SLAC-305
UC-32
(M)

NETWORK COMMUNICATION EXECUTIVE (NCX),
PROGRAMMER'S GUIDE*

Hans Frese
Tim Streater

*Stanford Linear Accelerator Center
Stanford University
Stanford, California 94305*

October 1986

Prepared for the Department of Energy
under contract number DE-AC03-76SF00515

Printed in the United States of America. Available from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, Virginia 22161. Price: Printed Copy A04, Microfiche A01.

* Manual

Table of Contents

1. PREFACE	2
1.1 INTENDED AUDIENCE	2
1.2 ADDITIONAL NCX INFORMATION	2
1.3 VM IMPLEMENTATION REQUIREMENTS	2
2. INTRODUCTION	3
2.1 GENERAL DESCRIPTION	3
2.2 TERMINOLOGY	3
2.3 DEVICE SUPPORT	4
2.4 USER PROGRAMS	4
3. NCX DATABASES	5
3.1 PROCESS QUEUES	5
3.2 PROCESS CONTROL BLOCK (PCB)	5
3.2.1 Structure of the PCB	5
3.3 SEMAPHORES	6
3.4 MAILBOXES	6
3.5 FREE MEMORY POOL	7
4. NCX PROCESS MANAGEMENT	7
4.1 NCX PROCESS FEATURES	9
4.2 THE FIRST NCX PROCESS	9
4.3 CREATING OTHER PROCESSES	9
4.3.1 Parent and Child Processes	9
4.4 HOW AN NCX PROCESS STARTS RUNNING	10
4.5 HOW AN NCX PROCESS STOPS RUNNING	10
4.5.1 Blocking and Unblocking	10
4.5.2 Functions that can cause a process to block	11
4.5.3 Other ways to unblock	11
4.5.4 Process Termination	11
4.5.5 The sched() Call	11
4.6 ENTRY POINTS	12
4.7 PROCESS STATES: A REVIEW	12
4.8 NCX TIMERS	14
4.8.1 How Timers Run	14
4.8.2 NCX Calls that Use Timers	14
5. NCX SEMAPHORES	15
5.1 SEMAPHORE DATA STRUCTURE	15
5.2 STATES OF A SEMAPHORE	15
5.3 SEMAPHORE ROUTINES	16

5.4	CREATING AND INITIALIZING SEMAPHORES	16
5.5	SETTING AND CLEARING A SEMAPHORE	16
5.6	RELEASING SEMAPHORES	16
5.7	REQUESTING SEMAPHORES	16
5.8	COMMUNICATING BETWEEN PROCESSES	17
5.9	ALLOCATING RESOURCES	17
6.	THE NCX SIGNAL MECHANISM	18
6.1	WHERE MESSAGES ARE STORED	18
6.2	CONDITIONS FOR RECEIVING SIGNALS	18
6.3	SIGNALING TO TERMINATE ANOTHER PROCESS	19
6.4	SIGNAL ROUTINES	19
7.	NCX MEMORY MANAGEMENT	20
7.1	NCX MEMORY MANAGEMENT ROUTINES	20
7.2	ALLOCATING MEMORY TO CREATE NCX PROCESSES	20
7.3	WARNINGS	20
8.	NCX QUEUE MANAGEMENT	21
8.1	DATA STRUCTURES FOR NCX QUEUES	21
8.2	NCX QUEUE MANAGEMENT ROUTINES	21
9.	THE NCX MAILBOX MECHANISM	22
9.1	CREATING MAILBOXES	22
9.2	DATA STRUCTURES	22
9.3	MAILBOX ROUTINES	23
10.	ERROR HANDLING IN NCX	24
10.1	TWO ERROR FIELDS	24
10.2	WRITING AND READING ERRORS	24
10.3	ERRORS AND THE SIGNAL MECHANISM	25
10.4	NCX CALLS THAT CAN GENERATE ERRORS	25
10.5	ERROR-HANDLING ROUTINES	25
10.6	ERROR CODES	25
11.	NCX ROUTINES	26
11.1	SUMMARY OF NCX ROUTINES	26
11.1.1	Process Management	26
11.1.2	Memory Management	26
11.1.3	Semaphore Management	26
11.1.4	Queue Management	26
11.1.5	Mailbox Management	26
11.1.6	Miscellaneous Services	26
11.1.7	VM Routines	26

11.2	Reference Descriptions of NCX Routines	28
12.	EXTRA FEATURES AVAILABLE WITH VM-NCX	61
12.1	NCX ROUTINES FOR VM	61
12.1.3	ncxpost() - Restore SYSMASK and post NCXECB	64
A.	Appendix: NCX DISTRIBUTION FILES FOR VM	64
A.1	INCLUDE FILES	64
A.2	SOURCE FILES (Limited Distribution)	65

ACKNOWLEDGEMENTS

This document is adapted from MICOM-Interlan's Network Communications Executive Programmer's Manual, Document Part Number 950-1045-00. Permission to use large portions of the MICOM-Interlan document was kindly given by MICOM-Interlan.

The authors acknowledge Les Cottrell and Teresa Downey for their efforts in formatting and editing this document, and Billie Bennett for the figures.

NOTICE

The software described in this document is subject to change without notice. It is furnished under a license and may be used or copied only in accordance with the terms of such license.

The information in this document is subject to change without notice and should not be construed as a commitment by MICOM-Interlan. While reasonable precautions have been taken, MICOM-Interlan assumes no responsibility for any errors that may appear in this document.

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of MICOM-Interlan.

PORTACOLS is a trademark of MICOM-Interlan, Inc.

VM and CMS are trademarks of International Business Machines Corporation.

Copyright (c) 1985 by MICOM-Interlan, Inc.

All Rights Reserved

Printed in the United States of America

1. PREFACE

This manual describes the internal structure and function of the NETWORK COMMUNICATIONS EXECUTIVE (NCX) software.

NCX runs in various environments, both hardware and software. This document describes the features that all versions of NCX have in common. It also presents the idiosyncrasies of the VM implementation of NCX. This information is to be found in chapter 12.

1.1 INTENDED AUDIENCE

This document is intended for VM programmers who have a need for the facilities of NCX. It assumes the reader has experience programming in assembly language and in the C language.

1.2 ADDITIONAL NCX INFORMATION

For more information on MICOM-Interlan products that use NCX, see the appropriate MICOM-Interlan user documentation, or contact your MICOM-Interlan representative.

Suggested Reading:

1. The NTS10 Network Manager's Guide MICOM-Interlan manufacturing part number 950-0014.
2. XNS ITP/MS-DOS Protocols Software User's Guide MICOM-Interlan manufacturing part number 950-1043-00.
3. Freeman, P., Software Systems Principles, A Survey Science Research Assocs., Inc., 1975.

1.3 VM IMPLEMENTATION REQUIREMENTS

The VM implementation of NCX is designed to be used with programs compiled with the Waterloo C compiler, or assembly language programs and routines which use the same register conventions as that compiler. The compiler should be run with at least the options as in the following example:

CW <filename> (ALIGN OPT PRM ibm 1 PRM ebclic 1.

2. INTRODUCTION

2.1 GENERAL DESCRIPTION

NCX is a software package that you link with C or assembly language programs, enabling these programs to create and schedule their own processes, communicate with these processes, and allocate resources among them. NCX is intended for use with MICOM-Interlan's network protocol software products, such as ITP (Internet Transport Protocols) and NTS (Network Terminal Server) software.

NCX can be described as a guest operating system; it includes a scheduler, memory management, and resource allocation via semaphores. Most versions of NCX do not provide or specify any device-driving software, and do not supersede the user program's relationship to the host operating system. (Some stand-alone versions of NCX handle devices, and their user programs do not use host operating system services.)

To use NCX, an application program links the NCX software into its own executable code. Most versions of NCX require that user programs follow specific compiling and linking procedures, using particular compilers and linking to special libraries.

A program that includes NCX software is not limited to NCX resources; it can use all the features of its compiler as well as the resources of its host operating system.

Many NCX services are of use to networking applications, such as:

1. NCX's fast context switching – that is, its ability to quickly suspend one process and run another; and
2. NCX's simple mechanisms for synchronizing processes and communicating among processes.

2.2 TERMINOLOGY

To distinguish NCX software from user-written software, and NCX processes from the host operating system, the following terminology is used throughout this document:

NCX: refers to the NCX software, which consists of several modules that are linked together with a user program to produce a single executable program.

NCX process: is a process created by a user program issuing a call to the NCX fork() routine.

Host process: is the process created by the host operating system to run the user program, which in turn uses NCX to create NCX processes.

User program: is a user-written, assembly-language or C-language program that makes calls to NCX routines. "User program" also refers to MICOM-Interlan's ITP and NTS software.

NCX system: refers to the user program, its NCX software, plus all the user program's NCX-created processes.

2.3 DEVICE SUPPORT

NCX does not specify an interface between I/O devices or I/O device drivers and NCX user programs. This lets NCX user programs handle I/O devices and their characteristics as the programmer wishes.

User programs can handle I/O in various ways – by using NCX semaphores, by using a virtual device driver, or by handling I/O devices directly, without involving NCX in any way.

2.4 USER PROGRAMS

To use NCX, your program must be written in C or assembly language. Your program calls NCX routines in the same way it calls regular C or assembly language routines. You compile the program using the compiler specified for your version of NCX, and you link your compiled program to the NCX software using the link utility specified for your version of NCX.

Since the interface between NCX and NCX user programs is always the same, programs that use NCX can be ported easily from one host system to another. Regardless of the host system, NCX provides the same basic services. It hides most of the host operating system's features from the user program, performing the essential task of obtaining these features on behalf of the program.

3. NCX DATABASES

To keep track of NCX's current status, the basic NCX routines use several fundamental databases. These databases vary slightly from one version of NCX to another. An introduction to some of the typical NCX databases is given in this chapter. Later chapters provide additional information on these databases.

For exact definitions of all the database structures in your particular version of NCX, refer to the "ncx.h" database distributed with your version.

WARNING: User programs should not alter these databases directly. You should access them only indirectly, via NCX calls.

3.1 PROCESS QUEUES

NCX maintains three major queues, which are linked lists. On operating systems that support multiple users (such as VAX/VMS), each user program that is linked with NCX will have one set of these queues.

1. The Main Queue is a linked list of all currently existing NCX processes.
2. Semaphore Queues are linked lists of those processes that have requested but not yet obtained a semaphore (resource). There is one queue per semaphore. A process on this queue is said to be "blocked" until it obtains the semaphore, at which time it is ready to run.
3. The Ready Queue is a linked list of those processes that are ready to run.

Each of these queues is a circular doubly-linked list with a forward and backward pointer; the last element's forward pointer points to the first element, and the first element's backward pointer points to the last element.

3.2 PROCESS CONTROL BLOCK (PCB)

The Process Control Block (PCB) is the primary NCX data structure for each NCX process. When a process is created (via a `fork()` call), a PCB for the new process is also created. The PCB acts as a "handle" for the process; it maintains the process state.

The PCB does the following:

1. Identifies the process for all activities, including creating, scheduling, running, and terminating the process, and communicating among processes.
2. Holds the process's place in the various NCX queues.
3. Posts error messages produced by the routine currently running in the various queues.

NCX routines use the PCB for various purposes. Some examples:

<code>fork()</code>	returns a pointer to a new process PCB.
<code>getpcb()</code>	reads the PCB of the currently running process.
<code>release()</code>	moves a PCB, if there is one, from a semaphore Wait Queue to the Ready Queue.
<code>request()</code>	if this call fails, it moves the caller's PCB onto a semaphore Wait Queue (blocks the process).

User programs running as NCX processes should never modify the PCB.

3.2.1 Structure of the PCB

The following is the PCB for the VM version of NCX.

```
struct pcb
{
    struct pcb *p_sqfl;    /* redyq forward pointer    */
    struct pcb *p_sqbl;    /* redyq backward pointer   */
    char      p_pri;       /* Process priority         */
    struct pcb *p_pqfl;    /* PCB list forward pointer */
    struct pcb *p_pqbl;    /* PCB list backward pointer */
    unsigned  *p_sp;       /* Value of process stack pointer */
    unsigned  *p_stak;     /* Address of bottom of stack */
    unsigned  p_ssiz;      /* Size of process stack */
    unsigned  *p_clok;     /* Pointer to clock */
    struct sem *p_state;   /* 0 = ready, 1 = running, n = psem */
    unsigned  p_err;       /* Process error number */
    unsigned  p_sig;       /* Signal someone with error */
    struct sem p_vig;      /* Process die semaphore */
}
```

3.3 SEMAPHORES

An NCX semaphore represents the state of a resource or an event. NCX semaphores are designed so that NCX processes can coordinate the sharing of resources.

A semaphore contains an integer that can be set to a negative value, a positive value, or zero. Each semaphore also contains a queue of the PCBs of processes waiting for the semaphore. Semaphores can contain or be contained by other data structures.

When representing an event, a semaphore is a way for NCX processes to communicate the state of the event to other processes. A process waiting for the event "requests" the semaphore. If the event has yet to take place, the semaphore is zero, and the "outstanding" request causes it to become negative (-1). When the event happens, another process (asynchronously/independently) "releases" the semaphore; the process that was waiting then gets the semaphore and eventually runs.

When a semaphore represents a resource, NCX processes compete to acquire it; when they acquire it, they can use the resource. When they finish using the resource, they release the semaphore, and other NCX processes can then acquire it.

Semaphores, resources, and the request/release mechanism are discussed further in Chapter 5.

3.4 MAILBOXES

Mailboxes let NCX processes send messages to each other. Mailboxes contain semaphores. They also contain a linked list of messages.

The mailbox system is described in Chapter 9.

3.5 FREE MEMORY POOL

The user program gains access to the free memory pool via calls to the `alloc()` and `free()` routines. NCX memory management is described in Chapter 7.

4. NCX PROCESS MANAGEMENT

This chapter describes how processes are created and managed under NCX.

An NCX process is an independently-schedulable segment of user-written program code. NCX processes are distinct from processes that are created and managed by the host operating system. Both the NCX processes and the NCX process scheduler are subject to the host operating system's scheduling of the user program.

Figures 1 and 2 illustrate host system processes and NCX processes.

Figure 1.

A host system containing three user processes, one of which contains a program that uses NCX.

Figure 2.

A host process containing a user program that uses NCX. This program has three NCX processes.

4.1 NCX PROCESS FEATURES

An NCX process has the following features:

1. Each NCX process is an independently-schedulable entity within the user-program's host process.
2. NCX processes can have different priorities. A priority is assigned to a process when the process is created (via an NCX fork() call). The range of priority values is usually 1 to 255, with 255 as the highest priority.
3. Each NCX process has a runtime stack, providing the process with its own context. The stack is allocated when the fork() routine is called to create the process.
4. One NCX process cannot be preempted by another until the first process causes itself to block or terminates.
5. NCX processes can be preempted as a group when the host operating system reschedules the entire user program of which they are a part.
6. NCX processes compete for shared resources.
7. NCX processes can dynamically allocate and free blocks of memory from the NCX memory pool.
8. NCX processes can communicate with each other in several ways – see the semaphore, signal, and mailbox mechanisms, in later chapters.
9. NCX processes can create new, independent processes – see the fork() call.
10. NCX processes can terminate other NCX processes that are willing to cooperate – see the signal() call.
11. NCX processes can terminate themselves – see the exit() call.
12. NCX processes can acquire and release resources – see Chapter 5, on semaphores.

4.2 THE FIRST NCX PROCESS

The first NCX process is automatically the main() routine in your NCX program, and it is the only NCX process that is not created by a fork() call. Instead, it is created by the NCX runtime library, with which you must link your program in order to use NCX. When you run your program, it uses a PCB called MANE, and runs it as the first process at priority 1. Thereafter, your program must issue fork() calls in order to create NCX processes.

4.3 CREATING OTHER PROCESSES

To create an NCX process other than main(), your program must issue an NCX fork() call. This creates a PCB (Process Control Block) for the new process and places the PCB onto the NCX Main Queue and the Ready Queue. The two routines – the main() routine that issued the fork() and the routine that becomes the new NCX process – are now two separately schedulable NCX processes.

A new NCX process can, in turn, create other new – and independent – processes. In other words, any NCX process can issue a fork() call. NCX permits the user program to create and use as many NCX processes as it wishes, providing the memory is available for each process. Not every one of those processes are guaranteed to run, however, especially if its priority is low and/or if some processes don't cooperate and give it a chance to run.

The fork() call is described in Chapter 11.

4.3.1 Parent and Child Processes

After an NCX fork() call creates a new process, there is no implicit relationship between the creating process (the “parent”) and the created process (the “child”).

If an NCX parent process calls the exit() routine (i.e., terminates), its child processes are not notified, and they continue to run independently.

In order to know who its child processes are, a parent process must record the child processes’ PCBs when it issues fork() calls. If the parent wants a child to be able to communicate with it, the parent must “publish” its own PCB – for example, by using a global variable or by passing its PCB as a parameter to the fork() routine that creates the child.

4.4 HOW AN NCX PROCESS STARTS RUNNING

A fork() call does not immediately start a new NCX process running, but merely places it onto the Ready Queue in the order of its given priority.

An NCX process actually begins to run when it has the next highest priority on the Ready Queue and the currently executing process stops running (temporarily or permanently). The next section discusses the ways in which this can occur.

4.5 HOW AN NCX PROCESS STOPS RUNNING

There are three situations in which an NCX process stops running: when it “blocks”, when it terminates (via an exit() call), and when it issues a sched() call.

4.5.1 Blocking and Unblocking

An NCX process blocks when it cannot obtain a resource that it asks for via the request() call. Typically, it unblocks when it has the highest priority on the resource’s semaphore Wait Queue and another NCX process issues a release() call for that semaphore. Blocking and unblocking occur as follows:

1. The current NCX process issues a request() call for a semaphore.
2. The semaphore is not available, i.e., its value is less than, or equal to, zero.
3. request() decrements the semaphore by one.
4. request() puts the PCB of the calling process onto the semaphore’s Wait Queue.
5. request() does not return – yet.
6. request() causes the highest-priority process on the NCX Ready Queue to run.
7. Sometime later, an NCX process issues a release() call that frees the semaphore for use.

This release() removes the highest priority process on the semaphore’s Wait Queue and puts it on the NCX Ready Queue. Eventually, the “highest priority” process on the semaphore Wait Queue will be the original request()ing process from step (1), above, and that process is finally unblocked – it is now on the NCX Ready Queue.

8. When the process that issued the release() blocks, NCX will eventually select the unblocked process from the NCX Ready Queue and run it.
9. When this unblocked process gets to run, its request() call finally returns – with the value TRUE. TRUE means the process successfully obtained the semaphore.

NOTE: When a process blocks, NCX saves the state of the process on the process stack; when the process unblocks, NCX ensures that the process resumes execution where it left off.

4.5.2 Functions that can cause a process to block

Request() is the only NCX call that blocks directly, but other NCX routines can block indirectly by calling request(). For example, sleep() blocks by creating a semaphore, making sure that the semaphore can never be released, and then issuing a request() call for the semaphore.

4.5.3 Other ways to unblock

In addition to release(), two other mechanisms can cause a process to unblock:

1. A signal() call from another NCX process; signal() always moves the blocked process's PCB from the semaphore Wait Queue to the NCX Ready Queue; or
2. A timer, set by the blocked process, expires; the timer service routine calls signal() to move the waiting process's PCB onto the NCX Ready Queue. (For information on timers, see the end of this chapter.)

4.5.4 Process Termination

An NCX process can terminate only by doing so itself. There are two ways for a process to terminate itself.

1. By calling the NCX exit() function.
2. By returning from the function that comprises the main body of the process. When this happens, the exit() routine is called automatically, on behalf of the process.

When an NCX process terminates, its PCB and stack space are returned to the memory pool. Since a process can only be terminated by itself, its PCB will not be found on any of the semaphore Wait Queues.

The exit() function does not free memory that the process has obtained by alloc() calls. The process must free this memory itself via calls to the free() function; otherwise, the memory will be lost to the NCX system.

After an exit() call, the NCX process ceases to exist – until it is created by a subsequent call to fork().

4.5.5 The sched() Call

If an NCX process does not block or terminate, it can run indefinitely. This is because NCX does not preemptively reschedule processes; that is, it does not automatically reschedule processes at regular intervals. It is therefore incumbent upon the NCX programmer to ensure that processes release the CPU. A process can do this by issuing a sched() call; sched() causes the scheduler to select the highest-priority process from the NCX Read Queue and run it.

Some processes need never call sched(); others should do so regularly. Examples:

1. A process that executes only a small amount of code before exiting does not need to issue sched() calls since it will return quickly.
2. A process that requests semaphores frequently enough to be blocked regularly does not need to issue sched() calls, since blocking always causes a reschedule.
3. On the other hand, a process that runs long loops in which it does not request any semaphores may monopolize the CPU. This program should build sched() calls (or sleep() calls) into its code at regular intervals.

For more on the signal mechanism, see Chapter 6. For more on NCX errors, see Chapter 10.

4.6 ENTRY POINTS

When it has a process to run, NCX determines where in the user code execution should begin, as follows:

1. When the process is first created by `fork()`, NCX enters the code at the point indicated by the first argument passed in the `fork()` call, which is usually the beginning of a procedure.
2. When the process blocks, NCX saves the state of the process on the process's stack, so that when the process eventually unblocks, NCX can ensure that the process resumes execution where it left off.

4.7 PROCESS STATES: A REVIEW

As has been discussed, an NCX process can be in one of three states: running, ready, or blocked. Figure 3 illustrates these three states and shows how an NCX process moves from one state to another. What follows is a review of the steps shown in the figure.

1. An NCX process is created by the `fork()` routine, and the process PCB is placed onto the Ready Queue.
2. NCX, or a `sched()` call from the currently running process, selects the highest priority process from the Ready Queue. Processes of the same priority are queued first-in-first-out on the Ready Queue.
3. NCX takes the process selected by `sched()` and runs it.
4. That process will run until it calls `sched()`, terminates, or issues a `request()` call.
 - (a) If the process calls `sched()`, the process is placed onto the Ready Queue.
 - (b) If the process issues a `request()` call and the semaphore is available, the process continues to run.
 - (c) If the process calls `request()` and the semaphore is not available, the process blocks.
 - (d) If the process terminates, it is no longer an NCX process.
5. A blocked process can get onto the Ready Queue in one of three ways:
 - (a) The NCX process successfully obtains a semaphore; that is, another NCX process calls `release()` for that semaphore. A `release()` always moves the highest-priority PCB from the semaphore's Wait Queue to the NCX Ready Queue.
 - (b) A timer (sometimes called a pseudo-clock in NCX), which the process has asked the NCX scheduler to run, expires. Timers are described in section 4.8.
 - (c) The process is `signal()`ed by another NCX process. The signal is written into the target process PCB, and the target process receives this signal the next time it runs.
6. At this point, the original process is back on the Ready Queue, and the cycle begins again.

Figure 3.
NCX Process States

4.8 NCX TIMERS

Timers provide a useful way for NCX processes to set limits on the length of time they wait for semaphores.

Every NCX process can have one active timer. To use the timer, the NCX process issues one of the NCX calls that start the timer running. These calls are listed below.

Once the timer starts running, the process is blocked, and the timer can be interrupted only by a `signal()` from another NCX process; otherwise, the timer simply runs out.

The caller specifies the length of time that the timer should run, in units of 10 ms. The actual elapsed time depends upon the clock granularity of the host machine.

4.8.1 How Timers Run

Every NCX PCB has a pointer to a timer. When this pointer is null, NCX assumes the process has no timer running. When the pointer is non-null, NCX follows the pointer and decrements the timer as follows:

1. The timer service routine is called by some system-dependent mechanism. Typically, this is an interrupt routine connected to the system clock.
2. The timer service routine walks down the Main PCB Queue and decrements all timers it finds.
3. If a timer expires, the timer service routine moves the PCB onto the NCX Ready Queue by issuing `signal()` with the message `E_TIMEOUT`.

4.8.2 NCX Calls that Use Timers

<code>request()</code>	creates a timer
<code>receive()</code>	can create timers because it calls <code>request()</code> .
<code>sleep()</code>	creates a timer by calling <code>request()</code> .
<code>vigil()</code>	creates a timer in the same way as <code>sleep()</code> .
<code>signal()</code>	turns off any timers it finds by setting the timer pointers to null.

5. NCX SEMAPHORES

An NCX semaphore is a shared data structure that contains:

1. An integer that represents the state of a resource or an event.
2. A queue head that points to PCBs of processes waiting for the resource.
3. A queue tail.

The acquire/release cycle, which under NCX is handled by two routines, `request()` and `release()`, comes from Dijkstra's model of a method for resolving deadlocks where two or more processes are competing for shared resources.

5.1 SEMAPHORE DATA STRUCTURE

The data structure of an NCX semaphore includes an integer and the head of a queue:

```
struct sem
{
    int sem_val;          /* how many units of resource avail. */
    struct q sem_pcbq;    /* head of queue containing          */
}                        /* waiting processes                */
```

5.2 STATES OF A SEMAPHORE

Unlike bit-flags, which have only two states, zero and one (or negative and positive), semaphores can be set to at least three states: zero, positive, and negative.

The positive and negative states, in turn, can be incremented and decremented to indicate the degree of a condition. (Releasing a semaphore causes it to increment. Requesting a semaphore causes it to decrement.)

The three states of a semaphore are:

1. **< 0 UNAVAILABLE:** One or more requests are waiting for the semaphore. How much less than zero indicates how many processes are waiting.
2. **= 0 IDLE:** No processes are waiting for the semaphore, and no resources are available.
3. **> 0 AVAILABLE:** The semaphore has been `release()`d for requests. How much greater than zero indicates how many times it has been released and, therefore, how many requests will succeed in obtaining the resource.

5.3 SEMAPHORE ROUTINES

The NCX semaphore management routines are:

<code>release()</code>	Release a semaphore
<code>request()</code>	Request a semaphore
<code>semini()</code>	Initialize a semaphore

5.4 CREATING AND INITIALIZING SEMAPHORES

An NCX process can create a semaphore by using the NCX data structure "sem" to declare the semaphore and then issuing the `semini()` macro to initialize the semaphore. The `semini()` macro sets the semaphore's value to 0 (zero) and calls `qini()` to initialize the semaphore's Wait Queue.

NOTE: A semaphore must be `release()`d at least once – by any process or program, either NCX or non-NCX – to make it greater than zero and hence available for use.

The semaphore's location must be "published" somehow – e.g., as a parameter or a global variable – to all processes that need it. This is because NCX does not notify all potential semaphore request()ers and release()rs of the existence of the new semaphore.

5.5 SETTING AND CLEARING A SEMAPHORE

Setting and clearing an NCX semaphore is done by two NCX routines: `request()` and `release()`. Normally, an NCX process `request()`s the semaphore, and the resource (or the process responsible for handling the resource) `release()`s the semaphore.

An NCX process can use a private semaphore; `sleep()` does this to keep its calling process off the Ready Queue until the calling process's timer expires.

5.6 RELEASING SEMAPHORES

Any NCX process can release a semaphore. For example, a device driver that sends and receives packets releases a semaphore when it receives a packet intended for a process. The device driver calls `release()` directly, or indirectly (for example, via `send()`), to increment the "packet-received" semaphore and thereby indicate that one more packet is available for a customer.

If `release()` is called, `release()` also moves the first PCB on the semaphore's Wait Queue (if any) to the NCX Ready Queue.

There is no such thing as an unsuccessful `release()`, except when the target semaphore doesn't exist (which can cause unpredictable results).

5.7 REQUESTING SEMAPHORES

Continuing the example above, the NCX process that wanted the packet has already called `request()` and is already on the semaphore's Wait Queue. Eventually, a `release()` call from another NCX process moves the waiting process's PCB from the Wait Queue to the NCX Ready Queue.

Sometimes, the semaphore is greater than zero because there is a packet waiting to be fetched. In this case, the `request()` call decrements the semaphore by 1 (one) and returns TRUE to the caller immediately. The caller continues to run and can now fetch the packet.

The caller can continue to fetch packets until the semaphore equals zero, which indicates there are no more packets to fetch.

When there are other processes waiting for the semaphore, the semaphore is negative. The `request()` call decrements the semaphore by 1 (one), but does not return immediately. Instead, `request()` places the caller's PCB on the semaphore's Wait Queue. The caller is now blocked and cannot run until the driver `release()`s the semaphore.

5.8 COMMUNICATING BETWEEN PROCESSES

The semaphore can be used to signify the occurrence of an event, such as the expiration of a timer. The `sleep()` call uses a timer this way; `sleep()` is described in Chapter 11.

Semaphores are also used in mailboxes, where the integer indicates the presence of a mail message (or the presence of processes waiting for messages) and the queue contains the messages.

5.9 ALLOCATING RESOURCES

By using semaphores, NCX processes can share resources and compete for them. Typical resources include mailbox messages, data packets going to and from a network, and timers.

Processes must wait in line when the resource is not available. The semaphore allocates the resource to only one process at a time.

As we have seen in the example above, when an NCX process wins the contention for a semaphore, its `request()` call returns `TRUE`, and the NCX process can then proceed to use the associated resource.

When an NCX process loses the contention for a semaphore, that process's PCB goes onto the semaphore's Wait Queue, and the `request()` does not return until the process either obtains the resource or gets `signal()`ed and moved off the Wait Queue and onto the Ready Queue.

After the process obtains the resource, the process performs whatever operations are appropriate with the resource, such as unloading a buffer, constructing a message header, or setting a status flag.

NCX processes use resources on the honor system. An NCX process must voluntarily request the appropriate semaphore before it uses a resource (unless, of course, the resource has no semaphore). A semaphore does not automatically prevent an NCX process from abusing a resource.

6. THE NCX SIGNAL MECHANISM

The NCX signal() mechanism lets an NCX process write a non-zero one-word value into a reserved location in the PCB of another NCX process.

Signaling is useful when one NCX process wishes to terminate another NCX process, remove it from a semaphore Wait Queue, or direct it to another course of action.

The signal() mechanism depends upon the cooperation of the process receiving the signal. A process is notified that it has been signalled when request(), or one of the NCX functions that calls request(), returns FALSE. Upon seeing this event, the process then must take the appropriate action, depending on the signal value returned by a call to geterr().

6.1 WHERE MESSAGES ARE STORED

Since the signal() call writes to NCX processes that are not currently running, it must make sure not to obliterate an error message that the target process has received but has not had a chance to read yet.

For this reason, NCX provides two error fields: one for regular error messages (P_ERR), and one for messages that arrive via signal() when the regular error message is already set (P_SIG). The P_SIG and P_ERR fields are in each NCX process PCB.

The NCX error mechanism is described in Chapter 10.

6.2 CONDITIONS FOR RECEIVING SIGNALS

The signal() routine works differently depending on whether the target process is blocked or ready to run.

Target Process is Blocked – signal() simply posts its message:

1. signal() moves the blocked target process's PCB from a semaphore Wait Queue to the NCX Ready Queue,
2. signal() posts the signal message in the target process's PCB's P_ERR field, and
3. signal() causes the target process's outstanding request() call to return FALSE.
4. The target process can receive further signal()s while it is still on the Ready Queue.

Target Process is in NCX Ready Queue – signal() must protect any messages that are already in P_ERR:

1. signal() posts the signal message in P_SIG (the pending error message field) rather than P_ERR (the regular error message field) in the target process's PCB. This is because there may already be an error posted in P_ERR, and the process may need to read it.
If there is already a message in P_SIG, signal() overwrites it only if the new message is E_KILL. Otherwise, signal() discards the new message.
2. The target process runs.

Target Process Runs – the signal() mechanism works as follows:

1. If the target process entered the Ready Queue because of another signal(), the target process's request() returns FALSE, and the target process can find out why by reading its own P_ERR field.
2. In any case, the target process can run, incurring further errors, until it issues a request() or a sched(), as follows:

Target Process Calls Request()

- (a) If the target process does not have a message pending in P_SIG, request() tries to obtain the semaphore.
- (b) If the target process does have a message pending in P_SIG, request() copies P_SIG into P_ERR, sets P_SIG to null, and puts the process back onto the Ready Queue.
In other words, NCX does not allow a process that has a pending signal error to complete a request() for a semaphore.

3. Target Process Calls Sched()

- (a) If the target process does not have a message pending in P_SIG, sched() places the process onto the Ready Queue, then selects the next process to run.
- (b) If the target process does have a signal message pending, sched() copies P_SIG into P_ERR, sets P_SIG to null, and puts the process onto the Ready Queue.

6.3 SIGNALING TO TERMINATE ANOTHER PROCESS

An NCX process can use signal() to ask another NCX process to terminate itself. To do so, a process issues signal() with an E_KILL message. After it issues signal(), the process should call vigil(), to make sure that the target process really terminated.

Keep in mind that a program must read its own error field whenever it gets to run. This means that after each request() call, a program should branch on a FALSE return and, if necessary, call geterr(), which reads P_ERR, before proceeding.

6.4 SIGNAL ROUTINES

The signal routines are:

signal()	Signals another NCX process
vigil()	Awaits target process's exit()
geterr()	Reads the signal() message

7. NCX MEMORY MANAGEMENT

NCX memory management is system-dependent. If the host operating system has a memory allocation scheme, NCX may (or may not) use it. If NCX is the host operating system, as in MICOM-Interlan's Network Terminal Server family of products, then NCX employs its own memory management stratagem.

7.1 NCX MEMORY MANAGEMENT ROUTINES

The NCX memory management routines are:

<code>alloc()</code>	Allocates memory to an NCX process
<code>free()</code>	Returns allocated memory to NCX

7.2 ALLOCATING MEMORY TO CREATE NCX PROCESSES

Whenever a new process is created (via the `fork()` routine), NCX automatically allocates:

1. a user-specified amount of stack space for the process, and
2. a Process Control Block

The NCX software permits a user program to create an essentially unlimited number of NCX processes, providing NCX has enough free memory to allocate stack space and PCB space for each new process.

NCX processes can request more memory from NCX with the `alloc()` call; they return this memory to NCX using the `free()` call. NCX automatically coalesces pieces of returned memory whenever possible.

7.3 WARNINGS

1. NCX processes must make sure to return the same memory with `free()` that they obtained with `alloc()`, not fragments of that memory.
In particular, before an NCX process terminates, it must return all the memory it requested. Otherwise, this memory will be lost to the NCX system, since NCX cannot recover the memory itself.
2. NCX does not zero-out memory when that memory is allocated or freed.
3. NCX does not protect one process's allocated memory from another process. A dangerous example: stack overflow from one process can overwrite a section of memory belonging to another process.

8. NCX QUEUE MANAGEMENT

The major NCX queues – the Main PCB Queue, the Process Ready Queue, and the Semaphore Wait Queues – have already been discussed in Chapter 4.

Other NCX queues are based on a common data structure, which defines a single queue cell. An NCX process must construct this cell; the process can use the NCX qini() call to initialize the queue.

8.1 DATA STRUCTURES FOR NCX QUEUES

All NCX queues are doubly linked. The first element is a forward link, the second is a backward link. Prioritized queues are followed by a “char” priority field (0 to 255). The NCX queue structure is defined in “ncx.h” and is called “q”. The definition of “q” is:

```
struct q
{
    struct q  *q_qlf;           /* forward pointer */
    struct q  *q_qlb;           /* backward pointer */
}
```

8.2 NCX QUEUE MANAGEMENT ROUTINES

The NCX queue management routines are:

qini()	Initializes a queue
insq()	Inserts a cell into a queue
insqp()	Inserts a cell into a queue by priority
remq()	Removes a cell from a queue
qempty()	Sees if the specified queue is empty
qpred()	Finds a queue cell's predecessor
qsucc()	Finds a queue cell's successor

9. THE NCX MAILBOX MECHANISM

The NCX mailbox system lets NCX processes send messages to each other. Unlike the `signal()` or semaphore mechanisms, which deal with “messages” of one word in length, the mailbox system allows messages of essentially unlimited length.

Each NCX mailbox is built using a semaphore to which is attached a queue of received messages.

To send a message to a mailbox, an NCX process uses `send()`, which inserts the associated message into the mailbox’s queue and then `release()`s the mailbox’s semaphore. Messages are enqueued by priority. Within a single priority level, messages are enqueued first-in-first-out.

To receive a message from a mailbox, an NCX process uses `receive()`. The `receive()` routine first requests the mailbox’s semaphore, and, if successful, removes the message from the mailbox’s queue. Higher priority messages are received before lower priority messages.

The mailbox semaphore, when positive, is a count of the number of messages in the mailbox, and, when negative, is a count of the number of processes waiting for messages.

9.1 CREATING MAILBOXES

Any NCX process can create a mailbox by declaring an “mbox” structure, such as “`struct mbox my_mail;`”. To initialize the mailbox, the process uses the NCX `mbini()` call.

The NCX process that creates the mailbox must make sure to communicate (via, for example, public variables or the `fork()` call) the existence of the mailbox to the appropriate NCX processes. This is because NCX does not notify all potential mail senders or mail receivers of the existence of a new mailbox.

9.2 DATA STRUCTURES

There are two data structures for mailboxes: the mailbox itself and the message that goes in the mailbox.

1. Mailbox Structure

```
struct mbox
{
    struct sem mbox_sem;
    struct q mbox_msgq;
}
```

2. Message Structure

```
struct msg
{
    struct q msg_msgq;    /* fwd. and bkwd. links */
    char msg_pri;         /* message priority */
}
```

Mailboxes are useful for moving network packets between NCX user processes and network device drivers.

9.3 MAILBOX ROUTINES

The mailbox management routines are:

<code>mbini()</code>	Initializes a mailbox
<code>send()</code>	Sends a mailbox message
<code>receive()</code>	Receives a mailbox message

10. ERROR HANDLING IN NCX

When an error occurs, NCX, the currently running NCX process, or the routine within the process that incurred the error, uses `seterr()` to post the error in the incurring process's PCB error field.

10.1 TWO ERROR FIELDS

Some errors need to be protected from being overwritten by later errors. To do this, NCX uses two error fields, which appear in every NCX process's PCB: the regular error field, `P_ERR`, and the pending error field, `P_SIG`.

Normally, all errors are posted to the target process's `P_ERR` field, either by `seterr()` or by `signal()`. However, when an NCX process is running or is on the Ready Queue, `signal()` writes its messages to `P_SIG` instead of `P_ERR`. This is to protect the signal message from being overwritten by a later error message.

By writing to `P_SIG` instead of `P_ERR`, `signal()` allows the target process to continue to run, incur errors, read them, and resolve them, until completion. Errors appear in the target process's `P_ERR` field, as usual. The errors can be read and set with `geterr()` and `seterr()` calls, respectively.

When the target process finishes running, NCX then transfers the signaled error from `P_SIG` to `P_ERR`. The next time the process runs, it can find the message that `signal()` sent by examining its own error field – `P_ERR` – with a `geterr()` call.

NOTE: If the target process happens to request() a semaphore, NCX does not permit the request() to go through, because the target process's `P_SIG` error field is full. Instead, NCX immediately moves the target process to the Ready Queue and copies `P_SIG` to `P_ERR`.

10.2 WRITING AND READING ERRORS

To record an error, an NCX process uses the `seterr()` call. The `seterr()` routine modifies the `P_ERR` field in the calling process's own PCB; `seterr()` does not read or modify the `P_SIG` error field.

To find out what errors have occurred, an NCX process uses `geterr()`. Once `geterr()` reads an error, it zeros out the `P_ERR` field; `geterr()` does not read or modify the `P_SIG` error field.

Universal NCX error numbers are listed in the table at the end of this chapter.

If an NCX process that incurs an error does not call `geterr()`, its errors can move up through several callers, as lower routines return to higher ones without any caller or callee interpreting the error. During this time, another error could occur and the error field would be changed to show the new error. When a routine finally got around to reading the error (using the `geterr()` routine), there would be no guarantee that it would be the original error.

To transmit an error message from one NCX process to another, NCX processes can use the signal mechanism.

10.3 ERRORS AND THE SIGNAL MECHANISM

When one NCX process signals another, `signal()` posts the message as follows:

1. The message goes to `P_ERR`, if the target process is on a semaphore Wait Queue.
NCX does not allow a process to be placed on a Wait Queue if the process has been signaled and has a message still in `P_SIG`. This means processes entering the Ready Queue can have one unread error at most: the error, if any, in `P_ERR`.
2. The message goes to `P_SIG`, if the target process is already on the Ready Queue. The process may have an unread message in `P_ERR` - i.e., the message that caused the process to get onto the Ready Queue in the first place.

The `signal()` routine then writes the message into the process's pending error field, `P_SIG`. However, if both `P_ERR` and `P_SIG` are already full, `signal()` overwrites `P_SIG`, unless `P_SIG` contains the `E_KILL` message, in which case `signal()` returns.

10.4 NCX CALLS THAT CAN GENERATE ERRORS

The NCX calls that can generate errors are:

```
alloc()  -- Allocates memory to an NCX process
fork()   -- Creates a new NCX process
receive() -- Receives a mailbox message
request() -- Requests a semaphore
sched()  -- Selects the next NCX process to run
sleep()  -- Suspends execution
vigil()  -- Awaits target process's exit()
```

10.5 ERROR-HANDLING ROUTINES

The error-handling routines in NCX are:

```
geterr() -- Reads the contents of P_ERR
seterr() -- Writes the contents of P_ERR
signal() -- Writes the contents of P_ERR or P_SIG,
           as appropriate.
sched() and
request() -- If P_SIG is not equal to 0 (zero),
             copies P_SIG into P_ERR and sets P_SIG
             to zero.
```

10.6 ERROR CODES

These error codes are based on the error formatting of the PDP-11. All error codes are 16 bits (4 hexadecimal digits) long and can be found in the "ncxerr.h" file.

11. NCX ROUTINES

This chapter begins with a summary of NCX routines, grouped by function, and then provides a description of each routine, in alphabetical order.

Some routines are implemented as macros by #defines in "ncx.h".

Note to reader: all NCX routines start with c_ however, all preceding chapters have not given this prefix since the text is intended to be common to all NCX versions.

11.1 SUMMARY OF NCX ROUTINES

11.1.1 Process Management

c_exit()	Stops running and releases PCB and stack
c_fork()	Creates a new NCX process
c_geterr()	Reads the caller's current error message
c_getpcb()	Reads the caller's PCB
c_sched()	Selects the next NCX process to run
c_seterr()	Sets the caller's error message
c_signal()	Signals another NCX process
c_sleep()	Suspends execution
c_vigil()	Awaits target process's exit()

11.1.2 Memory Management

c_alloc()	Allocates memory to an NCX process
c_free()	Returns allocated memory to NCX

11.1.3 Semaphore Management

c_release()	Releases a semaphore
c_request()	Requests a semaphore
c_semini()	Initializes a semaphore

11.1.4 Queue Management

c_insq()	Inserts a cell into a queue
c_insqp()	Inserts a cell into a queue by priority
c_qempty()	Sees if the specified queue is empty
c_qini()	Initializes a queue
c_qpred()	Finds a queue cell's predecessor
c_qsucc()	Finds a queue cell's successor
c_remq()	Removes a cell from a queue

11.1.5 Mailbox Management

c_mbini()	Initializes a mailbox
c_receive()	Receives a mailbox message
c_send()	Sends a mailbox message

11.1.6 Miscellaneous Services

c_crash()	Responds to unrecoverable error
c_disable()	Turns host system's interrupts OFF
c_enable()	Turns host system's interrupts ON
c_movc3()	Moves bytes
c_movc3s()	Moves bytes and swaps
c_panic()	Displays an error message
c_random()	Provide 32 bit random integer
c_swap()	Swaps bytes
c_swapw()	Swaps words

11.1.7 VM Routines

The following VM routines are discussed in detail in Chapter 12.

```
c_stkper()  
  c_trap()  
ncxpost()
```

11.2 Reference Descriptions of NCX Routines

11.2.1 c_alloc() – Allocate memory to an NCX process

Purpose

The `c_alloc()` routine allocates memory to an NCX process. The memory allocated is from the space that NCX has obtained from the host system. See also `c_free()`.

Calling Sequence

```
char *c_alloc(n_bytes)
```

Parameters

```
UNSIGNED n_bytes;      /* number of bytes requested */
```

Side Effects

None.

Returns

Memory address – memory was allocated successfully.

NIL – requested memory is not available.

11.2.2 c_crash() – Respond to an unrecoverable error

Purpose

The purpose of `c_crash()` is to bring the NCX system to a halt, in response to a major error. It results in a call to the C library routine `exit()`, not to be confused with `c_exit()`.

Calling Sequence

```
VOID c_crash();
```

Parameters

None.

Side Effects

Call `exit()` and returns to CMS.

Returns

The `c_crash()` function does not return to the caller. (The caller will never execute code following a `c_crash()` call.)

11.2.3 c_disable() – Turn hosts interrupts off

Purpose

Use `c_disable()` to disable hardware interrupts from devices on the host system. This is useful for preventing the current NCX process and an interrupt service routine outside the NCX world from simultaneously accessing the same data. It is also useful for disabling interrupts prior to certain critical code. See `c_enable()`.

`c_disable()` calls may be nested; each `c_disable()` must be matched by a corresponding `c_enable()` call. For example, after the fragment of code shown below, interrupts remain disabled:

```
c_disable();
c_disable();
c_enable();
[no matching enable here]
```

Calling Sequence

```
VOID c_disable()
```

Parameters

None.

Side Effects

The global variable `CSYSMASK` is updated to reflect the disable/enable level.

Returns

No return value.

11.2.4 c_enable() – Turn hosts interrupts ON

Purpose

Use `c_enable()` to enable interrupts. This is useful after `c_disable()` has disabled them. The `c_enable()` routine does not enable interrupts until the number of `c_enable()` calls equals the number of `disable()` calls. See `c_disable()`.

Calling Sequence

`VOID c_enable()`

Parameters

None.

Side Effects

See `c_disable()`.

Returns

No return value.

11.2.5 c_exit() – Stop running and release PCB and stack

Purpose

An NCX process calls `c_exit()` in order to stop running and return its stack and PCB to NCX. After `c_exit()` completes, the caller no longer exists as a schedulable NCX process.

Calling Sequence

`VOID c_exit()`

Parameters

None.

Side Effects

All processes waiting for this process's exit with a `c_vigil()` will be unblocked, then `c_exit()` causes NCX to remove the highest-priority process from the Ready Queue and run that process.

Returns

The `c_exit()` routine does not return.

11.2.6 c_fork() – Create a new NCX process

Purpose

The `c_fork()` routine creates a new NCX process and allocates a stack and Process Control Block (PCB) for the new process. The `c_fork()` routine returns a pointer to the new process's PCB. This pointer is valid while the newly-created process is active. The pointer becomes invalid and should NOT be used after the new process exits or otherwise terminates.

The `c_fork()` routine inserts the new PCB into the Main PCB Queue(). PCBs are inserted in the order in which they are created.

If `c_fork()` fails, it will set `P_ERR` and return `NIL` pointer to the caller.

The data type for a process is `INT`.

Calling Sequence

```
struct pcb *c_fork(function, arg, pri, stack)
```

Parameters

The caller must supply a priority, a stack size, the name of the routine that will run in the new process, and an argument for the routine.

```
PROCESS *function(); /* function to run as NCX PROCESS */
UNSIGNED arg,        /* argument to pass to function */
        pri,         /* process priority 1-255. */
        stack;       /* stack size that will be */
                    /* provided for the process in */
                    /* 32-bit words */
```

Side Effects

None.

Returns

PCB address – process successfully created.

`NIL` – insufficient resources.

11.2.7 c_free() – Return allocated memory to NCX

Purpose

The `c_free()` routine returns to the NCX memory pool memory that an NCX process previously obtained with an `c_alloc()` call.

Calling Sequence

```
VOID c_free(ptr, n_bytes);
```

Parameters

```
char *ptr;          /* base of block to free */
UNSIGNED n_bytes; /* number of bytes to free */
```

Side Effects

WARNING: Make sure that `c_free()` returns the entire chunk of memory that was obtained with the complementary `c_alloc()` call, not just a fragment of the memory. NCX does not retrieve memory that is not returned to it. Most versions of NCX do not check returned memory, as a piece or as a whole.

Returns

No return value.

11.2.8 c_geterr() – Read the callers current error message

Purpose

The `c_geterr()` routine gets the current contents of the `P_ERR` field in the calling process's PCB. The `c_geterr()` routine does not read or modify the `P_SIG` error field.

Both `c_signal()` and `c_seterr()` may change the value in `P_ERR`.

Calling Sequence

```
UNSIGNED geterr(); /* returns current process error code */
```

Parameters

None.

Side Effects

None.

Returns

Error code – last error set in `P_ERR`.

11.2.9 c_getpcb() – Read the callers PCB

Purpose

The `c_getpcb()` routine obtains the PCB of the NCX process that is currently running; in other words, it lets the caller read its own PCB.

Calling Sequence

```
struct pcb *c_getpcb(); /* get PCB of current process */
```

Parameters

None.

Side Effects

None.

Returns

The routine returns the PCB address of the current process. It cannot produce an error.

11.2.10 c_insq() – Insert a cell into a queue

Purpose

The `c_insq()` routine inserts a cell into a queue. It turns interrupts off and on briefly.

Calling Sequence

```
struct q *c_insq(item, pred)
```

Parameters

The `item` parameter is the cell to be inserted; “`pred`” is the cell after which “`item`” is to be inserted.

```
struct q
    *item, /* queue element to insert */
    *pred; /* insert after this element */
```

Side Effects

None.

Returns

The routine returns the address of the queue element “`item`”.

11.2.11 c_insqp() – Insert a cell into a queue by priority

Purpose

The `c_insqp()` routine inserts a cell into the specified queue, according to priority. Items of the same priority are ordered in a first-in first-out basis. The highest priority item is at the front of the queue.

The `c_insqp()` routine assumes that each cell in the queue has a priority byte, and that this byte immediately follows the cell's backward pointer. Both the PCB (Process Control Block) and MSG (mailbox message) data structures have priority bytes in this position, so `c_insqp()` is useful for enqueueing these structures.

Calling Sequence

```
struct q *c_insqp(item, head)
```

Parameters

```
struct q *item;    /* item to insert */
struct q *head;    /* queue to insert element into */
```

Side Effects

None.

Returns

The routine returns the address of the queue element "item". The "item" may actually be any sort of structure; the only limitation is that a "struct q" and priority are expected as its first elements.

11.2.12 c_mbini() – Initialize a mailbox

Purpose

The `c_mbini()` routine initializes a mailbox. The user program must create the mailbox first.

Calling Sequence

```
VOID c_mbini(mboxa)
```

Parameters

```
struct mbox *mboxa; /* mailbox to initialize */
```

Side Effects

The `c_mbini()` routine calls `c_semini()` and `c_qini()`.

Returns

No return value.

Example

```
struct mbox my_box; /* declaration */  
c_mbini (&my_box); /* initialization */
```

11.2.13 c_movc3() – Move bytes

Purpose

The `c_movc3()` routine moves a specified number of bytes (“count”) from “src” (source) to “dst” (destination).

Calling Sequence

```
VOID c_movc3(src, dst, count)
```

Parameters

```
char    *src, *dst;  
UNSIGNED count;
```

Side Effects

None.

Returns

No return value.

11.2.14 c_movc3s() – Move bytes and swap

Purpose

The `c_movc3s()` routine moves a specified number of bytes (“count”) from “src” (source) to “dst” (destination), and swaps the two bytes within each word.

This routine does not swap the last byte if “count” specifies an odd number of bytes.

NOTE: The swap is performed only on systems (such as the Intel 8088 and 8086) that store bytes in lowest-to-highest order. On systems (such as IBM 360/370) that store bytes in the reverse order, no swap is actually performed. Actually, `c_mov3cs` is `#defined` to be `c_mov3c`.

Calling Sequence

```
VOID c_movc3s(src, dst, count)
```

Parameters

```
char    *src,*dst;  
UNSIGNED count;
```

Side Effects

None.

Returns

No return value.

11.2.15 c_panic() – Display an error message

Purpose

The `c_panic()` routine formats a message and outputs it to the “console” port. `C_panic()` is useful as a fail-safe method for outputting error messages.

In VM, `c_panic()` calls `printf()` and `fflush()`.

Calling Sequence

```
VOID c_panic(format, data ... )
```

Parameters

Like `printf()`, `c_panic()` requires that the caller specify the format of the data to be printed as well as the data itself.

Side Effects

None.

Returns

No return value.

11.2.16 c_qempty() – See if the specified queue is empty

Purpose

The `c_qempty()` routine checks to see if a queue is empty.

Calling Sequence

`BOOLEAN c_qempty(qa)`

Parameters

`struct q *qa; /* header cell of the queue to be tested */`

Side Effects

None.

Returns

TRUE - if queue is empty. FALSE - if there is an element in the queue.

11.2.17 c_qini() – Initialize a queue

Purpose

The `c_qini()` routine initializes a header cell for a queue. The forward and backward pointers are both set to the header cell itself.

Calling Sequence

```
VOID c_qini(qa)
```

Parameters

```
struct q *qa; /* queue to initialize */
```

Side Effects

None.

Returns

No return value.

11.2.18 c_qpred() – Find a queue cells predecessor

Purpose

The c_qpred() routine obtains the predecessor of the given queue element.

Calling Sequence

```
struct q *c_qpred(qe)
```

Parameters

```
struct q *qe; /* queue element whose      */  
              /* predecessor is desired */
```

Side Effects

None.

Returns

The routine returns the address of the queue element that was linked before “qe”.

11.2.19 c_qsucc() – Find a queue cells successor

Purpose

The `c_qsucc()` routine gets the successor of the given queue element.

Calling Sequence

```
struct q *c_qsucc(qe)
```

Parameters

```
struct q *qe; /* queue element whose */  
              /* successor is desired */
```

Side Effects

None.

Returns

The routine returns the address of the queue element linked after “qe”.

11.2.20 c_random() – provide random number

Purpose

The `c_random()` routine reads the CPU's time-of-day clock to provide random 32 bit integers for networking purposes such as “numbering” incarnations of connections.

Calling sequence

```
unsign32  c_random()
```

Parameters

None.

Side Effects

None.

Returns

32 bit random integer.

11.2.21 c_receive() – Receive a mailbox message

Purpose

The `c_receive()` routine waits a specified amount of time for a message to arrive at a specified mailbox. If a message arrives before the time has elapsed, the routine returns a pointer to the message. If the time expires without a message arriving, the routine returns NIL. Whenever `c_receive()` has to wait for a message, NCX causes a scheduling pass.

Calling Sequence

```
struct msg *c_receive(mbox, pclock);
```

Parameters

```
struct    mbox *mbox; /* mailbox where message will arrive */
UNSIGNED *pclock;    /* pseudo-clock or NIL pointer          */
```

Side Effects

None.

Returns

pointer to message - if message received.
NIL - if signal received.

11.2.22 c_release() – Release a semaphore

Purpose

The `c_release()` routine increments by 1 the value of the specified semaphore. By doing so, `c_release()` frees an associated resource for use by other callers.

The `c_release()` routine also unblocks the highest-priority NCX process that is waiting on the semaphore's wait queue (if there is one waiting) and places it on the NCX Ready Queue. It also turns off the timer (if any) of the waiting process.

Calling Sequence

```
VOID c_release(sema);
```

Parameters

```
struct sem *sema; /* semaphore to release */
```

Side Effects

Although `c_release()` adjusts the Ready Queue, it does not perform a reschedule; this means the caller remains running until a reschedule takes place via `c_sched()`.

Returns

No return value.

11.2.23 c_remq() - Remove a cell from a queue

Purpose

The `c_remq()` routine removes an indicated element from a queue.

This routine works on links named `q_qlf` and `q_qlb`. These are the forward and backward links, respectively, in the standard NCX queue cell. The routine returns a pointer to the element, whether the element was actually removed from the queue or not.

`c_remq()` turns interrupts off and on briefly.

Calling Sequence

```
struct q *c_remq(item)
```

Parameters

```
struct q *item;    /* element to be removed */
```

Side Effects

None.

Returns

queue element address - if element available.

"item" - if queue is empty.

11.2.24 c_request() – Request a semaphore

Purpose

The `c_request()` routine attempts to obtain a semaphore. The actions of the routine depend upon the value of the semaphore and of the `P_SIG` field in the PCB.

A semaphore is unavailable whenever it is less than zero, which means that another caller has requested it and has not released it.

If the caller has been signaled – that is, if `P_SIG` is non-zero – `c_request()` returns immediately without investigating the semaphore.

If `P_SIG` is zero, then:

1. If the semaphore is available, `c_request()` decrements the semaphore and returns `TRUE`. The caller continues to run.
2. If the semaphore is not available, the caller blocks – that is, the caller's PCB is linked onto the semaphore's wait queue, and the NCX kernel selects another NCX process to run.

The “`pclock`” argument specifies the address of the timeout value for the calling process timer, in the event that the caller block.

Calling Sequence

```
BOOLEAN c_request(sema, pclock);
```

Parameters

```
struct sem *sema; /* the requested semaphore */
UNSIGNED *pclock; /* pseudo-clock or NIL pointer */
```

NOTE: If `NIL` is specified, the request will be pending forever.

Side Effects

None.

Returns

`TRUE` - resource obtained. `FALSE` - if signal received or specified timer expired.

11.2.25 c_sched() – Select the next NCX process to run

Purpose

The `c_sched()` routine examines the Ready Queue. If there is at least one process waiting, `c_sched()` places the PCB of the current process onto the Ready Queue, according to the priority of the process, and runs the highest priority process. If there is no other process on the Ready Queue, `c_sched()` continues to run the current process.

If two processes have the same priority, `c_sched()` runs the one that has been on the Ready Queue the longer amount of time.

Calling Sequence

```
BOOLEAN c_sched();
```

Parameters

None.

Side Effects

None.

Returns

FALSE - if current (i.e., calling) process had previously been signaled, or if it was suspended and an error was detected while it was suspended.

TRUE - if there were no signals or errors.

11.2.26 c_semini() – Initialize a semaphore

Purpose

The `c_semini()` routine initializes a semaphore's value to 0 (zero) and initializes the semaphore's Wait Queue. This routine does not create the semaphore.

NCX does not notify potential requesting and releasing processes of the existence and name of the semaphore; the programmer must do so.

Calling Sequence

```
VOID c_semini(sema)
```

Parameters

```
struct sem *sema; /* semaphore address */
```

Side Effects

None.

Returns

No return value.

11.2.27 c_send() – Send a mailbox message

Purpose

The `c_send()` routine sends a message to a mailbox. `C_send()` calls `c_release()` for the mailbox's semaphore.

Calling Sequence

```
VOID c_send(mbox, msg);
```

Parameters

```
struct mbox *mbox; /* target mailbox */
struct msg *msg;   /* message to send */
```

Side Effects

None.

Returns

No return value.

11.2.28 c_seterr() – Set the callers error code

Purpose

The `c_seterr()` routine sets the caller's error code. It writes to the `P_ERR` field in the calling process's PCB (process control block).

Calling Sequence

```
VOID c_seterr(err_val)
```

Parameters

```
UNSIGNED err_val; /* error code to set */
```

Side Effects

The `c_seterr()` routine may overwrite an error message that the process has not read yet.

Returns

No return value.

11.2.29 c_signal() – Signal another NCX process

Purpose

The `c_signal()` routine writes the specified value into the target process PCB, as follows:

1. If the target process is in a resource wait queue, `c_signal()` posts the message in `P_ERR`, removes the process PCB from that wait queue, and inserts the PCB on the Ready Queue.
2. If the target process is in the Ready Queue, waiting to run, `c_signal()` posts the message in `P_SIG`, since the process may already have an unread message in `P_ERR`.
3. If both `P_ERR` and `P_SIG` are already full, `c_signal()` overwrites `P_SIG`, unless `P_SIG` contains the `E_KILL` message. In that case, `c_signal()` returns.

Calling Sequence

```
VOID c_signal(pcb, sig_val);
```

Parameters

```
struct pcb *pcb;      /* target process */
UNSIGNED sig_val;     /* value to pass */
```

Side Effects

None.

Returns

No return value.

11.2.30 c_sleep() – Suspend execution

Purpose

The `c_sleep()` routine suspends execution of the calling process for a specified amount of time. Other processes execute in the meantime.

`C_sleep()` initializes and then requests a new semaphore, and creates a timer. The semaphore is never released, so the caller's request always puts the caller onto the semaphore's wait queue.

When a timer expires, a system-specific timer routine signals the timer's calling process – in this case, it is the process that called `c_sleep()` – and puts the process onto the Ready Queue.

Specify the time to sleep in 10 millisecond units. Each tick of the timer decrements the time parameter by 1, but actually represents 10 milliseconds of wall-clock time.

The `c_sleep()` routine returns prematurely if the calling process was signaled.

Calling Sequence

```
BOOLEAN c_sleep(time)
```

Parameters

```
UNSIGNED time;    /* time to sleep in 10ms units */
```

Side Effects

None.

Returns

TRUE - if sleep completed successfully, i.e., the full time expired.
FALSE - if signal received before the time expired.

11.2.31 c_swap() – Swap bytes

Purpose

The `c_swap()` routine swaps the position of the bytes in the specified 16-bit word, “w”.

This routine is useful for loading data into packets in such a way that the most significant byte goes in first.

Calling Sequence

```
UNSIGN16 c_swap(w)
```

Parameters

```
UNSIGN16 w;
```

Side Effects

None.

Returns

The value of “w” with bytes swapped.

11.2.32 c_swapw() - swap words

Purpose

The `c_swapw()` routine swaps words of the specified longword, "lw", if the "remote" parameter is TRUE. If not, the specified longword is returned as is.

Calling Sequence

```
UNSIGN32 c_swapw(lw, remote)
```

Parameters

```
BOOLEAN remote  
UNSIGN32 lw
```

Side Effects

None.

Returns

The value of "lw" with words swapped.

11.2.33 c_vigil() – Await target process exit

Purpose

The `c_vigil()` routine detects the completion of an `c_exit()` call in the target process. Specifically, `c_vigil()` requests a special semaphore in the target process's PCB, and the `c_exit()` call always releases this semaphore.

`c_vigil()` is particularly useful for ensuring that a target process has actually terminated itself – via `c_exit()` – after receiving a `c_signal()` `E_KILL` message. If the target process never issues `c_exit()`, `c_vigil()` never returns – unless a timer value is specified.

Two things can delay and obscure the target process's self-termination: if `c_exit()` gives no notification of its own completion, or if the target process cannot give the `exit()` call until it is scheduled to run.

Calling Sequence

`BOOLEAN c_vigil(pcb, pclock)`

Parameters

```
struct pcb *pcb;    /* PCB of process to die      */
UNSIGNED *pclock;   /* pseudo-clock or NIL pointer */
```

Side Effects

None.

Returns

`TRUE` - if target process issued `exit`. `FALSE` - if signal received.

WARNING: If `c_signal()` is received, do not call `c_vigil()` again, since the PCB may no longer be valid.

12. EXTRA FEATURES AVAILABLE WITH VM-NCX

To use the VM version of NCX, a C-language program must be compiled under the Waterloo C compiler.

You may also write an assembly language program that uses NCX, provided that your program adheres to the calling conventions of the Waterloo C compiler.

12.1 NCX ROUTINES FOR VM

The following routines are specific to the VM version of NCX. These routines should be used with discretion, since they make the NCX application non-portable.

<code>c_stkper()</code>	Enable/disable SDS yellow stack region check
<code>c_trap()</code>	Force OC1 trap
<code>ncxpost()</code>	Post NCXECB if necessary

The routines are described on the following pages.

12.1.1 c_stkper() – Enable/disable yellow stack zone checking

Purpose

The 30xx PER hardware is used to trap instructions addressing the dangerous zone close to the end of a processes stack. Switching the PER hardware using SDS every time a process switch occurs introduces substantial overhead, so this routine should only be used for debugging purposes.

Calling Sequence

```
VOID c_stkper(flag);
```

If flag is TRUE, checking is enabled.

If flag is FALSE, checking is disabled.

Parameters

```
BOOLEAN flag;
```

Side Effects

No other PER traps should be active when using c_stkper since this would lead to an interrupt into CP after every instruction with corresponding overhead and uselessness of any real-time behaviour. Also, the MVCLS instruction can produce spurious PER traps.

Returns

None.

12.1.2 c_trap() – Generate instruction trap

Purpose

Provides go/no go indicator of a program reaching a certain statement without any change in real time behaviour.

Calling Sequence

```
VOID c_trap();
```

Side Effects

Traps to CP, producing an OC1 trap.

Returns

None.

12.1.3 ncxpost() – Restore SYSMASK and post NCXECB

Purpose

Any real life application of NCX will involve the use of timers and other hardware devices which will signal completion by generating interrupts. This routine provides the interface between interrupt handlers and NCX. Normally, an exit routine is called from CMS with a specific register convention. It should then clear CSYSMASK to reflect the interrupt state, and in order to inform waiting processes that the external event has completed, should release a semaphore that the programmer has associated with the external device. Finally, if no process was running at the time the interrupt occurred, then NCX will have entered an OS/370 wait. This routine will post the ECB for that wait, so that NCX will resume execution and can reschedule the process that was waiting on the semaphore.

Calling Sequence

```
VOID ncxpost();
```

Parameters

None.

Side Effects

CSYSMASK is restored to -1, its original value before the exit routine was entered from CMS.

Returns

None.

A. Appendix: NCX DISTRIBUTION FILES FOR VM

A.1 INCLUDE FILES

1. TYPES H: special NCX data types.
2. ERRDEF H: error code format support; contains macros that handle the standard error code format used by NCX.
3. NCXERR H: NCX error code definitions; contains the error codes specific to NCX, and NCX calls it automatically.
4. NCX H: defines the basic NCX data structures: queues, PCBs, etc.
5. OSHEAD H: defines "P", which is a pointer to the current process, and "PCBS", which is the head of the PCB list.

A.2 SOURCE FILES (Limited Distribution)

1. NCX ASSEMBLE