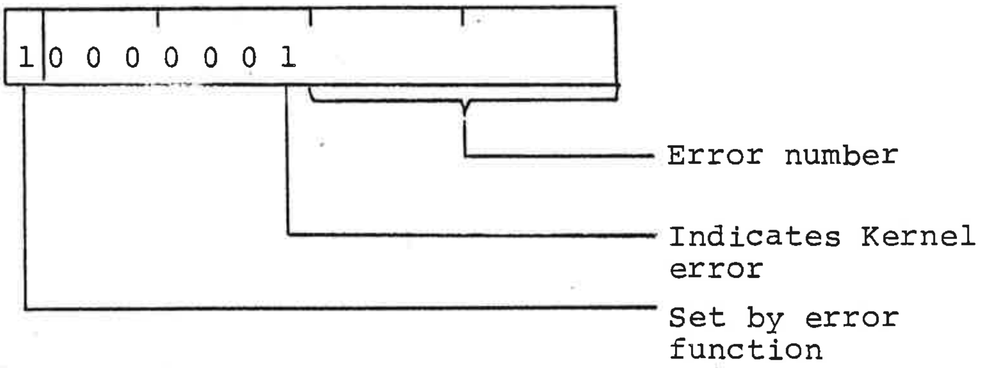
The Kernel performs a validation of all parameters used when calling a Kernel function or procedure.

A invalid parameter may either cause a return to an error exit or it may cause an invokation of Error (refer to 4.3).
An example of the former case is create process (refer to 4.9), and all critical region procedures.
In the latter case an error code is used with the upper byte equal to 1 and an error number in the lower byte. The error numbers used are listed in section 3.11.1.
The return link generated at call of the function or procedure is used as error location.

The inability to perform a function will either cause an automatic re-call of the function (as in the case of send message) or a return to an error exit.

3.11.1 Kernel Error Codes

The error codes used by the Kernel have the following format.



The error numbers applicable are:

1: Trap or illegal instruction executed
2: Parity error encountered
3: Time out (illegal addressing)
4: Bound violation (XAMOS only)
5: Reference is made to a not existing process
6: Parameter reference exceeds the local process memory area
7: Invalid event parameter
8: Calling process is not sender or receiver of this message buffer.
9: Invalid message buffer state for this function
10: Invalid Intrpt parameter
11: Invalid Intrpt parameter
12: Invalid Item type
13: Attempt to use too many message buffers
14: Not implemented monitor function
15: Monitor level too large (XAMOS only)

30: Process not allowed to call create process.

3.12 Initialization

After boot loading of a system a separate initialization program, INIT (CSS/308), prepares the system initialization to be performed by the kernel.

INIT checks if the kernel is going to have its local data structures resident in memory section 0 or 1.

If section 0 is used, INIT performs the following tasks:

- The space required for kernel pools (message buffer pool, pcb pool, rcb pool) is calculated from the kernel init list prepared by SYSGEN (CSS/121).

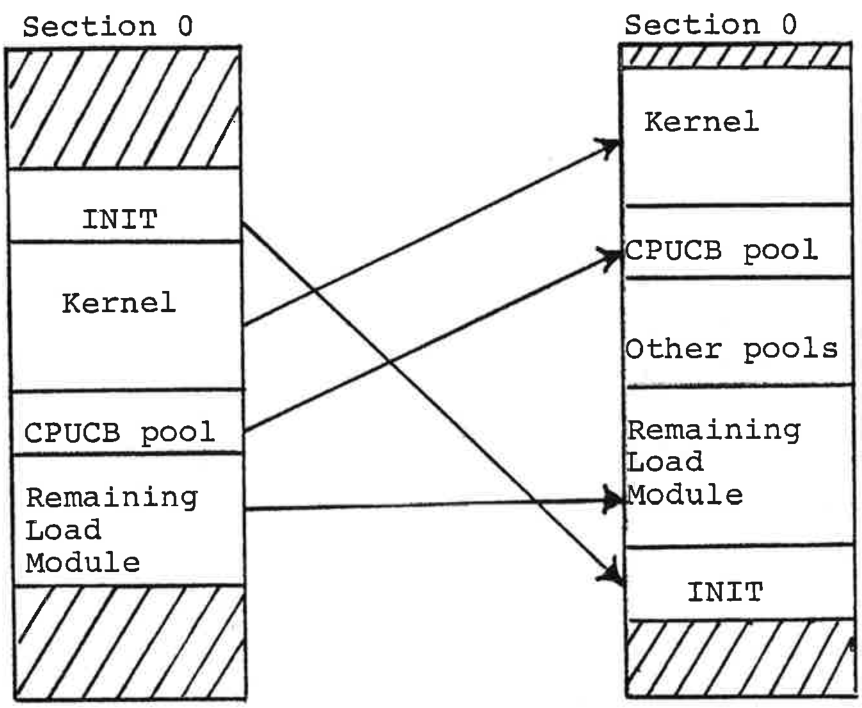
- The load module above the kernel is displaced to make room for the pools.

- The top of the load module is determined.

- INIT moves itself above the top of the load module.

- The kernel module is moved to location 15.

- The kernel process (base 19) is loaded.



Before initialization After

If section 1 is used, INIT performs the following tasks:

- The kernel data is moved to section 1 location 0.

- The space required for pools is determined and the pools laid out.

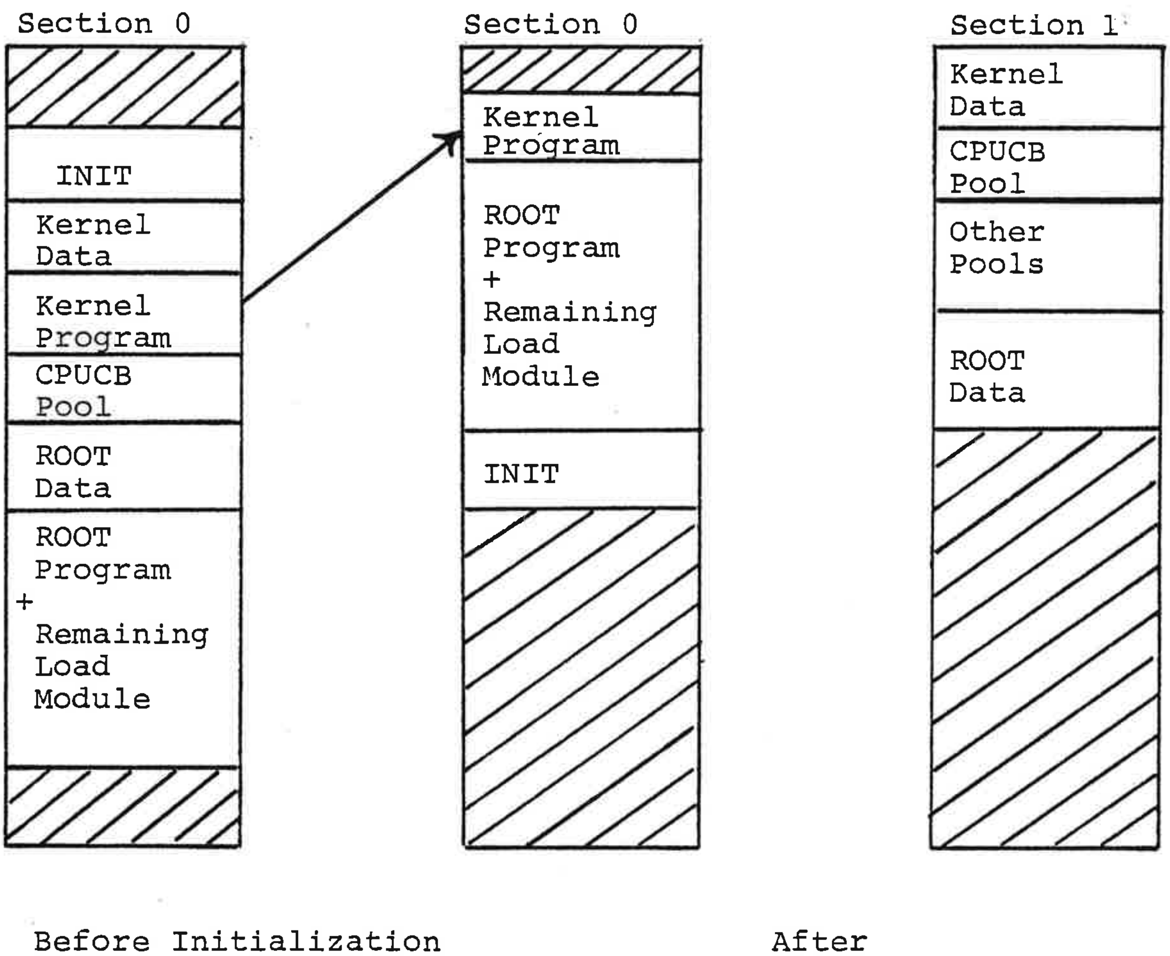
- The ROOT data part is moved to section 1 following the pools.

- INIT moves itself to the top of load module.

- The kernel program is moved to location 320 (=256+64) leaving space for the monitor jump table and interrupt table.

- The load module is compressed.

- The kernel process (base 4) is loaded.



During system initialization, the Kernel uses an initialization list. This list has the following format:

Init list + Ø :	Kernel context relative pointer to message buffer pool.
Init list + 1 :	Kernel context relative pointer to PCB pool
Init list + 2 :	Kernel context relative pointer to CPUCB pool
Init list + 3 :	Kernel context relative pointer to RCB pool
Init list + 4 :	Kernel context relative pointer to first location of ROOT process data part

The processing performed by the Kernel is shown in the flowchart fig. 3.12.a. Ther initialization is performed in the Kernel context.

The last step in the initialization is to switch to the ROOT process.

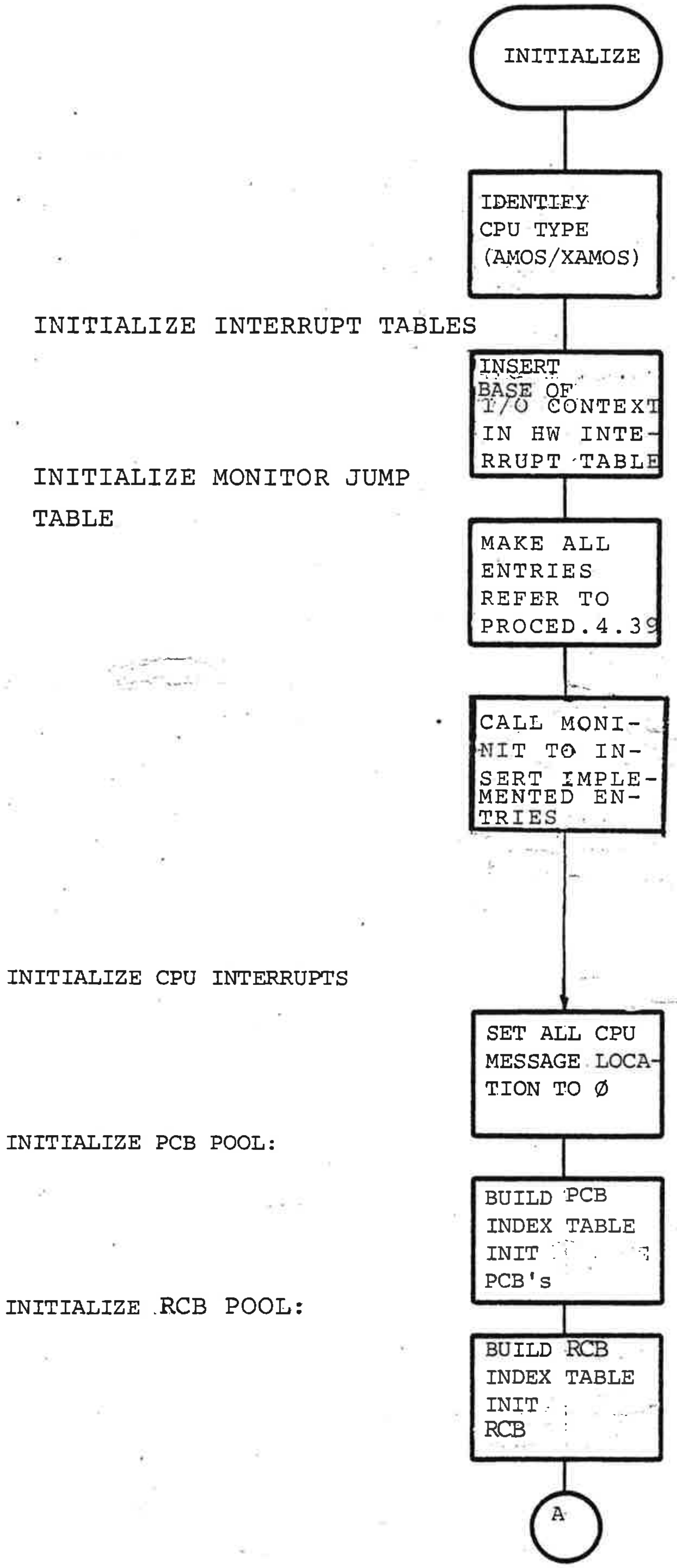


Fig. 3.12.a-1 SYSTEM INITIALIZATION FLOWCHART, PART 1/2.

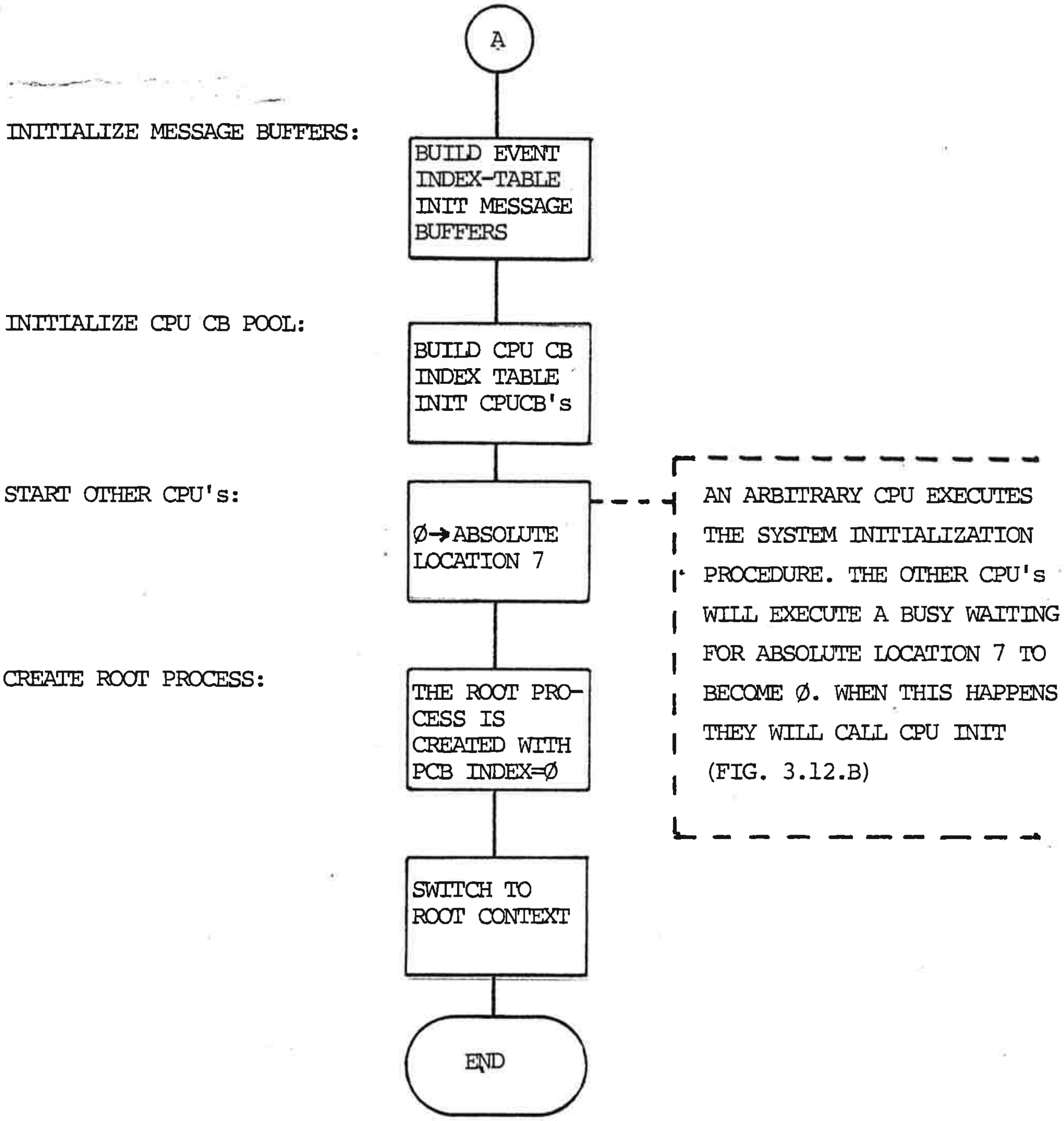


Fig. 3.12.a-2 SYSTEM INITIALIZATION FLOWCHART, PART 2/2.

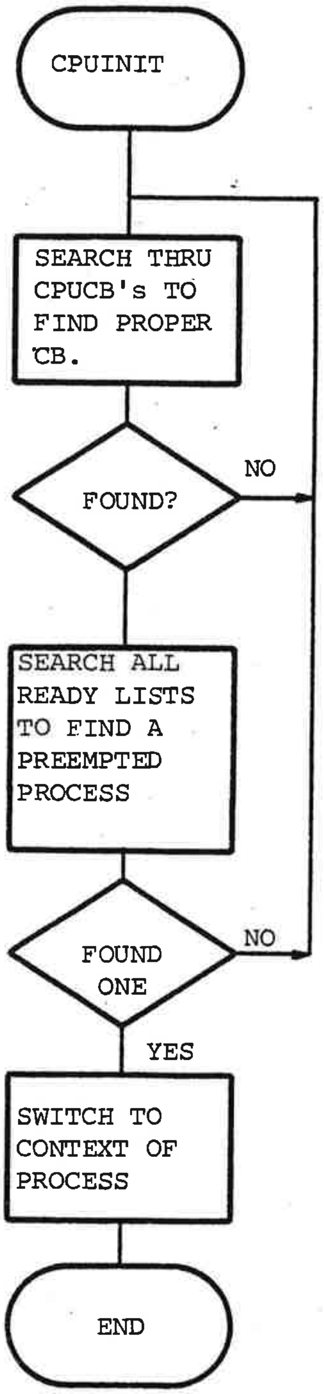


Fig. 3.12.b INITIALIZATION OF CPU's.

3.13 Root Process

The Root process is part of CPCl CSS/360.
The Root process fulfils three purposes:

- it takes over initialization after the Kernel initialization
- it receives events which are sent to not existing processes
- it receives parent signals from its own child processes.

3.13.1 Root Initialization Processing

When loaded by the Kernel, the Root process starts initializing of assembled/compiled modules. The Root requires that the modules are laid out contiguously in main memory and that they follow immediately after the Root program part.

Root expects modules to be programs, data modules, or tables. When anything different from this is encountered Root terminates initialization.

The format of modules is defined in appendix A.

When a program of type Monitor is encountered, Root performs a subroutine branch to the program entry (refer to Appendix B, file X2GEN1, item XSTART). The return link is generated in register 4.

A monitor program module should therefore contain the following construct:

```
   myinit:
                 ┌MON MONITIT
                 │LOC
(refer to 4.38) <  <myargumentl>, <mylabell>
                 │ :
                 │<myargumentn>, <mylabeln>
                 └Ø
                  JMP        Ø.   X4; Return to Root
                   :
   XSTART = myinit          ; define program entry.
```

When a data part is encountered, a process is created. The process is prepared to execute the last preceeding program. Data parts need not be assembled/compiled to full size. If a process requires more data space than it is compiled with, Root will move all succeeding modules accordingly.

Table modules encountered by Root are skipped; no processing is performed.

Root prints on the operators console, a log of the programs and processes as they are encountered. An example of such a log is shown in figure 3.13.1.a.

Programs and processes are placed in memory as required in the XPGMEM and XPRMEM parameters. Monitor programs are allways placed in memory section 0. Other programs are preferably placed in memory section 0. Processes are preferably placed outside memory section 0.

```
Program  -->   CSS360 VERSION:  1001 PROG:#0F16/0
Process  -->   ROOT   BASE:#0CD4/1
CPU type -->   XAMOS
               RTC    BASE:#15B4/1
               MEMMGR BASE:#0D44/1
               CSS306 VERSION:     1 PROG:#1464/0
               P00003 BASE:#16A4/1
               FMS    VERSION:   403 PROG:#1489/0
               FILSYS BASE:#001C/3
               CSS321 VERSION:     4 PROG:#30D5/0
               FDD000 BASE:#3C1C/3
               CORU   VERSION:   514 PROG:#346A/0
               CSS311 VERSION:   802 PROG:#35C1/0
               TTY000 BASE:#3E1C/3
               CSS361 VERSION:   102 PROG:#3A72/0
               CSS316 VERSION:   102 PROG:#3AE1/0
               CSS317 VERSION:   102 PROG:#3BD2/0
               CSS355 VERSION:   803 PROG:#3CFA/0
               PASRTS VERSION:   203 PROG:#4A07/0
               CSS380 VERSION:   801 PROG:#5A92/0
               S      BASE:#401C/3
```

Fig. 3.13.1.a Example of log generated by Root.

During initialization the following error message may be output from Root on the operator's console:

INIT ERROR #HHHH #KKKK

HHHH and KKKK are hexadecimal error numbers.

HHHH is an error code with the following possible values and corresponding errors:

0 failed to create memory manager process

1 failed to allocate memory for ROOT itself.

2 failed to allocate memory for the next program.

3 failed to allocate memory for the next process.

4 a module is encountered with illegal type (neither program nor process).

5 failed to create next process.

6 failed to start next process.

7 failed to start memory manager.

3.13.2 Event Processing

Following initialization Root enters a loop when all event types but interrupts, signals, and delays are awaited. The handling of events received depends on the eventtype as follows:

Messages	: an answer is returned with the first word set to 1<BNUNKNOWN.
System Messages	: a system answer is returned with the first word set to 1<BNUNKNOWN
Path Messages	: a path answer is returned with the first word set to 1<BNUNKNOWN
Answers	: no action
System Asnwers	: no action
Path Answers	: the path is closed.
Parent Signals	: the child processes are inspected. When a child with a nonzero SERROR is found, a log line is generated and printed on the operator's console. The form of the message is

PROCESS <name> TERMINATED WITH CAUSE,LOC: #HHHH, # HHHH

3.14 Real Time Clock Process

The RTC is part of CPCI CSS/360.
The Real Time Clock (RTC) process receives the interrupts generated every 10th msec by the hardware clock.
Everytime 10 interrupts have been received the RTC updates a local timer consisting of the following 7 words:

RTCYR: current year
RTCMTH: current month
RTCDAY: current day
RTCHOUR: current hour
RTCMIN: current minute
RTCSEC: current second
RTCMSEC: current millisecond

From these 7 words a 3 word timer is built:

min	sec
day	hour
year-1900	month

This timer is accessible through procedure Read RTC (refer to 4.32)

The timer can be reset by sending a message to RTC
The message contents will be copied to RTCYR through RTCMIN, and RTCSEC and RTCMSEC are cleared to zero.

Every 100th millisecond the RTC scans through the chain of PCB's:

When a zero SPHASE (refer to fig 3.3.1.a) is encountered, SCYCLE is copied to SPHASE. When a nonzero SPHASE is met, it is decremented.

If the elapse of a delay is awaited, the PCB item SDELAY is inspected:

- if it is zero, the process will be set executing and receive the delay,
- if it is nonzero, SDELAY is decremented.

3.15 Idle Process

The Idle process is CPCI CSS/306.

The scheduling algorithm described in section 3.5 requires that at least one process is ready to execute. This is ensured by having an Idle process for each CPU.

The Idle process executes the following program:

```
START:
      MOVC   Ø          R2
      MON WAITEVENT
      MOVC   100        RØ
      SOB    RØ LOC       ;wait 100 usec.
      JMP       START
```

When scheduled, the Idle process waits 100 usec and then calls the Kernel again.

3.16 Memory Management

The Memory manager is part of CPCI CSS/360.

The memory manager process allocates and deallocates memory on request from user processes.

The memory management functions are invoked by sending system messages to the memory manager process 'MEMMGR'.

Memory is allocated in segments of 128 words.

A segment allocated to a process belongs to that process.

The following functions are performed on request of the memory manager:

- allocate memory
- release memory
- verify that an area of memory belongs to a process and provide the physical address of that memory area
- transfer memory ownership to another process
- release all memory belonging to a process

The format of system messages sent to the memory manager is shown below:

FUNCTION / MSG	Allocate	Release	Verify & convert	Transfer	Release all
+0	1	0	2	4	3
+1	TYPE	MEM	MEM	MEM	-
+2	SIZE	-	-	PCB INDEX OF NEW OWNER	-
+3	CPU	-	-	-	-
+4	RANGE	-	-	-	-
ANSWER					
+0	RESULT	RESULT	RESULT	RESULT	RESULT
+1	MEM	-	MEM	-	-
+2	ADDR	-	ADDR	-	-
+3	PGCPU	-	PGCPU	-	-
+4	SIZE	-	SIZE	-	-

TYPE: defines the use of the memory
  0: for program
  1: for data

MEM: is an internal identification of the memory area

SIZE: is the size of a memory area in words

CPU: is the logical CPU number
0-7: identifies a particular CPU
  8: any CPU suffices
CPU may be specified if a memory area is required to which the corresponding CPU has subbus access.

RANGE:
- lower byte contains the number of the lowest allowed 4 K memory block.
- upper byte contains the number of the highest allowed 4 K memory block.

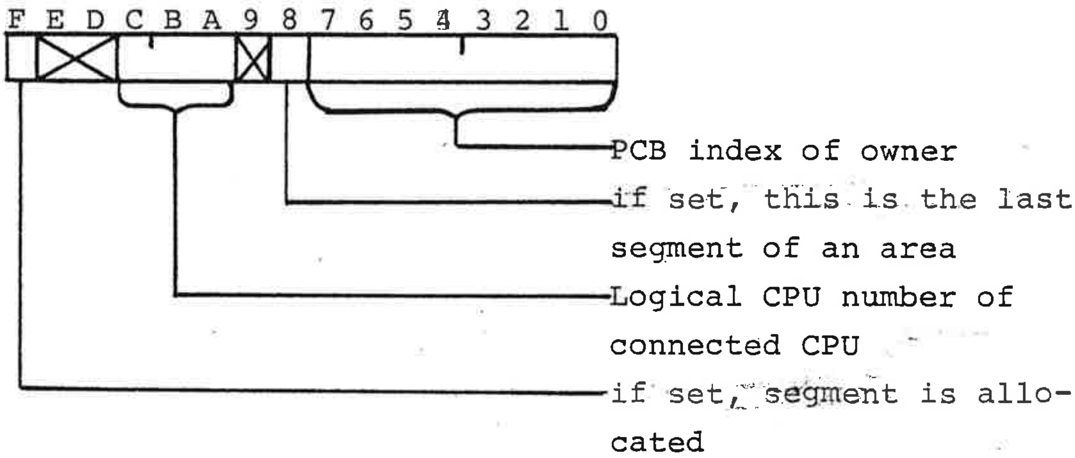
When used for program allocation, RANGE = 0 is interpreted as RANGE = # 0F00 (section 0)
When used for data allocation, RANGE = 0 is interpreted as RANGE = # 3F00 (any section)
When executing on an AMOS CPU, program memory will allways be allocated from section 0.

RESULT: the result of the request:
 =Ø: request process successfully
<>Ø: error

PGCPU:
- upper byte contains logical CPU number (0-7)
- lower byte contains the memory section number (0-3) of the memory area.

The memory manager contains a table of 2048 entries which describes the status of the memory.

Each entry has the following format



The table is preset to: all memory (256K) is connected to CPU Ø.

During initialization ROOT determines if any part of the possible memory space is PROM or does not exist, and if so updates the memory table.

4. FUNCTION DESCRIPTION

This section contains a detailed description of every Kernel procedure and function accessible from outside the Kernel by means of monitor call instructions.

4.1 Local Interrupts

When a process is created, its context item XLOCACT (refer to fig. 3.3.1.c) is initialized to refer to the entry point of a Kernel procedure for handling local interrupts.

When a process encounters a local interrupt, it will therefore automatically invoke this procedure. The procedure determines the local interrupt cause. If it is a timer action, the scheduling algorithm is activated. If it is illegal instruction executed on an AMOS CPU, which would legal on an XAMOS CPU, the instruction is replaced with the corresponding AMOS instruction and re-executed together with preceeding modify instructions, as defined in appendix E. Otherwise (i.e. illegal instruction, parity error, or time-out) the Kernel function Error (synonymous with terminate) is called. This causes the process to enter the STOPPED state, its PCB item SERROR (fig. 3.3.1.a) is set to

error code	:	cause code + ## 81ØØ
error location	:	XLOCRET + XPROG (fig. 3.3.1.c)

and a parent signal is sent to the parent process.

4.2 Wait Event

```
MONITOR FUNCTION AWAIT EVENT  I: (EVENTMASK,ADR,DELAY)
                              O: (EVENTTYPE,EVENT)
INVOKATION:
      MON   WAITEVENT              ;   OR ALTERNATIVELY:
      MON   AWAITEVENT
EVENTMASK IS A BIT MASK WHICH SPECIFIES THOSE EVENTTYPES TO BE AWAITED
 IF TIMEOUT (ELAPSE OF DELAY) IS INCLUDED THE EFFECTIVE DELAY IS DEL
AY + PHASE. (REFER TO SET CYCLE FUNCTION)
IF NONE OF THE EVENTTYPES SPECIFIED HAVE YET OCCURRED, THE PROCESS IS
SUSPENDED UNTIL AN OCCURRENCE.
ELSE IT RETURNS WITH THE MOST URGENT EVENT AS DESCRIBED BELOW.
WHEN ONE OF THE EVENTS OCCURS THE PROCESS IS SCHEDULED FOR EXECUTION.
IT RETURNS WITH THE RESULTING EVENTTYPE(A NUMBER) AND IF THE EVENTTYPE
IS A MESSAGE OR ANSWER TYPE ALSO AN IDENTIFICATION OF THE MESSAGE/ANSW
ER IN EVENT.
THE CONTENTS OF MESSAGES OR ANSWERS ARE DELIVERED IN THE FIVE WORDS
STARTING AT RELATIVE LOCATION ADR.
R0                  DELAY                EVENTTYPE
R1                  ADR                  KEPT
R2                  EVENTMASK            EVENT
R7                  LINK                 DEST
```

The ADR parameter is checked not to point outside the area belonging to the process.

Calling wait event with a zero event mask is equivalent to encountering a timer action, and will not suspend the process, only preempt it.

Symbolic names for event masks and event types are defined in Appendix A. Masks have names BMxxxx and types have names BNxxxx.

Programming Example

In the following example 3 event types are awaited: messages answers and signals:

```
USE BASE
MYBUF: 0  REPEAT 4    ;  5 words
```

```
USE PROG
    MOVC MYBUF          R1;   set up adr
    MOVC BMSIG OR BMMSG
         OR BMANS       R2;   set up event mask
    MON WAITEVENT         ;   wait (mask,adr,-,type,
                              event)
    IEQ  RØ  BNSIG        ;   if type = signal then
    JMP HANDLESIGNAL      ;   go to handle signal
    IEQ RØ  BNMSG         ;   if type = message then
    JMP HANDLEMSG         ;   go to handle message
                          ;   else continue; comment:
                              type is answer.
```

The order in which event occurrences are checked is:

- interrupt
- signal
- answer
- message
- system answer
- system message
- path answer
- path message
- parent signal , and finally
- delay

If wait event is called with a delay = - phase, and with an event mask including delay, the process will always resume processing immediately after the call.

4.3 Inspect Events

```
MONITOR FUNCTION INSPECT EVENTS  I: (EVENTMASK,ADDR,DELAY)
                                 O: (EVENTTYPE,EVENT)
INVOKATION:
      MON   INSPECTEVENTS

INSPECT EVENTS IS INTENDED TO BE USED FOR PROBING FOR OCCURRED EVENTS
WITHOUT RECEIVING THE EVENTS.
EVENTMASK IS A BITMASK WHICH SPECIFIES THOSE TYPES OF EVENTS TO BE
INSPECTED. EVENT TYPES ARE INSPECTED IN THE ORDER OF THEIR PRIORITY.
THE INSPECTION TERMINATES WHEN AN OCCURRED EVENT IS ENCOUNTERED

INSPECT EVENTS DOES NOT CHANGE THE STATE OF THE EVENTS INSPECTED. IN
ORDER TO RECEIVE AN EVENT, THE FUNCTION AWAIT EVENT MUST BE CALLED.
HOWEVER, THE CONTENTS OF MESSAGE AND ANSWER TYPE EVENTS ARE DELIVERED.

R0                DELAY              EVENTTYPE
R1                ADDR               KEPT
R2                EVENTMASK          EVENT
R7                LINK               DEST
```

Inspect events works similarly to wait event. If none of the eventtypes specified have occurred the process is delayed until an occurrance.

K

4.4 Suspend

```
MONITOR FUNCTION SUSPEND
INVOKATION:
      MON   SUSPEND
THE CALLING PROCESS IS SUSPENDED AND THE CPU IS SCHEDULED.
R7                  LINK                DEST
```

This function is only to be used as a tool in other monitor functions.

4.5 Ready

```
MONITOR FUNCTION READY  I:(PCB INDEX)
INVOKATION:
      MON   READY
THE PROCESS IDENTIFIED BY THE PCB INDEX IS LINKED TO ITS READY LIST.
R0                  PCB INDEX           KEPT
R7                  LINK                DEST
```

This function is reserved for use as a tool in other monitor functions.

4.6 Lookup CPU

```
MONITOR FUNCTION LOOKUP CPU  I:(REF(NAME)), O:(CPUCB INDEX)
                                          R: (NOT FOUND,FOUND)
INVOKATION:
      MON   LOOKUPCPU
THE CPU IDENTIFIED BY NAME IS LOOKED UP AND ITS CPUCB INDEX IS RETURNE
D IN CPUID.
R0                  REF(NAME)           CPUCB INDEX
R7                  LINK                DEST
RETURNS:
LINK+0:     NOT FOUND
LINK+1:     FOUND
```

The reference to NAME is checked not to violate the process memory space.

4.7 Set CPU parameter

```
MONITOR FUNCTION SET CPU PARAMETER I:(CPUCB INDEX,PAR,PRIO,VALUE)
                       R:(ERROR,OK)
INVOKATION:
      MON    SETCPUPARAMETER
CHECKS VALIDITY OF THE CPUCB INDEX AND OF THE PARAMETER IDENTIFICATION.
SETS THE VALUE OF THE PARAMETER.
NOTE THAT SOME PARAMETERS ARE A FUNCTION OF THE THE SOFTWARE PRIORITY.
R0                 CPUCB INDEX        KEPT
R1                 PAR                KEPT
R2                 PRIO               KEPT
R4                 VALUE              KEPT
R7                 LINK               DEST
RETURNS:
LINK+0:      ERROR
LINK+1:      OK
```

The parameters which can be modified are (see 3.4).

CCPUID (hardware CPU number)
CIMASK (default interrupt mask)

For each of the CPRIOS software priority levels the following parameters can be set

CSCHRS (schedule reset count)
CSLISZ (size of time slice)
CACTIM (accumulated time)
CHWPRI (hardware (PSW) priority bits)

The priority is specified in PRIO.
The parameter to be set must be specified in PAR (register 1). The following symbolic values of PAR are defined (appendix A).

```
ZCPUNMB   (for CCPUID)
ZINTMSK   (for CIMASK)
ZSCHRCNT  (for CSCHRS)
ZSLICESZ  (for CSLISZ)
ZACCEXECT(for CACTIM)
ZHWPRIO   (for CHWPRI)
```

4.8 Get CPU parameter

```
MONITOR FUNCTION GET CPU PARAMETER I: (CPUCB INDEX,PAR,PRIO)  O:(VALUE,
                    R:(ERROR,OK)
INVOKATION:
      MON   GETCPUPARAMETER
CHECKS VALIDITY OF THE CPUCB INDEX AND OF THE PARAMETER IDENTIFICATION


RETURNS THE VALUE OF THE PARAMETER.
NOTE THAT SOME PARAMETERS ARE A FUNCTION OF THE THE SOFTWARE PRIORITY.
R0                  CPUCB INDEX         VALUE
R1                  PAR                 KEPT
R2                  PRIO                KEPT
R7                  LINK                DEST
RETURNS:
LINK+0:      ERROR
LINK+1:      OK
```

See also 4.7, set CPU parameter.

4.9 Create Process

```
MONITOR FUNCTION CREATE PROCESS    I: (REF(PARAMETER BLOCK))
                                   O: (COMPLETION CODE)
                                   R: (ERROR,DONE)
INVOKATION:
      MON   CREATEPROCESS
THIS FUNCTION ALLOCATES AND INITIALISES A PCB IN ACCORDANCE WITH THE
PARAMETERS IN THE PARAMETER BLOCK.
THE STATE OF THE PROCESS IS SET TO STOPPED.
THE PROCESS DESCRIPTOR (REGISTERS AND BASIC PARAMETERS) ARE ALSO PRESE
T. THE LOGICAL PCB CREATED IS RETURNED IN THE PARAMETER BLOCK IN PARAM
ETER VIDENT.
THE PROGRAM MUST BE LOADED AND MEMORY FOR THE PROCESS MUST BE ALLOCAT
ED BEFORE CREATE PROCESS IS CALLED.
R0                 -              ,   COMPLETION CODE
R1                 REF(PARAMETER BLOCK)KEPT
R7                 LINK               DEST
COMPLETION CODES:
0:     NO ERRORS
1:     NO VACANT PCB'S
2:     REF(PARAMETER BLOCK) VIOLATES SIZE OF CALLING PROCESS
3:     CLASSIFICATION OF PROCESS TO BE CREATED TO HIGH
4:     CAPABILITIES OF PROCESS TO BE CREATED NOT A SUBSET OF PARENT'S.
5:     INVALID NAME
6:     INVALID CPU
7:     INVALID PRIORITY
8:     MESSAGE OVERRUN THREAT
RETURNS:
LINK+0:     ERROR
LINK+1:     DONE
```

The parameter block is checked to lie within the memory space of the calling process.
The layout of the parameter block is defined symbolically in Appendix A and in figure 4.9.a.
The size of a parameter block is VPARLGT words (18 words)

Create process makes the following use of the parameters:

<u>VNAMEØ, VNAME1, VNAME2 (name):</u>
It is checked that the name does not commence with 'P' (lower byte of VNAMEØ). If not all three parameters are zero, it is checked that the name is not already used by an existing process.

LOCATION	NAME	CONTAINS
0	VNAMEØ	Symbolic process name
1	VNAME1	
2	VNAME2	
3	VIDENT	index to PCB allocated
4	VPROG	absolute ref to program
5	VINIT	PROG relative start address
6	VMICRO	PROG relative ref to micro program load module (AMOS)or program page (XAMOS)
7	VCAPAB	process capability
8	VCPU	index of CPU control block
9	VPRIO	required SW priority
10	VLEVEL	preset value for system level
11	VBASE	absolute BASE for process
12	VSIZE	size of area belonging to process
13	VBOUND	preset value for BOUND register
14	VMEMORY	memory allocation parameter
15	VMSGS	max. numb. of message buffers used
16	VUSERID	userid
17	-	-

Fig. 4.9.a CREATE PROCESS PARAMETER BLOCK

If all three parameters are zero, a name is generated and returned in VNAMEØ - VNAME2. The name will be of the form PØØxxx, where xxx is a 3 digit number.

The name becomes the name of the process to be created (PCB parameter SNAME).

VIDENT
In this parameter the PCB index of the created process is returned.

VPROG
This becomes the PROG (program base register) of the created process.

VINIT
This is used to prepare the program counter for the process to be created.

VMICRO
If VMICRO is 0, 1, 2 or 3, it defines the memory section of the program.

If greater than 3, it is used to build a reference to a micro program load module. (PCB item SMICRO).
The scheduling algorithm will ensure that this module is always loaded before the process is executed.

VCAPAB
This becomes the PCB parameter SACCESS.
It is checked that VCAPAB is compatible with the SACCESS of the calling process (refer to 3.3.4).

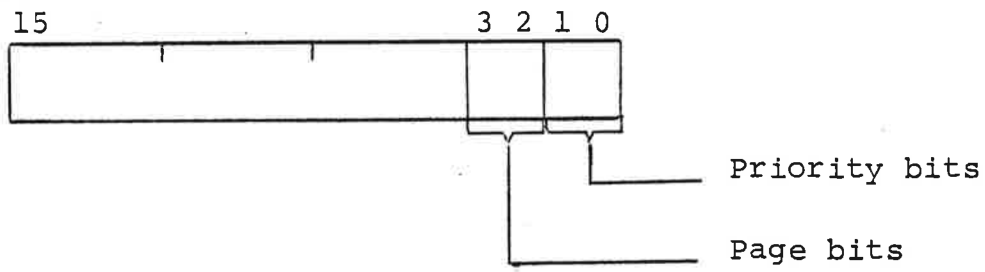
VCPU
Defines the CPU which shall be used to execute the process being created.

VPRIO
Defines the software priority level applicable (refer to 3.4 and 3.5).

VLEVEL
This value is copied to context item XLEVEL (fig. 3.3.1.c).

VBASE
This is used as the absolute BASE for the process to be created.
NOTE that the page and priority bits must be correctly set (this is one reason for restricting access to create process).

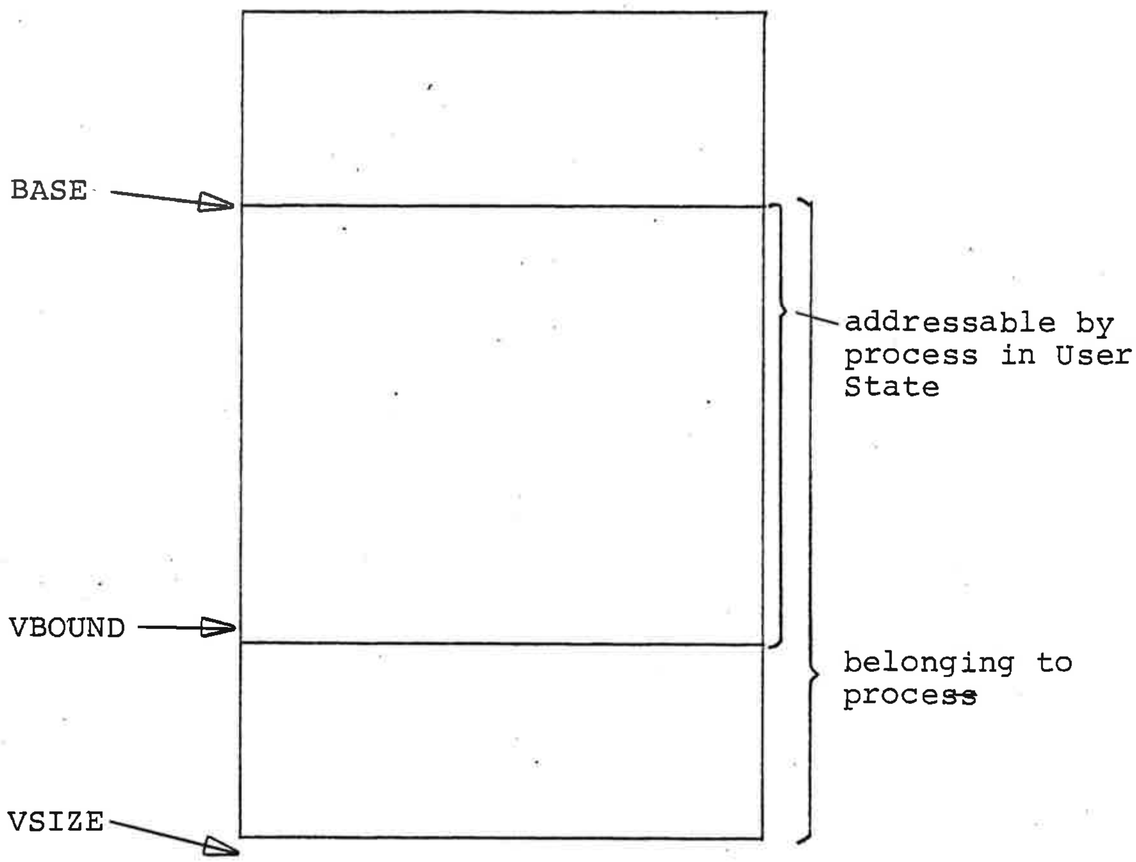
BASE:



VSIZE
This defines the size of the area above BASE belonging to the process. Copied to PCB item SSIZE.

VBOUND
Defines the value of the BOUND register for the process. Copied to context item XBOUND (fig. 3.3.1.c).
Note that VBOUND can at most be VSIZE-1. (see below).

If VLEVEL = 1 (system level), XBOUND is set to -1 in order to allow the process to write everywhere (XAMOS only)..



VMEMORY

This parameter is copied to PCB item SMEMORY. It is not interpreted by the Kernel.

VMSGS

This defines to the Kernel the maximum number of message buffers which the process should be able to allocate. VMGSGS+1 is copied to PCB item SMSGLIM.
The Kernel will only allow creation of a process if the total amount of SMSGLIM for all existing processes does not exceed the total amount of available message buffers.

VUSERID

VUSERID is copied to context locations XUSERIDØ and XUSERID1.

Create process initializes the following context words (fig. 3.3.1.c).

XUSERIDØ
XUSERID1
XCBASE
XPCB
XLEVEL
XBOUND
XBASE
XMOD
XPROG
XPRPC
XTIMER
XPSW
XLOCAT

When the process is created it is in the STOPPED state, and has to be started by a call of start process.

The Kernel has prepared the process to initially execute a call of IOINIT (see ref. 2.4).

The general purpose registers RØ-R7 will be undefined when the process is about to execute the first user defined instruction (at location VPROG+VINIT).

4.10 Start Process

```
 MONITOR FUNCTION START PROCESS I: (CHILD), R: (ERROR,DONE)
 INVOKATION:
       MON   STARTPROCESS
CHECKS THAT THE PCB INDEX CHILD IDENTIFIES A CHILD PROCESS OF THE
CALLING PROCESS AND THAT THE STATE OF THE CHILD IS STOPPED OR TO BE
STOPPED. IF THE CHECK FAILS, RETURN IS MADE TO ERROR.
ELSE THE STATE OF THE CHILD IS CHANGED TO PREEMPTED AND RETURN IS
MADE TO DONE.
R0                  CHILD               KEPT
R7                  LINK                DEST
RETURNS:
LINK+0:      ERROR
LINK+1:      DONE
```

Checks that the PCB index child identifies a child process of the calling process and that the state of the child is stopped (or to be stopped). If the check fails, return is made to error.

Else the state of the child is changed to preempted the process attributes SERROR (refer to 3.3.1) are cleared and return is made to done.

4.11 Stop Process

```
MONITOR FUNCTION STOP PROCESS I: (CHILD), R: (ERROR,DONE)
INVOKATION:
      MON    STOPPROCESS
CHECKS THAT THE PCB INDEX CHILD IDENTIFIES A CHILD PROCESS OF THE
CALLING PROCESS. THE CHILD IS STOPPED (THE TO BE STOPPED FLAG IS SET IN
SSTATE) AND RETURN IS MADE TO DONE.
RO                 CHILD              KEPT
R7                 LINK               DEST
RETURNS:
LINK+0:     ERROR
LINK+1:     DONE
```

Note that if the calling process and the process to be stopped execute on different CPU's, there may be a variable time between the return from call of Stop Process and the time when the process is STOPPED.

If certainty about the process being STOPPED is required, this information may be obtained by a call of Get Attributes (see 4.15).

4.12 Remove Process

```
MONITOR FUNCTION REMOVE PROCESS  I:(CHILD)  O:(MEMORY)  R:(ERROR,DONE)
INVOKATION:
      MON   REMOVEPROCESS
CHECKS THAT THE PCB INDEX CHILD IDENTIFIES A CHILD PROCESS OF THE
CALLING PROCESS.
IF NOT, RETURN IS MADE TO ERROR.
ELSE A REMOVE OPERATION IS PERFORMED ON THE CHILD:
      IF THE CHILD IS EXECUTING, THE REMOVE FLAG IS SET IN ITS PCB
      PARAMETER SSTATE AND THE CALLING PROCESS IS SUSPENDED.
      IF THE CHILD IS WAITING (I E SUSPENDED) IT IS PREPARED TO EXECUT
      A SELFREMOVE PROGRAM AND SCHEDULED.
THE CALLING PROCESS IS SUSPENDED UNTIL THE CHILD HAS COMPLETED ITS
SELFREMOVE. WHEN THIS HAPPENS THE PARENT IS SCHEDULED AND RETURNS WITH
THE MEMORY ALLOCATION PARAMETER FROM THE CHILD IN MEMORY.
WHEN THE CHILD HAS BEEN REMOVED, A CALL OF GET CHILD WILL DELIVER THE
NEXT CHILD
RO                  CHILD               MEMORY
R7                  LINK                DEST
RETURNS:
LINK+0:      ERROR
LINK+1:      DONE
```

4.13 Adopt Process

```
MONITOR FUNCTION ADOPT PROCESS  I:(CHILD)  R:(ERROR,DONE)
INVOKATION:
      MON    ADOPTPROCESS
CHECKS THAT THE PCB INDEX CHILD IDENTIFIES A CHILD PROCESS OF THE
CALLING PROCESS AND THAT THE CALLING PROCESS HAS A PARENT.
IF SO THE CHILD IS MOVED FROM THE CALLING PROCESS TO THE PARENT OF
THE CALLING PROCESS AND RETURN IS MADE TO DONE , ELSE TO ERRCR.
R0                  CHILD               KEPT
R7                  LINK                DEST
RETURNS:
LINK+0:      ERROR
LINK+1:      DONE
```

The calling process transfers its parenthood for one of its child processes to the grandparent of the child.

4.14 Get Child

```
MONITOR FUNCTION GET CHILD O:(CHILD), R:(NONE,DONE)
INVOKATION:
      MON   GETCHILD
DELIVERS THE PCB INDEX OF THE FIRST CHILD IF ANY, ADVANCES THE CHILD
REF TO THE NEXT CHILD AND RETURNS TO DONE, ELSE TO NONE (NO CHILDREN)
SUCCESSIVE CALLS OF GET CHILD WILL STEP THROUGH THE CIRCULAR LIST OF
CHILD PROCESSES, DELIVERING THEIR PCB INDICES ONE BY ONE.
R0                   -                   CHILD (PCB INDEX)
R7                   LINK                DEST
RETURNS:
LINK+0:      NONE
LINK+1:      DONE
```

Successive calls of getchild will step through the circular list of child processes, delivering their PCB indices one by one.

4.15 Get Attributes

```
MONITOR FUNCTION GET ATTRIBUTES  I:(PCB INDEX,RESULT)  R:(ERROR,OK)
INVOKATION:
      MON   GETATTRIBUTES
IT IS CHECKED THAT THE PCB INDEX DENOTES A PROCESS CONTROL BLOCK.
IF NOT, RETURN IS MADE TO ERROR. ELSE TO OK.
THE FOLLOWING PARAMETERS FROM THE PCB ARE DELIVERED AT THE DESTINATION
IDENTIFIED BY THE REFERENCE RESULT:
      SACCESS
      SSTATE
      SERROR (2 WORDS)
      SEXECT (3 WORDS)
      SCREAT (3 WORDS)
R0                  PCB INDEX           KEPT
R1                  RESULT              KEPT
R7                  LINK                DEST
RETURNS:
LINK+0:      ERROR
LINK+1:      OK
```

It is checked that the pointer RESULT does not violate the memory space of the calling process.

4.16 Identify Process, Lookup Process

```
MONITOR FUNCTION IDENTIFY PROCESS  I:(PCB INDEX)  O:(NAME)
INVOKATION:
      MON   IDENTIFYPROCESS
THE NAME OF THE PROCESS IDENTIFIED BY THE PCB INDEX IS RETURNED, IF
THE PROCESS EXISTS, ELSE A DUMMY NAME: "??????" IS RETURNED.
R0                  PCB INDEX          NAME0
R1                  -                  NAME1
R2                  -                  NAME2
R7                  LINK               DEST
```

```
MONITOR FUNCTION LOOKUP PROCESS  I:(REF(NAME))  O:(PCB INDEX)
                                 R: (NOT FOUND, FOUND)
INVOKATION:
      MON   LOOKUPPROCESS
R0                  REF(NAME)          PCB INDEX
R7                  LINK               DEST
RETURNS:
LINK+0:     NOT FOUND
LINK+1:     FOUND
```

It is cehcked that ref. (NAME) does not violate the memory space of the process.

4.17 Send Signal

```
MONITOR FUNCTION SEND SIGNAL I:(RECEIVER)
INVOKATION:
      MON   SENDSIGNAL
SETS THE SIGNAL BOOLEAN IN THE RECEIVER PROCESS. IF THE RECEIVER WAS
AWAITING THE SIGNAL IT IS LINKED TO ITS CPU READY QUEUE.
R0                  REF(NAME OF RECEIVER) KEPT
R7                  LINK                  DEST
```

If the receiver process does not exist, the signal is sent to ROOT (ref. to 3.13).

It is checked that ref. (NAME of RECEIVER) does not violate the memory space of the calling process.

4.18 Send Message

```
MONITOR FUNCTION SEND MESSAGE I:(RECEIVER,MESSAGE), O:(EVENT)
INVOKATION:
      MON   SENDMESSAGE
THE FIVE WORDS REFERENCED BY MESSAGE ARE COPIED TO A MESSAGE BUFFER.
THE CONTENTS OF THE MESSAGE BUFFER ARE DELIVERED TO THE RECEIVER, WHEN
THE RECEIVER CALLS WAIT EVENT WITH A PROPER EVENT MASK.
AN IDENTIFICATION OF THE MESSAGE BUFFER IS RETURNED IN EVENT
AND MAY BE USED AS A PARAMETER IN A SUBSEQUENT AWAIT CALL.
R0                  REF(NAME OF RECEIVER) KEPT
R1                  REF(MESSAGE)          KEPT
R2                  -                     EVENT
R7                  LINK                  DEST
```

Errors:

- If no message buffers are available, the calling process is forced to repeat the call of Send Message. (This situation will not occur due to the restrictive policy for creating new processes (refer to 4.9 and to 6)).

- If the receiver process does not exist, the message will be sent to ROOT (refer to 3.13) which in turn will return a dummy answer.

- If the process by calling send message attempts to use more message buffers than it is allowed to (refer to 4.9) the calling process will call ERROR with a Kerned produced error code: # 1ØD (see also 6).

- If one of the references (to RECEIVER or MESSAGE) violates the address space of the process, the process will call ERROR with a Kernel produced error code: #10C or #106 respectively.

4.19 Send Answer

```
MONITOR FUNCTION SEND ANSWER I:(ANSWER,EVENT)
INVOKATION:
      MON   SENDANSWER
THE FIVE WORDS REFERENCED BY ANSWER ARE SENT TO THE ORIGINAL SENDER
OF THE EVENT.
R1                  REF(ANSWER)        KEPT
R2                  EVENT              EVENT
R7                  LINK               LINK
```

It is checked that ref (Answer) does not violate the memory space of the calling process.

4.20 Await Answer

```
MONITOR FUNCTION AWAIT ANSWER I: (EVENT,ADR,DELAY)
                           O: (EVENTTYPE,EVENT),
INVOKATION:
      MON    AWTANSWER
THE PROCESS IS SUSPENDED UNTIL THE ANSWER OCCURS OR THE DELAY ELAPSES.
R0                  DELAY               EVENTTYPE
R1                  ADR                 KEPT
R2                  EVENT               EVENT
R7                  LINK
```

This function is used to wait for a specific answer.

It is checked that the pointer ADR does not violate the memory space of the calling process.

It is checked that EVENT is the index of a message buffer sent by the calling process and that no answer has yet been delivered.

4.21 Send System Message

```
MONITOR FUNCTION SEND SYSTEM MESSAGE I:(RECEIVER,MESSAGE), O:(EVENT)
INVOKATION:
      MON    SENDSYSTEMMESSAGE
R0                  REF(NAME OF RECEIVER) KEPT
R1                  REF(MESSAGE)          KEPT
R2                  -                     EVENT
R7                  LINK                  DEST
```

This function is similar to send message (refer to 4.18).

4.22 Send System Answer

```
MONITOR FUNCTION SEND SYSTEM ANSWER I:(ANSWER,EVENT)
INVOKATION:
      MON    SENDSYSTEMANSWER
SIMILAR TO SEND ANSWER.
R1                   REF(ANSWER)        KEPT
R2                   EVENT              EVENT
R7                   LINK               LINK
```

Refer to Send answer 4.19.

4.23 Await System Answer

```
MONITOR FUNCTION AWAIT SYSTEM ANSWER
                        I: (EVENT,ADR,DELAY), O: (EVENTTYPE,EVENT)
INVOKATION:
      MON   AWTSYSTEMANSWER
THE PROCESS IS SUSPENDED UNTIL THE ANSWER OCCURS OR THE DELAY ELAPSES.
R0                  DELAY                   EVENTTYPE
R1                  ADR                     KEPT
R2                  EVENT                   EVENT
R7                  LINK                    DETS
```

This function is similar to Await answer (ref. to 4.20).

4.24 Open Path

```
MONITOR FUNCTION OPEN PATH I:(RECEIVER), O:(EVENT)
INVOKATION:
      MON   OPENPATH
LOCATES (LOOKS UP) THE RECEIVER WHICH IS DENOTED BY NAME AND ALLOCATES
AND INITIALISES A MESSAGE BUFFER WHICH CAN BE USED IN SUBSEQUENT
CALLS OF SEND PATH MESSAGE/ SEND PATH ANSWER. THE BUFFER IS IDENTIFIED
BY EVENT.
R0                  REF(NAME)           KEPT
R2                  -                   EVENT
R7                  LINK                KEPT
```

It is checked that ref (Name) does not violate the memory space of the calling process.

If the receiver cannot be found, the path will be opened to ROOT (refer to 3.13).

4.25 Close Path

```
MONITOR FUNCTION CLOSE PATH I:(EVENT)

INVOKATION:
      MON   CLOSEPATH
RELEASES A MESSAGE BUFFER WHICH WAS ALLOCATED BY A PREVIOUS CALL OF
OPEN PATH.
A PATH CAN ONLY BE CLOSED BY THE PROCESS WHICH OPENED THE PATH AND ONL
Y IF THE MESSAGE BUFFER RESIDES WITH THIS PROCESS, I.E. IF IT HAS NEVE
R BEEN SENT BY A SEND PATH MESSAGE CALL OR IF IT HAS BEEN RECEIVED AFT
ER A SEND PATH ANSWER CALL.
R2                  EVENT               DEST
R7                  LINK                LINK
```

4.26 Send Path Message

```
MONITOR FUNCTION SEND PATH MESSAGE I:(MESSAGE,EVENT)
INVOKATION:
      MON    SENDPATHMESSAGE
THE FIVE WORDS IDENTIFIED BY REF(MESSAGE) ARE SENT TO THE PROCESS FOR
WHICH THE PATH WAS OPENED. THE WORDS ARE SENT USING THE MESSAGE BUFFER
WHICH WAS ALLOCATED WHEN OPEN PATH WAS CALLED.
R1                     REF(MESSAGE)        KEPT
R2                     EVENT               EVENT
R7                     LINK                LINK
```

The call of this function must have been preceeded by a call of open path.

4.27 Send Path Answer

```
MONITOR FUNCTION SEND PATH ANSWER I:(ANSWER,EVENT)
INVOKATION:
      MON   SENDPATHANSWER
SIMILAR TO SEND ANSWER.
```

Refer to Send answer 4.19.

4.28 Await Path Answer

```
MONITOR FUNCTION AWAIT PATH ANSWER
                        I: (EVENT,ADR,DELAY), O: (EVENTTYPE,EVENT)
INVOKATION:
      MON    AWTPATHANSWER
THE PROCESS IS SUSPENDED UNTIL THE ANSWER OCCURS OR THE DELAY ELAPSES.
R0                  DELAY                 EVENTTYPE
R1                  ADR                   KEPT
R2                  EVENT                 EVENT
R7                  LINK
```

Similar to Await answer (refer to 4.20).

4.29 Identify sender

```
MONITOR FUNCTION IDENTIFY SENDER  I:(EVENT)  O:(PCB INDEX)
                                  R:(ERROR,OK)
INVOKATION:
      MON    IDENTIFYSENDER
CHECKS THAT THE EVENT IS RECEIVED BY THE CALLING PROCESS.
IF NOT RETURN IS MADE TO ERROR.
DELIVERS THE PCB INDEX OF THE SENDING PROCESS AND RETURNS TO OK.
R0                  -                  PCB INDEX
R2                  EVENT              KEPT
R7                  LINK               DEST
RETURNS:
LINK+0:     ERROR
LINK+1:     OK
```

This function is used to deliver the PCB index of a sender process from which the calling process has received a message, system message, or path message.

4.30 Save Event

```
MONITOR FUNCTION SAVE EVENT I:(EVENT)
INVOKATION:
       MON    SAVEEVENT
IF THE EVENT IS A RECEIVED MESSAGE (ORRDINARY, SYSTEM, OR PATH) (E.G.
DELIVERED BY A CALL OF WAIT EVENT) OR THE FIRST MESSAGE OR ANSWER IN
AN EVENT QUEUE (E.G. DELIVERED BY A CALL OF INSPECT EVENTS) THE EVENT
IS MOVED TO THE TAIL OF THE CORRESPONDING LIST OF SAVED EVENTS.


R2                  EVENT               KEPT
R7                  LINK                DEST
```

Suppose a message is received and the receiving process is not prepared to process it e.g. because another message (not yet received) must be handled first. The process can defer processing of the message by calling Save event, and at a later time resume processing of it by calling Recover events.

4.31 Recover Events

```
MONITOR FUNCTION RECOVER EVENTS  I:(EVENTTYPE)
INVOKATION:
      MON   RECOVEREVENTS
IF THE EVENTTYPE IS A MESSAGE OR ANSWER TYPE (ORDINARY, SYSTEM, OR PAT
THE CORRESPONDING LIST OF SAVED EVENTS IS TRANSFERRED TO THE FRONT OF
THE CORRESPONDING EVENT QUEUE.
R2                  EVENTTYPE           KEPT
R7                  LINK                DEST
```

This function is to be used if reception of messages has been deferred by a call of Save event. After a call of recovery events the messages will be delivered by calling wait event.

4.32 Read RTC

```
MONITOR PROCEDURE READ RTC  O:(YEAR-1900,MONTH,DAY,HOUR,MIN,SEC)
INVOKATION:
      MON    READRTC
R0                    -                   LSB: SEC     MSB: MIN
R1                    -                   LSB: HOUR    MSB: DAY
R2                    -                   LSB: MONTH   MSB: YEAR-1900
R7                    LINK                DEST
```

4.3.2.1 Read System Time

```
MONITOR PROCEDURE READ SYSTEM TIME  O:( SYSTIME)
INVOKATION:
      MON    READSYSTIME

RETURNS THE SYSTEM ELAPSE TIME IN MILLI SECONDS

R0                    -                SYSTIME (LEAST SIGNIFICANT PART)
R1                    -                SYSTIME
R2                    -                SYSTIME (MOST SIGNIFICANT TIME)
R7                    LINK             DEST
```

4.33 Set Cycle

```
MONITOR FUNCTION SET CYCLE I:(CYCLE)
INVOKATION:
      MON   SETCYCLE
THE CYCLE WILL BE USED BY THE RTC PROCESS TO INITIALISE A PHASE.
THE PHASE IS DECREMENTED FOR EVERY 10TH OCCURRENCE OF THE 10 MS REAL
TIME CLOCK INTERRUPT. WHEN THE PHASE REACHES 0, IT IS RESET TO CYCLE.
WHEN A WAIT OPERATION INCLUDES THE TIMEOUT EVENT (ELAPSE OF A DELAY)
THE EFFECTIVE DELAY IS THE TOTAL OF THE DELAY PARAMETER AND PHASE.
R0                  CYCLE               KEPT
R7                  LINK                KEPT
```

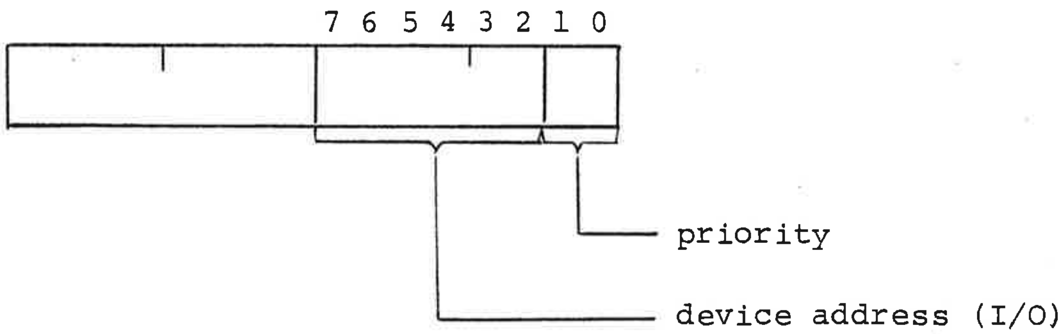
When a process is created, its cycle is set to zero. (PCB parameter SCYCLE). If the cycle is set to a non zero value by a call of Set cycle, this value will be used to reset and preset its phase (PCB parameter SPHASE).

The phase can be used to implement a synchronization to real time which is independent of the time elapsed between the wake up of a process and its next call of wait event (because its phase is constantly maintained by the RTC).

4.34 Reserve Interrupt

```
MONITOR FUNCTION RESERVE INTERRUPT I:(DEVPR), O:(INTRPT)
INVOKATION:
      MON   RESERVEINTERRUPT
CHECKS DEVPR (PRIORITY, DEVICE ADR). IF DEVPR IS VALID AND THE CORRES-
PONDING INTERRUPT IS NOT RESERVED BY ANOTHER PROCESS, THE CALLING PROC
ESS IS INSERTED AS RESERVER AND A LOGICAL REFERENCE IS RETURNED (INTR
PT). THE SAME PARAMETER IS INSERTED IN THE PCB AS THE CURRENTLY AWAITE
D INTERRUPT.
IF THE INTERRUPT IS ALREADY RESERVED BY A PROCESS, A VALUE OF -1 IS
RETURNED IN INTRPT.
R1                  DEVPR               INTRPT
R7                  LINK                LINK
```

DEVPR contains the device address and priority as follows:



4.35 Release Interrupt

```
MONITOR FUNCTION RELEASE INTERRUPT I:(INTRPT)
INVOKATION:
      MON    RELEASEINTERRUPT
IF INTRPT IS VALID AND CORRESPONDS TO AN INTERRUPT RESERVED BY THE
CALLING PROCESS, THE INTERRUPT IS RELEASED. OTHERWISE NO ACTION IS
TAKEN.
R1                  INTRPT              KEPT
R7                  LINK                LINK
```

A

4.36 Clear Interrupt

```
MONITOR FUNCTION CLEAR INTERRUPT I:(INTRPT)
INVOKATION:
      MON   CLEARINTERRUPT
CHECKS THE VALIDITY OF INTRPT AND THAT THE INTERRUPT IS RESERVED BY
THE CALLING PROCESS. THE INTERRUPT COUNTER IS CLEARED TO ZERO.
R1                 INTRPT              INTRPT
R7                 LINK                LINK
```

4.37 Set Interrupt

```
MONITOR FUNCTION SET INTERRUPT I:(INTRPT)
INVOKATION:
      MON   SETINTERRUPT
THIS FUNCTION VALIDATES THE INTRPT. IF IT CORRESPONDS TO AN INTERRUPT
RESERVED BY THE CALLING PROCESS, THE INTRPT PARAMETER IS INSERTED IN
THE PCB AS THE CURRENTLY AWAITED INTERRUPT.
R1                   INTRPT              KEPT
R7                   LINK                LINK
```

4.38 Inclusion of New Monitor Procedures

```
MONITOR FUNCTION INITIALISE MONITOR FUNCTION
INVOKATION:
      MON   MONINIT
PREPARES THE MONITOR JUMP TABLE TO CONTAIN ABSOLUTE POINTERS TO SPECI
FIED PROCEDURE ENTRIES.
THE INITIALISE FUNCTION CALL MUST BE SUCCEEDED BY A PARAMETER LIST:
      LOC, (FUNCTION,ENTRY).....(FUNCTIN,ENTRY),0
             FUNCTION: MUST BE A VALUE IN THE RANGE (64,255) SIGNIFYING
                     THE MONIOTR CALL ARGUMENT.
             ENTRY:  MUST BE A PROG REL REF TO THE CORRESPONDING PROCEDU
                     RE/FUNCTION.
R7                   LINK                DEST
```

It is checked that the entries to be initialized are not already used. If this check fails, the calling process is stopped by entering an infinite loop.

Programming Example

The procedure with label NEW is to be entered corresponding to an invokation by MON NEWPROC:

```
NEW:         .
             .
             .
             .
        MON  MONINIT
        LOC, NEWPROC, NEW, Ø
```

4.39 Error/Terminate

```
MONITOR FUNCTION ERROR I:(ERRORCODE,ERRORLOCATION)
INVOKATION:
     MON   ERROR                        ;   OR ALTERNATIVELY:
     MON   TERMINATE
BIT 15 OF THE ERROR CODE IS SET.
THE CALLING PROCESS IS SUSPENDED WITH SSTATE=STOPPED AND THE ERROR
CODE AND LOCATION ARE STORED IN SERROR.
A PARENT SIGNAL IS SENT TO THE PARENT OF THE CALLING PROCESS.
R0                 ERRORCODE
R1                 ERROR LOCATION
R7                 LINK
```

The following convention is adapted for error codes:

o the upper byte defines a subsystem which generated the error code:

Ø: utility generated code
1: Kernel generated code
2: I/O system generated code
3,4,5: File Management System generated code
6: Device driver generated code
7: Pascal Runtime generated code

o the lower byte contains a subsystem defined error code.

The error code Ø is used to express a normal termination.

4.40 Miscallaneous Functions

4.40.1 Write RTC

```
MONITOR PROCEDURE WRITE RTC  I:(YEAR-1900,MONTH,DAY,HOUR,MIN,SEC)
INVOKATION:
      MON    WRITERTC
R0                     LSB: SEC    MSB: MIN
R1                     LSB: HOUR   MSB: DAY
R2                     LSB: MONTH  MSB: YEAR-1900
R7                     LINK                  DEST
```

This procedure is used by the RTC driver to update the real time clock.

4.40.2 Clean Messages

```
MONITOR FUNCTION CLEAN MESSAGES.
INVOKATION:
      MON   CLNMESSAGE
CLEANS UP AFTER A PROCESS WHICH HAS USED THE MESSAGE SYSTEMS.
R7                  LINK                 DEST
```

This function is called by the Kernel during removal of a process.

4.41 Create Region

```
PROCEDURE CREATE_REGION I:(REF(CB)) O:(CC) R:(ERROR,OK)

INVOKED BY:
      MON    REGION,    RCREATE

INITIALIZES A CRITICAL REGION CONTROL BLOCK (CRCB).
IT IS CHECKED THAT A REGION DOES NOT ALREADY EXIST WITH A NAME AS
SPECIFIED IN THE CREATION BLOCK (CB).
IF POSSIBLE A CRCB IS ALLOCATED AND INITIALIZED AS SPECIFIED IN THE CB

REGISTER            CALL                EXIT
R0                  REF(CB)             KEPT
R7                  LINK                COMPLETION_CODE

RETURNS:
LINK+1:     ERROR (SPECIFIED IN THE COMPLETION_CODE)
LINK+2:     OK

POSSIBLE ERRORS:
      ILLEGAL NAME                (ZCRILLNAME)
      NO CRCB'S                   (ZCROVFL)
      REF(CB) VIOLATES PROCESS SIZE
                                  (ZCRPSZ)
```

4.41.1 Region completion codes

The following completion codes are defined for critical regions:

Ø no errors
1 unknown function (ZCRUNF)
2 parameter ref. violates address space of process (ZCRPSZ)
3 unknown region (ZCRUNR)
4 region not entered (ZCRILLSTA)
5 invalid process (ZCRPCB)
6 invalid region name (ZCRILLNAM)
7 address violation in VS (ZCRVSZ)
8 too many regions (ZCROVFL)

4.41.2 Region Parameter Definitions

The parameters used when calling the region procedures are defined formally in this section using pascal notation.

```
Type    Region-Name    = record
               name:     array [0..2] of integer;
               name-ident:  integer
        end;

Type    Variable-Space    = record
               addr, page, size: integer
        end;

Type    Region-Creation-Block    = record
               name:  region-name;
               VS:    variable-space
        end;
```

4.42 Enter Region

```
PROCEDURE ENTER_REGION I:(REF(NAME)) O:(CC) R:(ERROR,OK)
INVOKED BY:
      MON   REGION,    RENTER

IT IS CHECKED THAT THE REGION SPECIFIED BY NAME EXISTS.
IF NO PROCESS IS IN THE ENTERED STATE FOR THE REGION, THE CALLING PROC
IS SET IN THE ENTERED STATE, AND RETURN IS MADE TO OK.
OTHERWISE, THE PROCESS IS SUSPENDED AND LINKED TO THE TAIL OF THE 'ENT
QUEUE' FOR THE REGION. HERE IT IS DELAYED UNTIL ALL PROCESSES ALREADY
WAITING TO ENTER HAVE HAD THEIR TURN.

REGISTER            CALL                EXIT
R0                  REF(NAME)           KEPT
R7                  LINK                COMPLETION_CODE

RETURNS:
LINK+1:      ERROR (AS SPECIFIED IN THE COMPLETION CODE)
LINK+2:      OK

POSSIBLE ERRORS:
      ILLEGAL NAME                      (ZCRILLNAM)
      REF(NAME) VIOLATES PROCESS SIZE   (ZCRPSZ)
      REGION UNKNOWN                    (ZCRUNR)
```

4.43 Leave Region

```
PROCEDURE LEAVE_REGION I:(REF(NAME)) O:(CC) R:(ERROR,OK)
INVOKED BY:
      MON   REGION,    RLEAVE

IT IS CHECKED THAT THE REGION EXISTS, AND THAT THE PROCESS IS IN THE
ENTERED STATE FOR THIS REGION.
THE STATE OF THE PROCESS VIS A VIS THE REGION IS CHANGED TO 'REGION
LEFT'.
IF THE 'WAIT QUEUE' IS NOT EMPTY AND THE DIRTY FLAG IS SET THEN
THE 'WAIT QUEUE' IS MOVED TO THE HEAD OF THE 'ENTER QUEUE'.
THE DIRTY FLAG IS CLEARED.
IF THEN THE 'ENTER QUEUE' IS NOT EMPTY, THE FIRST PROCESS IN THE
QUEUE IS DEQUEUED, PUT IN THE ENTERED STATE, AND SCHEDULED FOR
EXECUTION.
THE CALLING PROCESS CONTINUES.

REGISTER             CALL                 EXIT
R0                   REF(NAME)            KEPT
R7                   LINK                 COMPLETION_CODE

RETURNS:
LINK+1:     ERROR (SPECIFIED IN THE COMPLETION_CODE)
LINK+2:     OK

POSSIBLE ERRORS:
      ILLEGAL NAME                                      (ZCRILLNAM)
      REF(NAME) VIOLATES THE PROCESS SIZE               (ZCRPSZ)
      UNKNOWN REGION                                    (ZCRUNR)
      REGION NOT ENTERED                                (ZCRILLSTA)
```

4.44 Wait Region

```
PROCEDURE WAIT_REGION I:(REF(NAME)) O:(CC) R:(ERROR,OK)
INVOKED BY:
      MON   REGION,    RWAIT

IT IS CHECKED THAT THE REGION EXISTS, AND THAT THE CALLING PROCESS
IS IN THE ENTERED STATE.
THE PROCESS STATE VIS A VIS THIS REGION IS CHANGED TO 'WAITING TO RE_
ENTER'.
IF THE 'WAIT QUEUE' IS NOT EMPTY AND THE DIRTY FLAG IS SET THEN THE
'WAIT QUEUE' IS MOVED TO THE HEAD OF THE 'ENTER QUEUE'.
THE DIRTY FLAG IS CLEARED.
IF THE 'ENTER QUEUE' IS THEN NOT EMPTY, THE FIRST PROCESS IN THE QUEUE
IS DEQUEUED, PUT IN THE ENTERED STATE, AND SCHEDULED FOR EXECUTION.
THE CALLING PROCESS IS LINKED TO THE TAIL OF THE WAIT QUEUE AND
SUSPENDED.

REGISTER            CALL               EXIT
R0                  REF(NAME)          KEPT
R7                  LINK               COMPLETION_CODE

RETURNS:
LINK+1:      ERROR (SPECIFIED IN THE COMPLETION CODE)
LINK+2:      OK

POSSIBLE ERRORS:
      AS FOR LEAVE_REGION
```

4.45 Get Item

```
PROCEDURE GET_ITEM I:(REF(NAME),ITEM INDEX) O:(ITEM,CC) R:(ERROR,OK)
INVOKED BY
      MON   REGION,    RGET

IT IS CHECKED THAT THE REGION EXISTS AND THAT THE PROCESS IS IN THE
ENTERED STATE.
THE WORD IN THE VARIABLE SPACE CONTROLLED BY THE REGION, THE ADDRESS
OF WHICH IS
      PAGE:         REGION.CRSTA
      WDADDR:       REGION.CRADDR + ITEM INDEX
IS RETURNED IN ITEM, PROVIDED THAT
      ITEM INDEX <= REGION.CRSIZE

REGISTER            CALL                EXIT
R0                  REF(NAME)           KEPT
R1                  ITEM INDEX          KEPT
R2                  -                   ITEM
R7                  LINK                COMPLETION_CODE

RETURNS:
LINK+1:     ERROR (SPECIFIED IN THE COMPLETION_CODE)
LINK+2:     OK

POSSIBLE ERRORS:
      UNKNOWN REGION                    (ZCRUNR)
      NOT ENTERED STATE                 (ZCRILLSTA)
      REF(NAME) VIOLATES PROCESS SIZE   (ZCRPSZ)
      ITEM INDEX VIOLATES VS SIZE       (ZCRVSZ)
```

4.46 Put Item

```
PROCEDURE PUT_ITEM I:(REF(NAME), ITEM INDEX, ITEM) O:(CC) R:(ERROR,OK)
INVOKED BY:
      MON   REGION,    RPUT

THIS FUNCTION IS SIMILAR TO GET_ITEM, EXCEPT THAT THE ITEM IS STORED I
THE VARIABLE SPACE.
THE REGION DIRTY FLAG IS SET.

REGISTERS           CALL                EXIT
R0                  REF(NAME)           KEPT
R1                  ITEM INDEX          KEPT
R2                  ITEM                KEPT
R7                  LINK                COMPLETION_CODE

RETURNS:           REFER TO GET_ITEM
POSSIBLE ERRORS:   REFER TO GET_ITEM
```

4.47 Get n Items

```
PROCEDURE GET_N_ITEMS I:(REF(NAME),ITEM INDEX, DESTINATION,N)
                      O: (CC) R: (ERROR,OK)


INVOKED BY:
      MON   REGION,    RGETN

IT IS CHECKED THAT THE REGION EXISTS, AND THAT THE CALLING PROCESS IS
THE ENTERED STATE.
THE RANGE OF ADDRESSES DEFINED BY ITEM INDEX AND N ARE CHECKED TO LIE
WITHIN THE VARIABLE SPACE OF THE REGION.
IT IS ALSO CHECKED THAT THE RANGE OF ADDRESSES DEFINED BY DESTINATION
AND N LIE WITHIN THE CALLING PROCESS.
THE N ITEMS IN THE VARIABLE SPACE DEFINED BY THE ADDRESS RANGE:
      PAGE:         REGION.CRSTA
      WDADDR:       REGION.CRADDR + ITEM INDEX,......
                    ...., REGION.CRADDR + ITEM INDEX +N -1
ARE DELIVERED IN THE N LOCATIONS
      DESTINATION,......, DESTINATION + N -1

REGISTER            CALL                EXIT
R0                  REF(NAME)           KEPT
R1                  ITEM INDEX          KEPT
R2                  DESTINATION (REL)   KEPT
R3                  N (WORDS)           KEPT
R7                  LINK                COMPLETION_CODE

RETURNS:
LINK+1:     ERROR  (SPECIFIED IN COMPLETION CODE)
LINK+2:     OK

POSSIBLE ERRORS:
      UNKNOWN REGION                     (ZCRUNR)
      REGION IS NOT ENTERED              (ZCRILLSTA)
      REF(NAME) VIOLATES PROCESS SIZE    (ZCRPSZ)
      DESTINATION,N VIOLATES PROCESS SIZE (ZCRPSZ)
      ITEM INDEX,N VIOLATES VS SIZE      (ZCRVSZ)
```

4.48 Put n Items

```
PROCEDURE PUT_N_ITEMS I:(REF(NAME), ITEM INDEX, SOURCE,N)
                      O:(CC)   R:(ERROR,OK)

INVOKED BY
      MON    REGION,     RPUTN

SIMILAR TO GET_N_ITEMS EXCEPT FOR THE DIRECTION OF MOVING DATA.
THE DIRTY FLAG IS SET.

REGISTER              CALL                 EXIT
R0                    REF(NAME)            KEPT
R1                    ITEM INDEX           KEPT
R2                    SOURCE  (REL)        KEPT
R3                    N (WORDS)            KEPT
R7                    LINK                 COMPLETION_CODE

RETURNS:                 REFER TO GET_N_ITEMS
POSSIBLE ERRORS:         REFER TO GET_N_ITEMS
```

K

4.49 Copy n Items

```
PROCEDURE COPY_N_ITEMS I:(REF(NAME),ITEM INDEX, DESTINATION,N)
                       O: (CC) R: (ERROR,OK)

INVOKED BY:
      MON   REGION,    RCOPYN

IT IS CHECKED THAT THE REGION EXISTS
THE RANGE OF ADDRESSES DEFINED BY ITEM INDEX AND N ARE CHECKED TO LIE
WITHIN THE VARIABLE SPACE OF THE REGION.
IT IS ALSO CHECKED THAT THE RANGE OF ADDRESSES DEFINED BY DESTINATION
AND N LIE WITHIN THE CALLING PROCESS.
THE N ITEMS IN THE VARIABLE SPACE DEFINED BY THE ADDRESS RANGE:
      PAGE:          REGION.CRSTA
      WDADDR:        REGION.CRADDR + ITEM INDEX,......
                     ...., REGION.CRADDR + ITEM INDEX +N -1
ARE DELIVERED IN THE N LOCATIONS
      DESTINATION,......, DESTINATION + N -1

REGISTER             CALL                 EXIT
R0                   REF(NAME)            KEPT
R1                   ITEM INDEX           KEPT
R2                   DESTINATION (REL)    KEPT
R3                   N (WORDS)            KEPT
R7                   LINK                 COMPLETION_CODE

RETURNS:
LINK+1:      ERROR  (SPECIFIED IN COMPLETION CODE)
LINK+2:      OK

POSSIBLE ERRORS:
      UNKNOWN REGION                        (ZCRUNR)
      REF(NAME) VIOLATES PROCESS SIZE       (ZCRPSZ)
      DESTINATION,N VIOLATES PROCESS SIZE   (ZCRPSZ)
      ITEM INDEX,N VIOLATES VS SIZE         (ZCRVSZ)
```

4.50 Buffer Allocation Procedures

The following buffer allocation procedures are provided via CSS/361:

4.50.1 Get Buffer

```
MONITOR PROCEDURE GET_BUFFER I:(SIZE), O:(MEMORY,ADDRESS,PAGE,SIZE)
                    R:(NOT_POSSIBLE,OK)
INVOKED BY:         MON         GETBUF
ALLOCATES A MEMORY AREA OF AT LEAST SIZE WORDS. THE ACTUAL SIZE, ADDRE
SS AND PAGE ARE RETURNED.


PAGE MAY BE USED DIRECTLY AS A PSW VALUE WHEN SUBSEQUENTLY ACCESSING
THE BUFFER.
R0                  -                   MEMORY (ALLOCATION PARAMETER)
R1                  -                   ADDRESS (ABS WORD)
R2                  -                   PAGE
R3                  SIZE                SIZE (UPDATED)
R7                  LINK                DEST
RETURNS:
LINK+0:      NOT_POSSIBLE
LINK+1:      OK
```

4.50.2 Release Buffer

```
MONITOR PROCEDURE RELEASE_BUFFER I:(MEMORY), R:(FAULT,OK)
INVOKED BY:         MON         RELBUF
VERIFIES THAT THE MEMORY DEFINED BY THE MEMORY ALLOCATION PARAMETER
MEMORY BELONGS TO THE CALLING PROCESS.
RELEASES THE MEMORY INTO THE VACANT AREA POOL.
R0                  MEMORY              DEST
R1                  -                   DEST
R7                  LINK                DEST
RETURNS:
LINK+0:      FAULT
LINK+1:      OK
```

4.50.3 Get Address

```
MONITOR PROCEDURE GET_ADDRESS I:(MEMORY), O:(ADDRESS,PAGE,SIZE)
                   R:(FAULT,OK)
INVOKED BY:        MON        ADRBUF
VERIFIES THAT THE MEMORY DEFINED BY THE MEMORY ALLOCATION PARAMETER
MEMORY BELONGS TO THE CALLING PROCESS.
CONVERTS MEMORY TO AN ADDRESS, A PAGE AND A SIZE.
PAGE MAY BE USED DIRECTLY AS A PSW VALUE WHEN SUBSEQUENTLY ACCESSING
THE BUFFER.
R0                 MEMORY              KEPT
R1                 -                   ADDRESS
R2                 -                   PAGE
R3                 -                   SIZE
R7                 LINK                DEST
RETURNS:
LINK+0:      FAULT
LINK+1:      OK
```

4.50.4 Clean Memory

```
MONITOR PROCEDURE CLEAN_MEMORY
INVOKED BY: MON    CLNMEM
ALL MEMORY BELONGING TO THE CALLING PROCESS IS RELEASED.
R0                 -                   DEST
R1                 -                   DEST
R7                 LINK                DEST
```

4.51 Double Precision Arithmetic

The following 32 bit multiply and divide functions are provided via CSS/316.

4.51.1 Multiply Long

Invoked by MON MULTIPLY LONG

```
MONITOR PROCEDURE MULTIPLYLONG (OP1,OP2, REF.RESULT,OVERFLOW)
THE PROCEDURE MULTIPLIES THE TWO DOUBLE WORD OPERANDS OP1 AND
OP2. THE RESULT IS DELIVERED AT FOUR LOCATIONS STARTING AT
REF. RESULT.
IF THE RESULT HAS MORE THAN 32 SIGNIFICANT BITS
(-2**31<=RESULT<=2**31-1) THE OVERFLOW FLAG IN
PSW WILL BE SET TO TRUE ELSE TO FALSE. THAT IS THE SIZE CAN BE
TESTED BY JVN.
OPERANDS OP1 AND OP2 ARE CONSIDERED 32 BIT OPERANDS
IN 2'S CORNPLEMENT REPRESENTATION. EACH OPERAND IS
CONTAINED IN TWO WORDS: A LEAST SIGNIFICANT PART (LOP) AND
A MOST SIGNIFICANT PART (MOP)
      REGISTER      CALL                EXIT
      R0            LOP1                0.RESULT
      R1            MOP1                1.RESULT
      R2            LOP2                2.RESULT
      R3            MOP2                3.RESULT
      R4                                DESTROYED
      R5            REF.RESULT          REF.RESULT
      R7            LINK                DESTROYED
THE RATIONALE FOR THE IMPLEMENTATION IS AS FOLLOWS:
      LET   A=(A(N),A(N-1),---,A(0)) BE A BINARY VECTOR
      THIS VECTOR CAN REPRESENT EITHER AN UNSIGNED
            U(N+1)(A)= A(N)*2**N+A(N-1)*2**(N-1)+...+A(0)
      OR A SIGNED INTEGER IN 2'S COMPLEMENT:
            S(N+1)(A)=-A(N)*2**N+A(N-1)*2**(N-1)+...+A(0)
      NOW LET
            F(N+1)(A)= A(N)*2**(N+1)
      THEN
            U(N+1)=S(N+1)+F(N+1)
      THE FOLLOWING IS THEN VALID FOR
            D=(D(31),----,D10))
            M=(D(31),---,D(16)), MOST SIGNIFICANT PART OF D
            L=(D(15),---,D(0)) , LEAST SIGNIFICANT PART OF D
            AND D',M',L':
            S(32)(D)*S(32)(D')=

                 ((2**16)*(S(16)(M)+D(15))+S(16)(L))*((2**16)*(S(16)(M')+D'(15))+
                                                        S(16)(L'))
```

4.51.2 Divide Long

Invoked by: MON DIVIDELONG

```
MONITOR PROCEDURE DIVIDELONG (OP1,OP2,RESULT,OVERFLOW)
THIS PROCEDURE DIVIDES A 2 WORD 2'S COMPLEMENT OPERAND -OP1 - BY A
2 WORD 2'S COMPLEMENT OPERAND - OP2 - AND DELIVERS THE QUOTIENT
AS A 2 WORD 2'S COMPLEMENT NUMBER AT RESULT.
THE OVERFLOW FLAG IN PSW WILL BE SET TRUE IF DIVISION BY 0 IS ATTEMPTED
OTHERWISE THE FLAG IS SET TO FALSE. THE FLAG MAY BE TESTED BY THE JVN
INSTRUCTION.


     REGISTERS     CALL            EXIT
     R0            LOP1            0.RESULT
     R1            MOP1            1.RESULT
     R2            LOP2            DEST
     R3            MOP2            DEST
     R4            -               DEST
     R5            REF.RESULT      REF.RESULT
     R6            -               DEST
     R7            LINK            DEST
```

4.52 XAMOS Bound procedures.

An XAMOS process which must write outside its own data memory (as f.ex. a driver) can get authorization to do so, either

- permanently, if it is created with LEVEL = 1 (system level)

  or
- temporarily, by calling the monitor procedure RELBOUND before and SETBOUND after each write to foreign memory. It is the responsibility of the process to save the original value of BOUND between the calls of RELBOUND and SETBOUND

4.52.1 Release Bound Protection.

MONITOR PROCEDURE RELEASE BOUND PROTECTION
INVOKATION:
```
   MON      RELBOUND
```
BOUND PROTECTION IS DISABLED, BY SETTING THE FIELD XBOUND IN THE PROCESS CONTEXT AND THE BOUND REGISTER IN THE CPU TO -1 (XAMOS ONLY).

REGISTER	CALL	EXIT
R4	-	OLD BOUND
R7	LINK	DEST

A

4.52.2 Set Bound Protection.

MONITOR PROCEDURE SET BOUND PROTECTION I:(BOUND VALUE)
INVOKATION:

```
    MON      SETBOUND
```

BOUND PROTECTION IS ENABLED, BY SETTING THE FIELD XBOUND IN THE PROCESS CONTEXT AND THE BOUND REGISTER IN THE CPU (XAMOS ONLY).

REGISTER	CALL	EXIT
R4	BOUND	OLD BOUND
R7	LINK	DEST

5. LIMITATIONS

The following limitations apply to the AMOS Kernel:

- Only a single CPU can execute with I/O interrupts enabled. This restriction arises from the CR80 interrupt handling hardware and firmware. The reason for the restriction is to prevent re-incarnations of processes and to be able to have control over the CPU executing a given process.

- The CPUs supported by the Kernel must all have access to the same main memory. Further must they have access to the first 4 Kword of main memory via the Mainbus (in order to be able to use hardware semaphores).

The following CR80 configurations are supported

- Up to 256 Kword of main memory
- Up to 8 CPUs (system generation parameter)
- CPUs with loadable control store.

K

6. SYSTEM ASSEMBLY PARAMETERS

In this section some assembly parameters are described which allow a tuning of the Kernel:

MULTIPAGE (Boolean)

Default value is true. If set to false, the Kernel will only support CR80 configurations with up to 64 Kwords of main memory and a minor gain in speed is obtained.

MSGCHK (Boolean)

Default value is true. If set to false, the Kernel will not check the number of message buffers allocated per process, and a small gain in speed is obtained.

MSGCHK1 (Boolean)

Default is false. If true a check is performed at process creation that the message buffer pool is never over_allocated.

NSEARCH (integer)

Default value is 10. Defines the maximum number of PCBs inspected a time by the Kernel during a search for a process. (Every time NSEARCH PCBs have been inspected a pause is made to allow other processes to enter the Kernel).

CPRIOS (integer)

Default value is 3. Defines the number of software priorities (= number of ready lists per CPU).

REGIONS (Boolean)

Defines whether critical regions are to be supported.

SECT1 (Boolean)

Default is true. If true the Kernel data are laid out in memory section 1 other wise in memory section Ø.

XAMOS (Boolean)

Default is true. If false, only AMOS CPUs are supported.

7. SYSTEM GENERATION

System generation consists of two phases:

- Assembling/compilation of modules
- Linking of modules to generate a boot module

The second phase is best performed by use of the CR80 AMOS UTILITY SYSGEN (ref. 2.6). The user manual for this program should be consulted for further details.

8. PERFORMANCE

This section is a summary of CR80 ececution times measured for selected AMOS kernel components.

Three different methods of measuring have been used:

(a) instruction count:
The number of instructions were multiplied with the average instruction time

2,2 us for CR8001
1,5 us for CR80101

(b) simulation
The simulation was performed by a Pascal program. The relevant prefix procedure was called a large number of times (e.g. 10000). The overhead caused by entering and leaving Pascal procedures was measured by calling a dummy procedure with identical parameter list but empty procedure body.
As a prefix procedure causes less overhead an average of 15 instructions was subtracted from the overhead measures.

(c) Using the time for a related operation.

N.B. Memory is always assumed to be accessed via the main bus and not via the sub bus.

Function	CR8001 execution time (us)		CR80101 execution time (us)	
WAIT EVENT:				
signal	195	b	163	b
delay	205	b	175	b
DIALOGUE: send message + wait message + send answer + wait answer	1280	b	890	b
Send signal	210	b	153	b
Wait answer: timeout	217	b	178	b
Save event + recover events	415	b	283	b
Path messages: Use the exrc. times for ordinary messages				
CRITICAL REGIONS:				
enter region	220	c	150	b
leave region	220	c	150	b
get item	270	c	183	b
put item	270	c	183	b
get N items	280+22. N	c	190+15˙N	b
put N items	280+22˙N	c	190+15˙N	b
copy N item	280+22˙N	c	190+15˙N	b
Read RTC	55	a	38	a

9. GUIDELINES FOR FUTURE IMPROVEMENTS

One obvious improvement would be to implement part of the Kernel code as microprogram.

The most often executed parts of the Kernel are the procedures called in connection with

- Entering the Kernel
- Exitting from the Kernel
- Scheduling
- Suspending a Process
- Readying a process

These subprograms are proper candidates for micro-programming.

APPENDIX A

S2SYSS

CR80 AMOS NAMES

The file S2SYSS is a text file written to be used as part of CR80 assembly program source files.

S2SYSS defines the values of the symbolic monitor call arguments to be used for calling AMOS monitor procedures. It also defines values of symbolic Kernel call parameters.

```
;-----------------------------------------------------------------------
;
;  PROJECT:          AMOS
;
;  MODULE NAME:      S2SYSS
;  MODULE ID NMB:    CSS/811
;  MODULE VERSION:   8
;  MODULE TYPE:      MERGE FILE
;  MODULE FILES:     S2SYSS.S
;  MERGE FILES:      NONE
;
;  SPECIFICATIONS:   CSS/302/PSP/0008
;  AUTHOR/DATE:      JHO
;
;  DELIVERABLE:      YES
;  SOURCE LANGUAGE:  CR80 ASSEMBLER
;  COMPILE COMPUTER: CR80
;  TARGET COMPUTER:  CR80
;  OPER. SYSTEM:     AMOS
;
;-----------------------------------------------------------------------
;
;  CHANGE RECORD:
;
;  VERSION    AUTHOR/DATE    DESCRIPTION OF CHANGE
;  -------    -----------    ---------------------
;
;    0501       JHO/801015     READSYSTIME AND PASCALINIT2
;                              INCLUDED
;
;    0601       JHO/801121     FILENAME INCLUDED
;
;    0701       AEK/800105     MONITORNAME DEVICE #86 CHANGED TO
;                              MONITORNAME TTYLOG #86 TO SUPPORT CSS/339
;
;    0801       HPT/820501     MONITORNAMES RELBOUND AND SETBOUND INCLUDED
;
;-----------------------------------------------------------------------

MESSAGE <:AMOS SYSTEM NAMES V820501:>
SYS2=                    TRUE
; EVENTTYPES
AX=0
BMSIG:=                  1<AX
BNSIG:=                 0 AX, AX=AX+1    ;    SIGNAL TYPE
BMMSG:=                  1<AX
BNMSG:=                 1 AX, AX=AX+1    ;    MESSAGE TYPE
BMANS:=                  1<AX
BNANS:=                 2 AX, AX=AX+1    ;    ANSWER TYPE
BMSYM:=                  1<AX
BNSYM:=                 3 AX, AX=AX+1    ;    SYSTEM MESSAGE TYPE
BMSYA:=                  1<AX
BNSYA:=                 4 AX, AX=AX+1    ;    SYSTEM ANSWER TYPE
BMPTM:=                  1<AX
BNPTM:=                 5 AX, AX=AX+1    ;    PATH MESSAGE TYPE
BMPTA:=                  1<AX
BNPTA:=                 6 AX, AX=AX+1    ;    PATH ANSWER TYPE
BMINTRPT:=               1<AX
BNINTRPT:=              7 AX, AX=AX+1    ;    INTERRUPT TYPE
BMDELAY:=                1<AX
BNDELAY:=               8 AX, AX=AX+1    ;    DELAY TYPE
BMPARSIG:=               1<AX
BNPARSIG:=              9 AX, AX=AX+1    ;    PARENT SIGNAL
CONTLENGTH:=             5               ;    ELNGTH OF MESSAGE BUFFER
; COMMAND BITS
; TRANSPUT OPERATIONS
AX=0
BNTPUT:=                 AX, AX=AX+1
```

```
BNOPUT:=                AX,  AX=AX+1
BNBYTE:=                AX,  AX=AX+1
BNSPEC:=                AX,  AX=AX+1
BNCONV:=                AX,  AX=AX+1
BNSTEP:=                AX,  AX=AX+1
BNNOEC:=                AX,  AX=AX+1
BNNOCP:=                AX,  AX=AX+1
; CONTROL OPERATIONS
AX=2
BNRELEASE:=             AX,  AX=AX+1
BNRESERVE:=             AX,  AX=AX+1
BNPOSITION:=            AX,  AX=AX+1
BNERASE:=               AX,  AX=AX+1
BNCLEAR:=               AX,  AX=AX+1
BNTERMINATE:=           AX,  AX=AX+1
BNDISCONNECT:=          AX,  AX=AX+1
; RESULT BITS
AX=     0
BNNOTREADY:=            AX,  AX=AX+1
BNTIMER:=               AX,  AX=AX+1
BNREJECT:=              AX,  AX=AX+1
BNILLEGAL:=             AX,  AX=AX+1
BNUNCOMPLETE:=          AX,  AX=AX+1
BNERROR:=               AX,  AX=AX+1
BNEOF:=                 AX,  AX=AX+1
BNPARITY:=              AX,  AX=AX+1
BNREADERROR:=           AX,  AX=AX+1
BNWRITEERROR:=          AX,  AX=AX+1
BNFULL:=                AX,  AX=AX+1
BNUNKNOWN:=             AX,  AX=AX+1
BNBUSY:=                AX,  AX=AX+1
BNNOTPOSS:=             AX,  AX=AX+1
;PAGE
```

```
;-----------------------------------------------------------------------
; SYSTEM CALLS
AX=0
CREATEPROCESS:=       AX, AX=AX+1

; PARAMETER BLOCK FOR CREATEPROCESS:
            AY=0
   XPRNAMELENGTH:=  4                ;   LENGTH OF PROCESS NAME
            VNAME0:=  AY,AY=AY+1     ;   NAME. IF VNAME0=0 THEN A STANDAR
                                     ;   D NAME IS GENERATED AND RETURNED
                                     ;   IT IS CHECKED THAT THE NAME DOES
                                     ;   NOT ALREADY EXIST NOR BEGINS WIT
                                     ;   TH "P".
            VNAME1:=  AY,AY=AY+1     ;
            VNAME2:=  AY,AY=AY+1     ;
            VIDENT:=  AY,AY=AY+1     ;   USED TO RETURN THE LOGICAL PCB

   VPROG:=            AY, AY=AY+1    ;   ABS PROGRAM BASE
   VINIT:=            AY, AY=AY+1    ;   PROGRAM RELATIVE START ADDRESS
   VMICRO:=           AY, AY=AY+1    ;   PROGRAM REL ADR TO MICRO PROGR
                                     ;   LOAD MODULE
                                     ;V8 PROGRAM PAGE
   VCAPAB:=           AY, AY=AY+1    ;   CAPABILITIES
   VCPU:=             AY, AY=AY+1    ;   LOGICAL CPU
   VPRIO:=            AY, AY=AY+1    ;   PRIORITY OF PROCESS TO BE CREATED
   VLEVEL:=           AY, AY=AY+1    ;   INITIAL SYSTEM LEVEL OF PROCESS
   VBASE:=            AY, AY=AY+1    ;   ABS BASE OF PROCESS TO BE CREATE
   VSIZE:=            AY,AY=AY+1     ;   SIZE OF PROCESS
   VBOUND:=           AY,AY=AY+1     ;   PRESET VALUE OF BOUND REGISTER.
   VMEMORY:=          AY,AY=AY+1     ;   MEMORY ALLOCATION PARAMETER.
   VMSGS:=            AY, AY=AY+1    ;   MAY NMB OF MSG BUFFERS ALLOWED
   XUSERIDLENGTH:=    2              ;   LENGTH OF USER ID
   VUSERID:=          AY, AY=AY+XUSERIDLENGTH;  USER ID
   VPARLGT:=          AY             ;   LENGTH OF PARAMETER BLOCK.

REMOVEPROCESS:=       AX, AX=AX+1
ADOPTPROCESS:=        AX, AX=AX+1
STARTPROCESS:=        AX, AX=AX+1
STOPPROCESS:=         AX, AX=AX+1
GETCHILD:=            AX, AX=AX+1
VANISH:=              AX, AX=AX+1
CLNMESSAGE:=          AX, AX=AX+1
CLNINTRPT:=           AX, AX=AX+1
ERROR:=               AX, AX=AX+1
TERMINATE:=           ERROR

;        ERROR CODE GROUPS
         USERER:=     0<8            ;     USER DEFINED ERRORS
         MONERR:=     1<8            ;     MONITOR KERNEL ERRORS (INCL HW)
         IOERR:=      2<8            ;     IO SYSTEM ERRORS
         FMSERR:=     3<8            ;   FILE MANAGEMENT SYSTEM ERROR
         FMUERR:=     4<8            ;   FILE MANAGEMENT SYSTEM ERROR
         FMDERR:=     5<8            ;   FILE MANAGEMENT SYSTEM ERROR
         DRVERR:=     6<8            ;     DEVICE DRIVER ERRORS
         PASERR:=     7<8            ;   PASCAL RUNTIME ERRORS
         OVLERR:=     8<8            ;   OVERLAY ERROR

LOOKUPCPU:=           AX, AX=AX+1
CLOSEPATH:=           AX, AX=AX+1
OPENPATH:=            AX, AX=AX+1
SETCYCLE:=            AX, AX=AX+1
CLEARINTERRUPT:=      AX, AX=AX+1
RELEASEINTERRUPT:=    AX, AX=AX+1
SETINTERRUPT:=        AX, AX=AX+1
RESERVEINTERRUPT:=    AX, AX=AX+1
IDENTIFYSENDER:=      AX, AX=AX+1
GETATTRIBUTES:=       AX, AX=AX+1
```

```
LOOKUPPROCESS:=         AX, AX=AX+1
SETCPUPARAMETER:=       AX, AX=AX+1
GETCPUPARAMETER:=       AX, AX=AX+1

BX=0                                    ;   CPU PARAMETERS
        ZCPUNMB:=       BX, BX=BX+1     ;     CPU NUMBER
        ZINTMSK:=       BX, BX=BX+1     ;     INTERRUPT MASK (PSW)
        ZSCHRCNT:=      BX, BX=BX+1     ;     SCHEDULE RESET COUNT .PRIO
        ZSLICESZ:=      BX, BX=BX+1     ;     SLICE SIZE .PRIO
        ZACCEXECT:=     BX, BX=BX+1     ;     ACC EXECUTION TIME .PRIO
        ZHWPRIO:=       BX, BX=BX+1     ;     HW PRIORITY BITS (PSW) .PRIO
        ZCPUMAXPAR:=    BX              ;

RECOVEREVENTS:=         AX, AX=AX+1
SAVEEVENT:=             AX, AX=AX+1
SUSPEND:=               AX, AX=AX+1
READY:=                 AX, AX=AX+1
        AX=AX+6                         ;   SPARE POSITIONS
        IF AX GT 63 THEN USE 16 FI
AX=64
CPUINIT:=               AX, AX=AX+1
MONINIT:=               AX, AX=AX+1
INITPASCAL:=            AX, AX=AX+1
OLTO:=                  AX, AX=AX+1
AWAITEVENT:=            AX, AX=AX+1
WAITEVENT:=             AWAITEVENT
SENDSIGNAL:=            AX, AX=AX+1
AWTANSWER:=             AX, AX=AX+1
SENDMESSAGE:=           AX, AX=AX+1
SENDANSWER:=            AX, AX=AX+1
AWTSYANSWER:=           AX, AX=AX+1
SENDSYMESSAGE:=         AX, AX=AX+1
SENDSYANSWER:=          AX, AX=AX+1
AWTPATHANSWER:=         AX, AX=AX+1
SENDPATHANSWER:=        AX, AX=AX+1
SENDPATHMESSAGE:=       AX, AX=AX+1
IDENTIFYPROCESS:=       AX, AX=AX+1
READRTC:=               AX, AX=AX+1
SENDTIMEOUT:=           AX, AX=AX+1
WRITERTC:=              AX, AX=AX+1
PROCESSPCBS:=           AX, AX=AX+1
READSYSTIME:=           AX, AX=AX+1
PASCALINIT2:=           AX, AX=AX+1
TTYLOG:=                AX, AX=AX+1
CLNDEVICE:=             AX, AX=AX+1
IO:=                    AX, AX=AX+1
CLNIO:=                 AX, AX=AX+1
IOINIT:=                AX, AX=AX+1
GETBUF:=                AX, AX=AX+1
ADRBUF:=                AX, AX=AX+1
RELBUF:=                AX, AX=AX+1
CLNMEM:=                AX, AX=AX+1
STREAM:=                AX, AX=AX+1
INSPECTEVENTS:=         AX, AX=AX+1
REGION:=                AX, AX=AX+1

BX=0                                    ;   REGION PROCEDURES
        RENTER:=        BX, BX=BX+1
        RLEAVE:=        BX, BX=BX+1
        RWAIT:=         BX, BX=BX+1
        RGET:=          BX, BX=BX+1
        RGETN:=         BX, BX=BX+1
        RPUT:=          BX, BX=BX+1
        RPUTN:=         BX, BX=BX+1
        RCREATE:=       BX, BX=BX+1
   ; PARAMETER BLOCK FOR CREATE REGION

              AY=0
              VCRNAME:=  AY, AY=AY+3;     NAME OF REGION
```

```
            VCRSTA:=   AY, AY=AY+1;     PSW ENCODED PAGE OF VS
            VCRADDR:=  AY, AY=AY+1;     ABSOLUTE WORD ADDRESS OF VS
            VCRSIZE:=  AY, AY=AY+1;     SIZE IN WDS OF VS
            VCRCBL:=   AY         ;     SIZE OF PARAMETER BLOCK

        RSEARCH:=    BX, BX=BX+1
        RCOPYN:=     BX, BX=BX+1

;       ERROR CODES FOR REGION PROCEDURES

        ZCRUNF:=     1   ;      UNKNOWN FUNCTION
        ZCRPSZ:=     2   ;      PARAMETER REF VIOLATES ADDRESS
                         ;      SPACE OF PROCESS.
        ZCRUNR:=     3   ;      UNKNOWN REGION
        ZCRILLSTA:=  4   ;      REGION IS NOT ENTERED
        ZCRPCB:=     5   ;      INVALID PROCESS (PCB INDEX)
        ZCRILLNAM:=  6   ;      INVALID REGION NAME
        ZCRVSZ:=     7   ;      ADDRESS VIOLATION IN VS
        ZCROVFL:=    8   ;      TOO MANY REGIONS

OVERLAY:=            AX, AX=AX+1
LOG:=                AX, AX=AX+1
MULTIPLYLONG:=       AX, AX=AX+1
DIVIDELONG:=         AX, AX=AX+1
FINDFILE:=           AX, AX=AX+1
INFILEID:=           AX, AX=AX+1
LOGP:=               AX, AX=AX+1
COR:=                AX, AX=AX+1
                         AX=AX+1     ;     PREVIOUS ENTRY FOR FILENAME
SETBOUND:=           AX, AX=AX+1     ;V8
RELBOUND:=           AX, AX=AX+1     ;V8
FILENAME:=           254             ;     CHANGED FROM 106 FOR COBOL USE
```

APPENDIX B

X2GEN1

CR80 AMOS PROGRAM AND DATA HEADER GENERATOR PART 1

The text file X2GEN1 is written to be used as part of CR80 assembly program source files.

X2GEN1 together with X2GEN2 (appendix C) generates program and/or data headers in the format used by ROOT and the CR80 AMOS I/O system.

X2GEN1 should be included in the start of CR80 assembly source files before any data or instruction words have been assembled. Improper use will generate a message:

X2GEN1 MUST BE CALLED INITIALLY IN SOURCE.

To control the header generation, a number of parameters must be defined. Some of these parameters are defaulted. The default values may be overridden by user assignments.

The parameters which the user may and/or must define are listed below together with their possible default values.

Parameters which must be defined

XPROGRAM

Type : Boolean
Effect: If true a program header is generated.
Note : Must be defined prior to call of X2GEN1.

XDATA

Type : Boolean
Effect: If true a data header is generated.
Note : Must be defined prior to call of X2GEN1.

XPGNAME0
XPGNAME1
XPGNAME2

Type : String (2 characters each)
Effect: Defines the name (6 characters) of the program.
May be assigned at any position in source.
Note : Need not be defined if XPROGRAM is false.
Convention:
XPGNAME0,1,2 is assigned the configuration identification of the assembled module.
(Example CSS302 for the AMOS Kernel).

XVERSION

Type : Integer

Effect: Defines the program release version by convention. May be assigned at any position in source.

Note : Need not be defined if XPROGRAM is false.

XSTART

Type : Program relative reference.

Effect: Defines the entry point in the assembled program.
Must be assigned prior to call of X2GEN2.

Optionally used parameters

XPGTYPE

Type : Integer
Effect: Defines the type of the program. The following bitmasks for XPGTYPE are defined:
BMREENTRANT defines the program part to be reentrant.
BMRESIDENT defines the program part to be not swappable.
BMPERMANENT defines the program part to be not removeable.
BMMONITOR defines the program part as a monitor procedure. These are initialized specially by ROOT.
BMUTILITY defines the program to be a CR80 AMOS utility program. This has a special implication if the program is also a pascal program.
BMPASCAL defines the source language to be Pascal.

Note : May be defined before call of X2GEN2.
Default: 0, set by X2GEN2.

XMICRO

Type : Program relative reference.
Effect: Defines the first location in the program part of a binary micro program load module.
Note : May be defined prior to call of X2GEN2.
Default: 0, (no micro module)
, set by X2GEN2

XPGMEM

Type:	Integer
Effect:	Defines the memory area in which the program must be placed.
Note:	May be defined prior to call of X2GEN2. The format of this parameter is defined in 3.16
Default:	## FF00 set by X2GEN2.

XPRLEVEL

Type:	Integer
Effect:	Defines the initial value of system call nesting. Should be 0 for application programs. If 1, XLEVEL is initiated to -1 which allows the process to write everywhere.
Note:	May be defined prior to call of X2GEN2.
Default:	0, set by X2GEN2.

XCAPABILITIES

Type:	Integer
Effect:	Defines the necessary process capabilities.
Note:	May be defined prior to call of X2GEN2.
Default:	0, set by X2GEN2.

XCPUNAME0
XCPUNAME1
XCPUNAME2

Type:	String (2 characters each)
Effect:	Used by ROOT to define the CPU which must execute the program.
Note:	May be defined prior to call of X2GEN2.
Default:	0, set by X2GEN2.

<u>XPROCESSNAME0</u>
<u>XPROCESSNAME1</u>
<u>XPROCESSNAME2</u>

Type:	String (2 characters each)
Effect:	Used by ROOT to define the process name.
Note:	May be defined prior to call of X2GEN2. (A name commencing with P (e.g. PROGXY) is illegal)
Default:	0, 0, 0, set by X2GEN2.

XPRIORITY

Type : Integer.
Effect: Used by ROOT to define the software priority.
Note : May be defined prior to call of X2GEN2.
Default: 1, set by X2GEN2.

XTRA

Type : Integer.
Effect: Defines the size of the not assembled data area between BOUND and IOAREA (refer to fig. B.1)
Note : May be defined after call of X2GEN1.
Default: 0, set by X2GEN1.

XTND

Type : Integer.
Effect: Defines the size of the not assembled data area below BOUND (refer to fig. B.3)
Note : May be defined after call of X2GEN1.
Default: 0, set by X2GEN1.

XMSGS

Type : Integer.
Effect: Defines the maximum number of message buffers allocatable by the process.
Note : May be defined after call of X2GEN1.
Default: 4, set by X2GEN1.

XFDS

Type : Integer
Effect: Defines the number of file descriptions to be laid out.
Note : May be defined after call of X2GEN1.
Default: 0, set by X2GEN1.

XIBS

Type : Integer
Effect: Defines the number of I/O control blocks to be laid out.
Note : May be defined after call of X2GEN1.
Default: 0, set by X2GEN1.

XSTS

Type : Integer
Effect: Defines the number of stream control blocks to be laid out.
Note : May be defined after call of X2GEN1.
Default: 0, set by X2GEN1.

XXFS

Type : Integer
Effect: Defines the number of transfer list elements to be laid out.
Note : May be defined after call of X2GEN1.
Default: 0, set by X2GEN1.

XPRMEM

Type: Integer

Effect: Defines the memory area, in which the process must be placed.

Note: May be defined prior to call of X2GEN2.
The format of this parameter is defined in 3.16.

Default: # FF00 set by X2GEN2.

XUSERID0
XUSERID1

Type: Integer

Effect: Defines the user id for the process.

Note: May be defined prior to call of X2GEN2.

Default: 0,0, set by X2GEN2.

The format of the headers generated by X2GEN1 is shown in figures B.1 and B.2.

The format of the object module for CR80 AMOS programs/data is shown in fig. B.3.

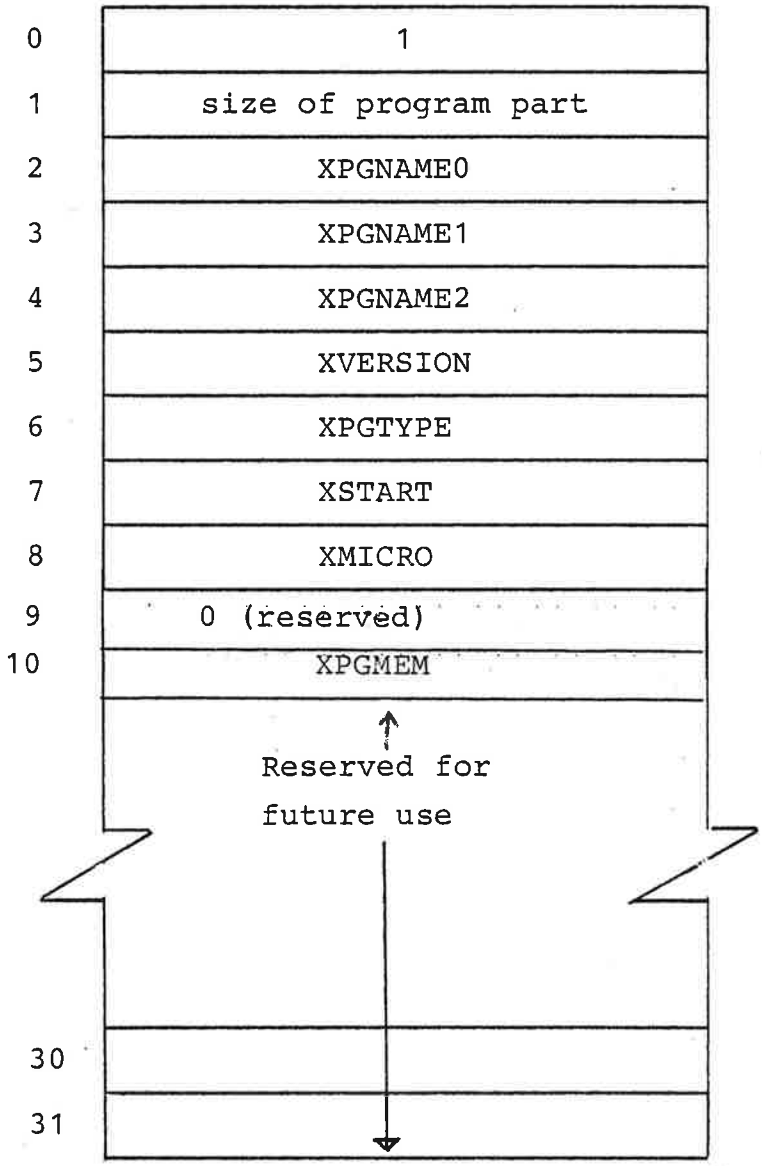


Fig. B.1 CR80 AMOS Program Header.

0	2
1	size of assembled data part
2	XPROCESSNAME0
3	XPROCESSNAME1
4	XPROCESSNAME2
5	XCPUNAME0
6	XCPUNAME1
7	XCPUNAME2
8	XPRIORITY
9	XCAPABILITIES
10	memory claim
11	size of executing process
12	XFDS
13	XIBS
14	XSTS
15	XXFS
16	XMSGS
17	0 (reserved)
18	XPRMEM
19	ref to I/O part
20	XUSERID0
21	XUSERID1

Fig. B.2-1 CR80 AMOS Data Header, part 1/2

22	0 (reserved)
23	0 (reserved)
24	XPRLEVEL
25	BOUND
26	Register area (←BASE)
38	100 (TIMER)
39	#6800 (PSW)
40	0
	Reserved
60	0

Fig. B.2-2 CR80 AMOS Data Header,
part 2/2

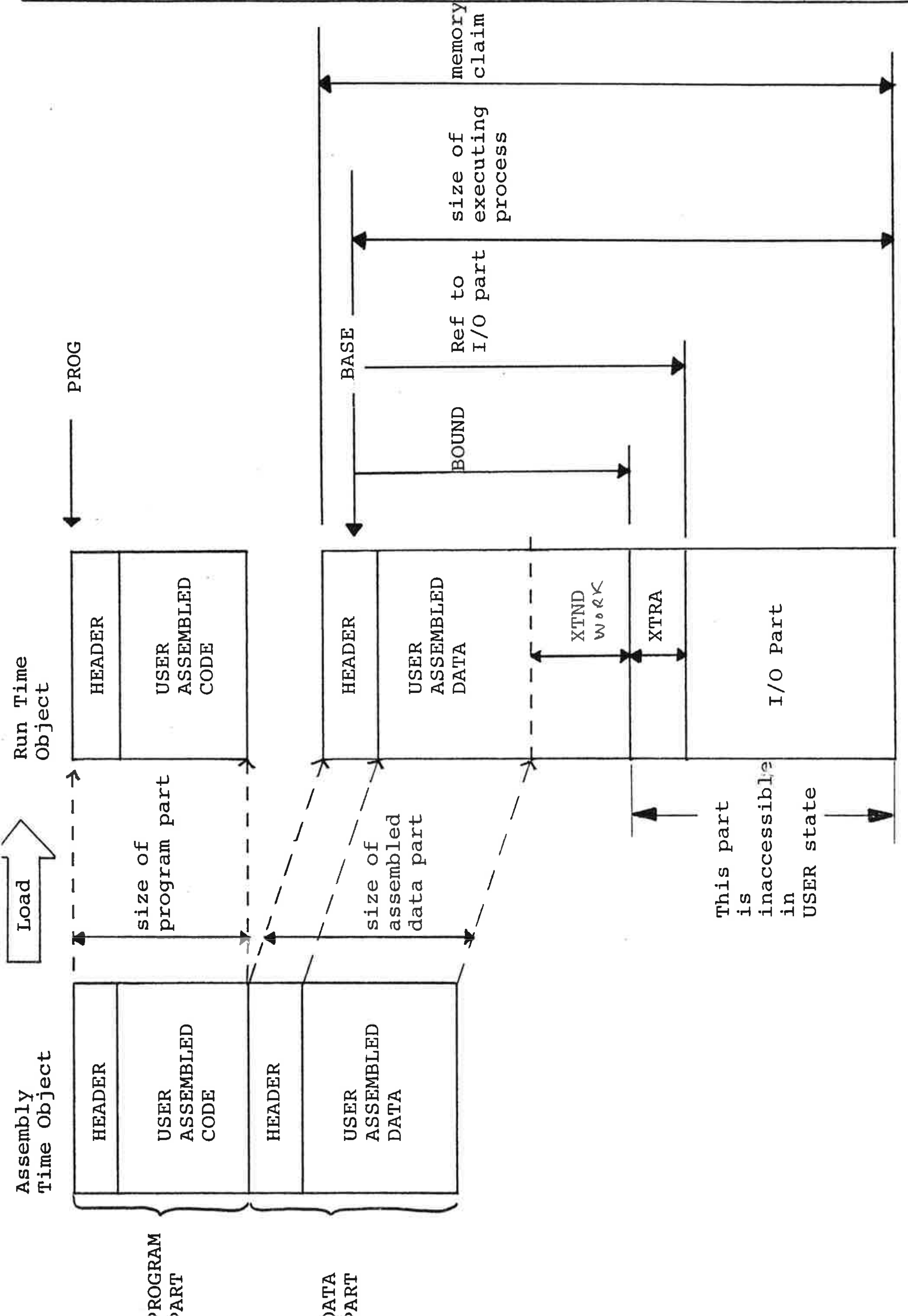


Fig. B.3 CR80 AMOS Object Code Lay-out at Assembly Time and at Run Time.

```
;*****************************************************************
;*                                                               *
;*          C R   8 0   A M O S                                  *
;*          D A T A   A N D   P R O G R A M   H E A D E R        *
;*          G E N E R A T O R   P A R T   1                      *
;*          CONFIG ID: CSS/831                                   *
;*          AUTHOR:    JHO                                       *
;*          DATE:      820501                                    *
;*          VERSION:   2                                         *
;*                                                               *
;*****************************************************************
;
;  CHANGE RECORD:
;
;  VERSION    AUTHOR/DATE    DESCRIPTION OF CHANGE
;  -------    -----------    ---------------------
;
;    0101      JHO/790827     INITIAL RELEASE
;
;    0201      HPT/820501     XAMOS DEFINITIONS INCLUDED
;
;-----------------------------------------------------------------
MESSAGE <:X2GEN1 V820501:>
USE PROG
IF WORDS NE 0 THEN MESSAGE <:X2GEN1 MUST BE CALLED INITIALLY IN SOURCE:>
FI
AREASWITCH=   1
XTRA=         0                     ;   SIZE OF NOT ASSEMBLED LOCAL DATA
                                    ;   ABOVE BOUND
XTND=         0                     ;   SIZE OF NOT ASSEMBLED DATA BELOW
                                    ;   BOUND
XFDS=         0                     ;   DEFAULT NMB OF FILE DESRIPTIONS
XIBS=         0                     ;   DEFAULT NMB OF IO CONTROL BLOCKS
XXFS=         0                     ;   DEFAULT NMB OF XFER LIST ELEMENT
XSTS=         0                     ;   DEFAULT NMB OF IO STREAMS
XIOSIZE=      0                     ;   DEFAULT SIZE OF IO AREA
XMSGS=        4                     ;   DEFAULT NMB OF MESSAGE BUFFERS
; GENERAL HEADER DECLARATION
AX=0
XHTYPE:=              AX, AX=AX+1   ;   HEADER TYPE
BX=0
XTABLE:=              BX, BX=BX+1   ;   TABLE HEADER
XCODE:=               BX, BX=BX+1   ;   PROGRAM HEADER
XPROCESS:=            BX, BX=BX+1   ;   PROCESS HEADER
XHSIZE:=              AX, AX=AX+1   ;   SIZE OF ITEM (IN WORDS)
XHNAME:=              AX, AX=AX+3   ;   NAME OF ITEM
XHGHL:=               AX            ;   LENGTH OF GENERAL HEADER
; PROGRAM HEADER DECLARATION
          AX=   XHGHL               ;   GENERAL HEADER HEADER
XPVERS:=              AX, AX=AX+1   ;   PROGRAM VERSION
XPTYPE:=              AX, AX=AX+1   ;   TYPE
BX=0
BNREENTRANT:=         BX, BX=BX+1   ;   REENTRANT VS NON REENTRANT
BNRESIDENT:=          BX, BX=BX+1   ;   RESIDENT VS SWAPPABLE
BNPERMANENT:=         BX, BX=BX+1   ;   PERMANENT VS REMOVEABLE
BNMONITOR:=           BX, BX=BX+1   ;   MONITOR CODE VS NON MONITOR CODE
BNUTILITY:=           BX, BX=BX+1   ;   UTILITY PROGRAM VS NOT UTILITY
BNPASCAL:=            BX, BX=BX+1   ;   PASCAL PROGRAM VS NOT PASCAL P
BMREENTRANT:=         1<BNREENTRANT
BMRESIDENT:=          1<BNRESIDENT
BMPERMANENT:=         1<BNPERMANENT
BMMONITOR:=           1<BNMONITOR
BMUTILITY:=           1<BNUTILITY
BMPASCAL:=            1<BNPASCAL
XPSTART:=             AX, AX=AX+1   ;   RELATIVESTART ADDRESS
XPMICRO:=             AX, AX=AX+1   ;   REL REF TO MICRO LOAD MODULE
XPCHKS:=              AX, AX=AX+1   ;   CHECKSUM
XPMEM:=               AX, AX=AX+1   ;   MEMORY PARAMETER
```

```
        IF AX GT 32 THEN USE 16 FI
                         AX=32          ;   SPARE POSITIONS
XPGHDL:=                 AX             ;   LENGTH OF PROGRAM HEADER
IF XPROGRAM THEN
        XCODE                           ;   PROGRAM TYPE HEADER
        XPGWDS                          ;   SIZE OF PROGRAM
        XPGNAME0                        ;   PROGRAM NAME
        XPGNAME1
        XPGNAME2
        XVERSION                        ;   PROGRAM VERSION
        XPGTYPE                         ;   TYPE OF PROGRAM
        XSTART                          ;   RELATIVE START ADDRESS
        XMICRO                          ;   RELATIVE ADDRESS TO MICRO
                                        ;   PROGRAM LOAD MODULE
        0                               ;   CHECKSUM
        XPGMEM                          ;V2 PROGRAM MEMORY RANGE
        0, REPEAT XPGHDL-LOC
FI
USE BASE
; PROCESS HEADER DECLARATION
        AX=   XHGHL                     ;   GENERAL HEADER HEADER
XPROCHL:=                AX             ;   LENGTH OF PROCESS HEADER
;-----------------------------------------------------------------------
; GENERAL PROCESS DESCRIPTION
;-----------------------------------------------------------------------
AX=     -19-XUSERIDLENGTH
XBEYLGT:=               -AX             ;   SIZE OF AREA BEYOND REGISTERS
XPCPUNAME:=             AX, AX=AX+3     ;   CPU NAME
XPRIO:=                 AX, AX=AX+1     ;   PRIORITY
XPCAP:=                 AX, AX=AX+1     ;   CAPABILITY REQUIREMENT
        BX=0                            ;   ACCESS:
        BNCLASS:=        12             ;     LOW ORDER BIT OF CLASS FIELD
        BNMAXCL:=        15             ;     MAXIMUM CLASSIFICATION CODE
        BNCREPR:=        BX,BX=BX+1     ;     CREATE PROCESS
        BNCCRPR:=        BX,BX=BX+1     ;     CREATE PROCESS WHICH CREATES
                                        ;     A PROCESS
        BNCREPG:=        BX,BX=BX+1     ;     CREATE AND LOAD PROGRAM
        BNCCRPG:=        BX,BX=BX+1     ;     CREATE PROCESS WHICH CREATES
                                        ;     AND LOADS PROGRAMS
        BNALDEV:=        BX,BX=BX+1     ;     ALLOCATE DEVICE
        BNALMEM:=        BX,BX=BX+1     ;     ALLOCATE MEMORY
        IF BX GT 12 THEN USE 16 FI      ;   UNDERLINE IF ERROR
XPRCLAIM:=              AX, AX=AX+1     ;   MEMORY CLAIM FOR PROCESS (WORDS)
XPRSIZE:=               AX, AX=AX+1     ;   SIZE OF EXECUTING PROCESS
XPFDSX:=                AX, AX=AX+1     ;   NUMBER OF FILE DESCRIPTIONS
XPIBSX:=                AX, AX=AX+1     ;   NUMBER OF IO CONTROL BLOCKS
XPSTSX:=                AX, AX=AX+1     ;   NUMBER OF STREAMS
XPXFSX:=                AX, AX=AX+1     ;   NUMBER OF TRANSFER LIST ELEMENTS
XPMSGX:=                AX, AX=AX+1     ;   NUMBER OF MSG BUFFERS
XCURDIR:=               AX, AX=AX+1     ;   CURRENT DIRECTORY
XFUNCS:=                AX, AX=AX+1     ;   INITIALIZATION FUNCTIONS CALLED
XIODATA:=               AX, AX=AX+1     ;
XUSERID:=               AX, AX=AX+XUSERIDLENGTH
XCBASE:=                AX, AX=AX+1     ;   BASE COPY
XPCB:=                  AX, AX=AX+1     ;   LOGICAL PCB REF
XLEVEL:=                AX, AX=AX+1     ;   SYSTEM LEVEL
XSYSTEM:=               1               ;   SYSTEM LEVEL
XUSER:=                 0               ;   USER LEVEL
XBOUND:=                AX, AX=AX+1     ;   REGISTER
XR0:=                   AX, AX=AX+1     ;   REGISTER
IF XR0 NE 0 THEN MESSAGE <:HEADER ERROR:> FI
XR1:=                   AX, AX=AX+1     ;   REGISTER
XR2:=                   AX, AX=AX+1     ;   REGISTER
XR3:=                   AX, AX=AX+1     ;   REGISTER
XR4:=                   AX, AX=AX+1     ;   REGISTER
XR5:=                   AX, AX=AX+1     ;   REGISTER
XR6:=                   AX, AX=AX+1     ;   REGISTER
XR7:=                   AX, AX=AX+1     ;   REGISTER
XBASE:=                 AX, AX=AX+1     ;   REGISTER
```

```
XMOD:=              AX, AX=AX+1     ;   REGISTER
XPROG:=             AX, AX=AX+1     ;   REGISTER
XPRPC:=             AX, AX=AX+1     ;   REGISTER
XTIMER:=            AX, AX=AX+1     ;   REGISTER
XPSW:=              AX, AX=AX+1     ;   REGISTER
XOLDPRC:=           AX, AX=AX+1     ;   PREVIOUS PROCESS
XLOCACT:=           AX, AX=AX+1     ;   LOCAL ACTION
XLOCRET:=           AX, AX=AX+1     ;   LOCAL ACTION RETURN LINK
XCAUSE:=            AX, AX=AX+1     ;   LOCAL INTERRUPT CAUSE CODE
XDEVICE:=           AX, AX=AX+1     ;   DEVICE ADDRESS
XTIMRS:=            AX, AX=AX+1     ;   TIMER RESET VALUE
XMONRET:=           AX, AX=AX+1     ;   MONITOR RETURN LINK
XTLINK:=            AX, AX=AX+1     ;   TIMER LINK
XLINK0:=            AX, AX=AX+1
XLINK1:=            AX, AX=AX+1
XLINK2:=            AX, AX=AX+1
XLINK3:=            AX, AX=AX+1     ;
XLINK4:=            AX, AX=AX+1     ;
XLINK5:=            AX, AX=AX+1     ;
XLINK6:=            AX, AX=AX+1     ;
XLINK7:=            AX, AX=AX+1     ;
XWORKLGT=           5               ;   SIZE OF WORK AREA
XWORK:=             AX, AX=AX+XWORKLGT;  WORK AREA
XPROCLGT:=          AX              ;   LENGTH OF PROCESS DESCIPTOR ABOV
                                    ;   REGISTERS
XFIRST= -(XBEYLGT+XPROCHL)
IF XDATA THEN
LOC=    XFIRST
        XPROCESS                    ;   PROCESS TYPE HEADER
        XPRWDS                      ;   LENGTH OF PROCESS FILE
        XPROCESSNAME0               ;   NAME OF PROCESS
        XPROCESSNAME1
        XPROCESSNAME2
        XCPUNAME0                   ;   NAME OF REQUIRED CPU
        XCPUNAME1
        XCPUNAME2
        XPRIORITY                   ;   REQUIRED PRIORITY FOR PROCESS
        XCAPABILITIES               ;   REQUIRED CLASSIFICATION LEVEL
                                    ;   AND CAPABILITIES OF PROCESS
        XTOTSZ                      ;   MEMORY CLAIM
        XPRLNG                      ;   SIZE OF EXECUTING PROCESS
        XPFDS                       ;   NUMBER OF FILE DESCRIPTOINS
        XPIBS                       ;   NUMBER OF IO CONTROL BLOCKS
        XPSTS                       ;   NUMBER OF IO STAREAMS
        XPXFS                       ;   NUMBER OF TRANSFER LIST ELEMENTS
        XPMSGS                      ;   NUMBER OF MESSAGE BUFFERS
        0                           ;   CURRENT DIRECTORY
        XPRMEM                      ;V2 PROCESS MEMORY RANGE
        XIOREF                      ;   REF TO IO DATA
        XUSERID0, XUSERID1          ;   USER ID
        IF XUSERIDLENGTH NE 2 THEN MESSAGE <:USERIDLENGTH ERROR:> FI
        0                           ;   BASE COPY
        0                           ;   XPCB
        XPRLEVEL                    ;   REQUIRED EXECUTION LEVEL OF PROC
        XBNDSZ                      ;   BOUND
        0, REPEAT 7                 ;   REGISTERS 0-7
        XABASE, XABASE, XAPROG, XAPRPC, 100, #6800
        0,     REPEAT (XPROCLGT-LOC)
FI
;
```

APPENDIX C

X2GEN2

CR80 AMOS PROGRAM AND DATA HEADER GENERATOR PART 2

The text file X2GEN2 is written to be used as part of the CR80 assembly language program source files.

X2GEN2 together with X2GEN1 (appendix B) generates program and data headers in the format used by ROOT and the CR80 AMOS I/O system.

```
;*********************************************************************
;*                                                                   *
;*            C R   8 0   A M O S                                    *
;*            D A T A   A N D   P R O G R A M   H E A D E R          *
;*            G E N E R A T O R   P A R T   2                        *
;*            CONFIG ID: CSS/833                                     *
;*            AUTHOR:    JHO                                         *
;*            DATE:      820501                                      *
;*            VERSION:   2                                           *
;*                                                                   *
;*********************************************************************
;
;  CHANGE RECORD:
;
;  VERSION    AUTHOR/DATE    DESCRIPTION OF CHANGE
;  -------    -----------    ---------------------
;
;    0101       JHO/790827     INITIAL RELEASE
;
;    0201       HPT/820501     XAMOS DEFINITIONS INCLUDED
;
;---------------------------------------------------------------------
MESSAGE <:X2GEN2 V820501:>
USE PROG
XASSEMBLED=   LOC
XPGTYPE=              0              ;    DEFAULT PROGRAMTYPE
XPGWDS:=              LOC            ;    PROGRAM AREA LENGTH
XMICRO=               0              ;    DEFAULT MICRO LOAD MODULE
XPGMEM=       #FF00                  ;V2 DEFAULT PROGRAM MEMORY RANGE
XPRMEM=       #FF00                  ;V2 DEFAULT DATA    MEMORY RANGE
USE BASE
XCPUNAME0=            0              ;    DEFAULT CPU NAME
XCPUNAME1=            0              ;    DEFAULT CPU NAME
XCPUNAME2=            0              ;    DEFAULT CPU NAME
XPRIORITY=            1              ;    DEFAULT PRIORITY
XPRLEVEL=             XUSER          ;    DEFAULT EXECUTION LEVEL
XCAPABILITIES=        0              ;    DEFAULT CAPABILITIES
XPROCESSNAME0=        0              ;    DEFAULT PROCESS NAME
XPROCESSNAME1=        0
XPROCESSNAME2=        0
XUSERID0=             0              ;    DEFAULT USERID
XUSERID1=             0              ;    DEFAULT USERID
XABASE=       0
XAPROG=       0
XAPRPC=       0+XSTART
XADJUST:=             0              ;    SIZE OF ADJUST AREA
XBNDSZ:=              LOC-1+XTND
XIOREF:=              XBNDSZ+1+XTRA
XPRLNG:=              XIOREF+XIOSIZE
IF XDATA THEN
XTOTSZ:=              XPRLNG+XADJUST-XFIRST
ELSE
XTOTSZ:=              0
FI
XPRWDS:=              LOC-XFIRST
XPSTS:=               XSTS
XPIBS:=               XIBS
XPFDS:=               XFDS
XPXFS:=               XXFS
XPMSGS:=              XMSGS
IF XDATA THEN
XASSEMBLED=   XASSEMBLED+LOC-XFIRST
ELSE
XASSEMBLED=   XASSEMBLED+LOC
FI
IF XASSEMBLED NE WORDS THEN
       MESSAGE <:LOCATION COUNTER CORRUPTED:> FI
;
```

APPENDIX D

PROGRAM EXAMPLE

```
LIST
BEGIN MODULE 0 USE BASE

;        CR80 AMOS SAMPLE PROGRAM

XPROGRAM=      TRUE                          ;   THERE IS A PROGRAM PART
XDATA=         TRUE                          ;   THERE IS A DATA PART
NOLIST                                       ;   SUPPRESS LISTING OF SYSTEM FILES
$S2SYSS                                      ;   INCLUDE S2SYSS
$X2GEN1                                      ;   INCLUDE X2GEN1
LIST                                         ;   ENABLE LISTING AGAIN
XPGNAME0=                <:SA:>              ;   WE REQUIRE THE PROGRAM TO
XPGNAME1=                <:MP:>              ;   BE CALLED 'SAMPLE'
XPGNAME2=                <:LE:>              ;
XVERSION=                7                   ;   IT IS VERSION 7 OF THIS PROGRAM
XPGTYPE=       BMREENTRANT                   ;   AND WE THINK IT REENTRANT
XPROCESSNAME0=           <:Q1:>              ;   WE REQUIRE THE PROCCESS TO BE
                                             ;   CALLED 'Q1'
XMSGS=         1                             ;   WE ONLY NEED 1 MESSAGE BUFFER
;        THE FOLLOWING LOCAL DATA ARE DEFINED
BUF:     1, 2, 3, 4, 5                       ;
RCVR:    <:COUNTR:>,0                        ;   A PROCESS NAME

USE PROG

INIT:
         MOVC    10                  R0      ;
         MON     SETCYCLE                    ;   SETCYCLE(1 SEC)
L0:
         MOVC    BMDELAY             R2      ;
         MON     WAITEVENT                   ;   EACH SECOND DO BEGIN
         MOVC    BUF                 R1      ;      REF(MESSAGE)
         MOVC    RCVR                R0      ;      REF(RECEIVER PROCESS)
         MON     SENDMESSAGE                 ;      SENDMESSAGE(MESSAGE,RECEIVER)
         MOVC    BMANS               R2      ;
         MON     WAITEVENT                   ;      AWAIT ANSWER
         JMP                 L0              ;   END

XSTART=  INIT                                ;   DEFINE ENTRY POINT
NOLIST                                       ;   SUPPRESS LIST OF X2GEN2
$X2GEN2                                      ;   INCLUDE X2GEN2
LIST
END
```

Source Program List

```
AU000001 0 0000 LIST                                                                  ;
AU000002 0 0000 BEGIN MODULE 0 USE BASE                                               ;
AU000003 0 0000                                                                       ;
AU000004 0 0000 ;          CR80 AMOS SAMPLE PROGRAM                                   ;
AU000005 0 0000                                                                       ;
AU000006 0 0000 XPROGRAM=      TRUE                     ;    THERE IS A PROGRAM PART  ;
AU000007 0 0000 XDATA=         TRUE                     ;    THERE IS A DATA PART     ;
         0 0000 NOLIST                                  ;    SUPPRESS LISTING OF SYSTEM FILES;
AU000008 0 0000  MESSAGE: AMOS SYSTEM NAMES V790827
AU000181 0 0000  MESSAGE: X2GEN1 V790827
AU000343 0 0023 LIST                                    ;    ENABLE LISTING AGAIN     ;
AU000344 0 0023 XPGNAME0=             <:SA:>            ;    WE REQUIRE THE PROGRAM TO ;
AU000345 0 0023 XPGNAME1=             <:MP:>            ;    BE CALLED 'SAMPLE'       ;
AU000346 0 0023 XPGNAME2=             <:LE:>            ;                             ;
AU000347 0 0023 XVERSION=             7                 ;    IT IS VERSION 7 OF THIS PROGRAM ;
AU000348 0 0023 XPGTYPE=       BMREENTRANT              ;    AND WE THINK IT REENTRANT ;
AU000349 0 0023 XPROCESSNAME0=        <:Q1:>            ;    WE REQUIRE THE PROCCESS TO BE ;
AU000350 0 0023                                         ;    CALLED 'Q1'              ;
AU000351 0 0023 XMSGS=         1                        ;    WE ONLY NEED 1 MESSAGE BUFFER ;
AU000352 0 0023 ;          THE FOLLOWING LOCAL DATA ARE DEFINED                       ;
AU000353 0 0023 BUF:       1, 2, 3, 4, 5                ;                             ;
AU000354 0 0028 RCVR:      <:COUNTR:>,0                 ;    A PROCESS NAME           ;
AU000355 0 002C                                                                       ;
AU000356 0 002C USE PROG                                                              ;
AU000357 1 0020                                                                       ;
AU000358 1 0020 INIT:                                                                 ;
AU000359 1 0020            MOVC  10                 R0  ;                             ;
AU000360 1 0021            MON   SETCYCLE               ;    SETCYCLE(1 SEC)          ;
AU000361 1 0022 L0:                                                                   ;
AU000362 1 0022            MOVC  BMDELAY            R2  ;                             ;
AU000363 1 0024            MON   WAITEVENT              ;    EACH SECOND DO BEGIN     ;
AU000364 1 0025            MOVC  BUF                R1  ;       REF(MESSAGE)          ;
AU000365 1 0026            MOVC  RCVR               R0  ;       REF(RECEIVER PROCESS) ;
         1 0027            MON   SENDMESSAGE            ;       SENDMESSAGE(MESSAGE,RECEIVER);
AU000366 1 0028            MOVC  BMANS              R2  ;                             ;
AU000367 1 0029            MON   WAITEVENT              ;       AWAIT ANSWER          ;
AU000368 1 002A            JMP              L0          ;    END                      ;
AU000369 1 002B                                                                       ;
AU000370 1 002B XSTART= INIT                            ;    DEFINE ENTRY POINT       ;
AU000371 1 002B NOLIST                                  ;    SUPPRESS LIST OF X2GEN2  ;
AU000372 1 002B  MESSAGE: X2GEN2 V790827
AU000416 0 002C LIST                                                                  ;
AU000417 0 002C END                                                                   ;
Z00000 T0071
P
 0000L 0001 002B 4153 504D 454C 0007 0001 0020
 0008L 0000 0000 0000 0000 0000 0000 0000 0000
 0010L 0000 0000 0000 0000 0000 0000 0000 0000
 0018L 0000 0000 0000 0000 0000 0000 0000 0000
 0020L 0A48 0DA6 0156 004A 44A6 2349 2848 47A6
 0028L 044A 44A6 0958 0002 0046 3151 0000 0000
 0030L 0000 0000 0000 0001 0000 0046 002C 0000
 0038L 0000 0000 0000 0001 0000 0000 002C 0000
 0040L 0000 0000 0000 0000 002B 0000 0000 0000
 0048L 0000 0000 0000 0000 0000 0000 0000 0000
 0050L 0020 0064 6800 0000 0000 0000 0000 0000
 0058L 0000 0000 0000 0000 0000 0000 0000 0000
 0060L 0000 0000 0000 0000 0000 0000 0000 0000
 0068L 0001 0002 0003 0004 0005 4F43 4E55 5254
       0000
       S6
```

Assembly Verification List

```
 MEMORY MAP:
   AREA 1  0000
   AREA 0  002B

      1 DIMENSION WARNINGS
      1 MODIFIES INSERTED
    113 WORDS OUTPUT
    113 WORDS ASSEMBLED
 ASSEMBLY OK?
```

APPENDIX E

Emulation of
XAMOS instructions on
AMOS CPU.

When a new XAMOS instruction is executed on an AMOS CPU, it is recognized as an illegal instruction. This applies to AMOS CPUs without loadable micro program storage and without function submodule, and a local interrupt type 1 is generated.

The kernel is invoked and tries to replace the instruction with a corresponding AMOS instruction, or to emulate the effect of the XAMOS instruction. In case the instruction could be replaced, it is then reexecuted together with preceeding modify instructions.

The table below defines the replacing instructions.

The execution speed of a program with XAMOS instructions is thus only effected once for each XAMOS instruction in the program.

Emulation/replacement of XAMOS instructions can be done in user programs and in system components only it cannot be done in the kernel itself when

- the monitor process executes with local interrupt bit set

- the I/O process executes with local interrupt bit set or

- the kernel executes in the context of a calling process after having saved the process by SVP(SSP).

XAMOS ins.	Corresponding AMOS ins.	Note
MMP	MVP	The formats of the instructions are incompatible: MMP M B0.X3 R3 MVP M B6.X2 R2 If the MMP cannot be replaced, it is considered illegal.
JPZI	JMPI S4	Preceeding modify instructions are not re-executed
RPZ	JMP	
RTM	JMP	
RTMI	JMPI	
LBR	MOV RO RO	A NOP which clears modifications
LDL	MOV RO RO	
SVL	MOVC 0	Level 0 assumed
SLS	LDS	
SSS	SVS clear bits 10..8	This instruction is emulated when destination is a register. It is considered illegal when destination is memory

XAMOS ins.	Corresponding AMOS ins.	Note
SLP	LDP	
SLN	LDN	
SSP	SVP	

Fig. E.1