RESEARCH SERIES

NISTIR 88-3838

COMMON MEMORY FOR THE PERSONAL COMPUTER

August 11, 1988

By:
Siegfried Rybczynski

National Institute of Standards and Technology
formerly
U.S. DEPARTMENT OF COMMERCE National Bureau of Standards Gaithersburg, Maryland

Common Memory for the Personal Computer

Siegfried Rybczynski

August 11, 1988

DISCLAIMER

Certain commercial equipment, instruments, or materials are identified in this document in order to adequately specify the experimental procedure. Such identification does not imply recommendation or endorsement, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

Table of Contents

List Of Figures

I. INTRODUCTION

1. ORGANIZATION OF THIS DOCUMENT

Section I serves as an introduction to the concept of common memory. It identifies the purpose of the accompanying software collection, and lists the personal computer (PC) configuration prerequisites of the PC common memory software library.

Section II provides an overview of the common memory architecture defined by the Automated Manufacturing Research Facility (AMRF) of the National Bureau of Standards. A short personal opinion of the drawbacks to common memory is provided at the end of the section.

Section III specifies the PC common memory architecture. The AMRF architecture allows for a range of solutions to meet specific management and access coordination problems. This section identifies which solutions (or alternatives) were implemented in the PC common memory architecture. Some minor extensions to the original architecture are presented.

Section IV is the programmer's reference section. It lists all PC common memory function calls and details their argument lists and return values. A functional description of each function is provided.

The various appendices provide detailed support information for the use of the PC common memory program library. Included are data structure definitions, a sample program and associated output, and the complete source listing of the PC common memory library.

2. HISTORY OF COMMON MEMORY

The Automated Manufacturing Research Facility (AMRF) at the National Bureau of Standards (NBS) is using an architecture called "common memory" for interprocess communication.

The AMRF common memory architecture originated as a consequence of an application that required real-time data reduction with concurrent robot control [5]. To meet that need, a multiprocessor configuration that included a physical common memory entirely contained within a single Multibus chassis was used. Each of the processors was a single-board computer. The memory designated for common use by all processors was resident on a separate board and had an address range that mapped into the address space of each of the processors. Hence, local common memory was defined to be a contiguous area of physical memory accessible to two or more distinct processes within a single computer system (Figure I-1).

The AMRF began work on an automated factory in 1981 [6]. The AMRF automated factory concept held that the old idea of a single huge computer controlling all machines in the factory was too inflexible. Instead, computer processes such as control programs would run on many different computers, of all sizes and models, and might possibly be located in different buildings.

This extended the local common memory concept of the robot control system in many ways. A major extension was the linking and coordination of local common memories using network services. That is, processes which had to communicate with each other were often in separate backplanes and used different operating systems. A single physical common memory was no longer possible or practical, so each computer system had to have its own local common memory. These common memories were connected using network services that were transparent to the application process. The linked local common memories established a global common memory (Figure I-2). The information contained therein was considered to represent a global memory-resident database.

A discussion of the AMRF network architecture is beyond the scope of this document. Detailed information about that architecture is available in reference [7].

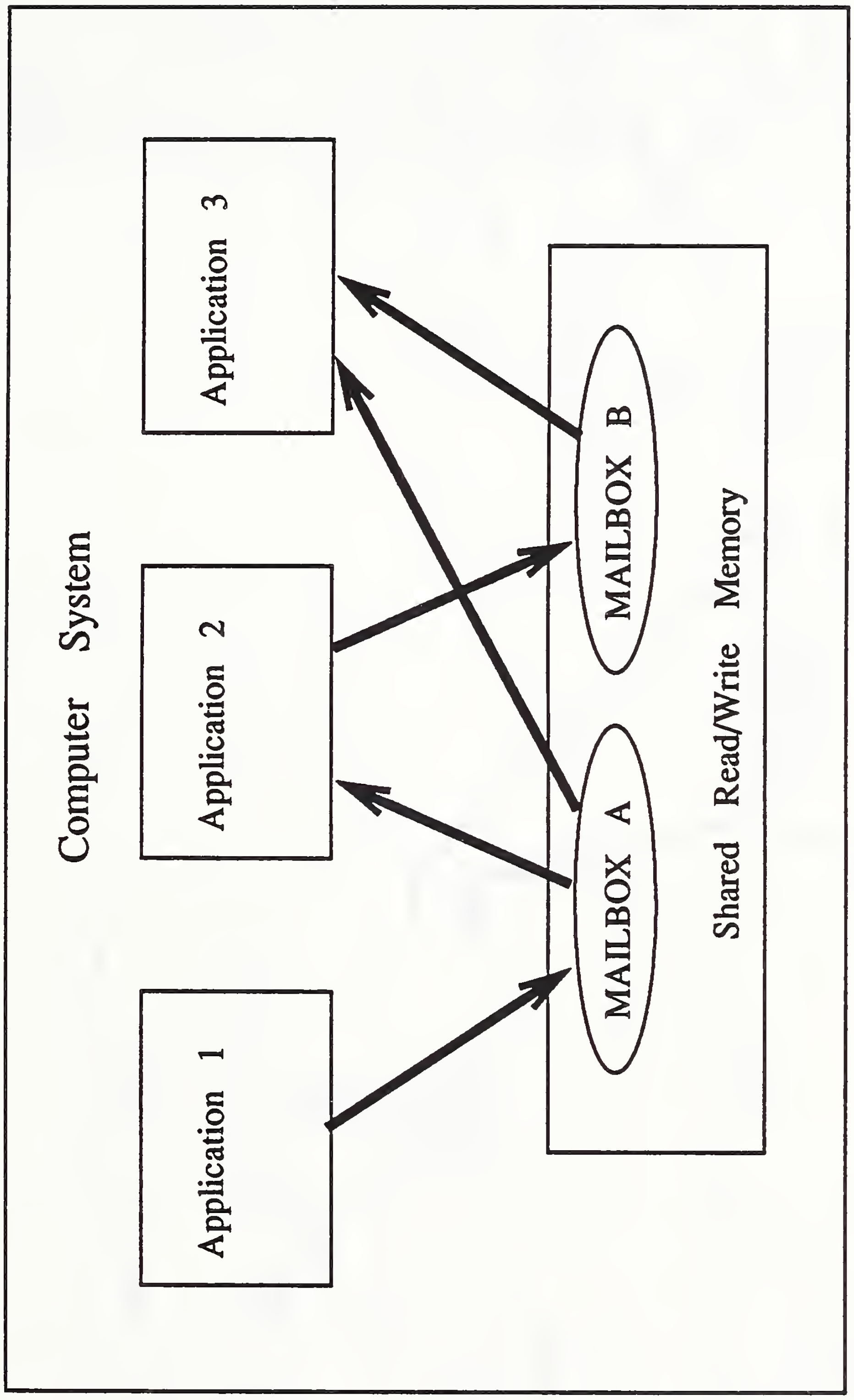

Figure I-1. Local Common Memory

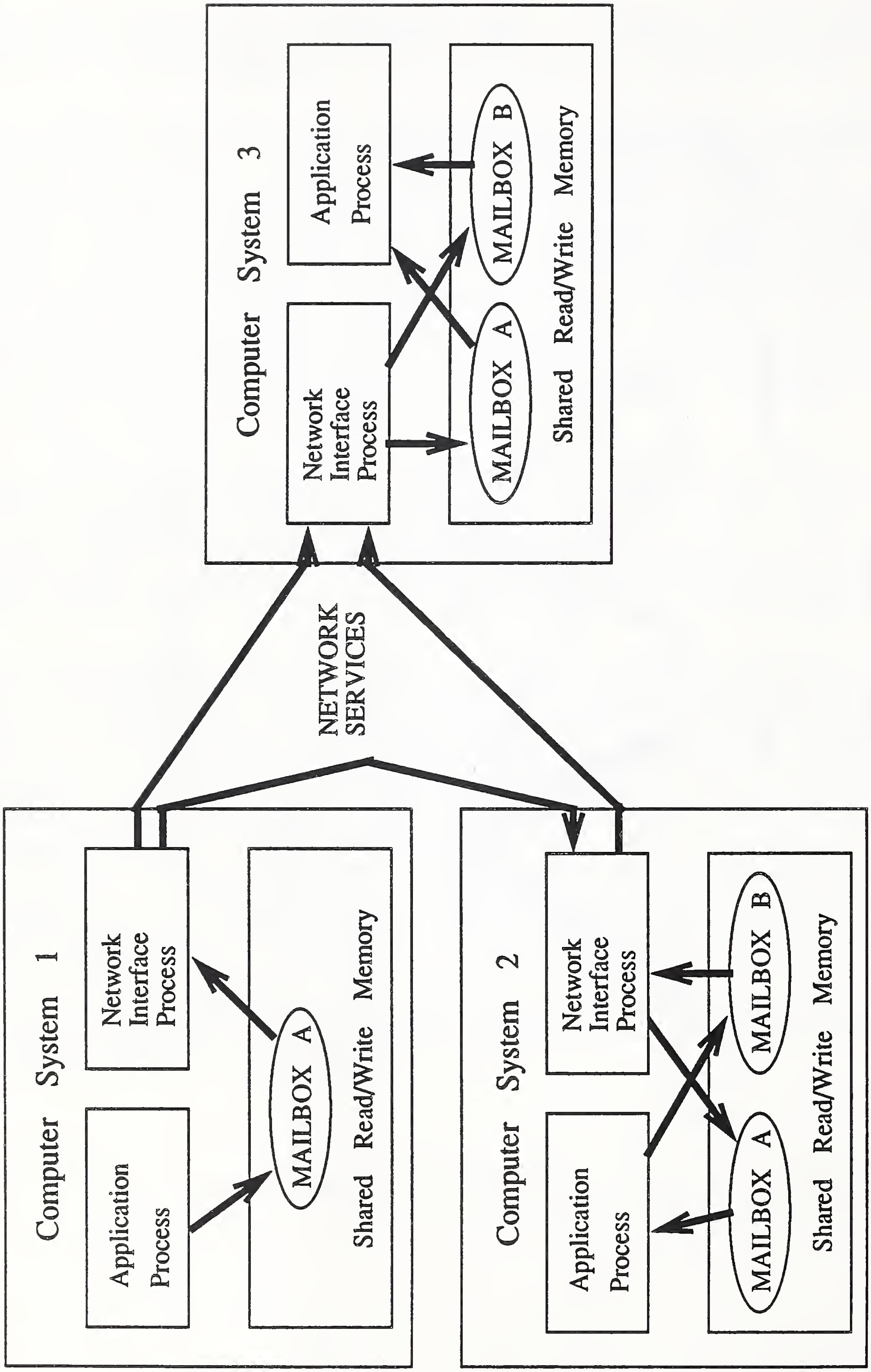

Figure I-2. Global Common Memory

3. PURPOSE OF THIS SOFTWARE COLLECTION

The intent of this software is to provide a local common memory environment for the IBM PC and compatibles. This generic group of personal computers will hereafter be referred to as PC's.

Given the increasing speed (e.g., 25 MHz 80386 personal computers) and capability (32 Mbytes of addressable space, multitasking operating systems, etc) of the PC and its rapidly decreasing price, it is feasible and desirable to consider the PC for implementation as a networked real-time controller. Providing the common memory for the PC is the first step in this process.

4. COMPUTER CONFIGURATION REQUIREMENTS

Computer configuration requirements are:

1. An IBM PC (or compatible) with at least 256 Kbytes of memory and DOS operating system.

2. The C programming language. Although the Turbo C (version 1.0) distribution was used during PC common memory development, Turbo C dependencies were studiously avoided -- with one exception, identified in Section IV.1.4. The use of C enabled the creation of object module libraries that can be used during the LINK process to incorporate common memory into user processes written in languages such as assembler, Prolog, FORTRAN, C and others.

II. OVERVIEW OF THE COMMON MEMORY ARCHITECTURE

The AMRF architecture of common memory and network communications is fully described in [7]. The following subsections discuss components of that architecture that are important to the understanding of PC common memory.

1. COMMON MEMORY COMPONENTS

1.1. Mailboxes and Mailgrams

All interprocess communication is accomplished through a mechanism called "mailboxes". Mailboxes are logical storage areas where messages (called "mailgrams") are placed by sender processes and picked up by receiver processes. From the point of view of the sender and receiver processes, the location of the correspondents does not affect their communication.

Mailboxes reside in a special area of memory, designated as local common memory. A common memory manager is responsible for assigning and managing the local common memory area.

Local common memories can be combined into a global common memory by implementing a unique common memory client process that interfaces with another similar process on a remote host for the purpose of exchanging common memory information. The physical connections can be point-to-point or utilize various local area network media, such as Ethernet or broadband. (Figure I-2.)

The user interface to local common memory is discussed below.

1.1.1. Mailgram Format

By convention, mailgrams are placed into a mailbox beginning at the lowest memory location allocated for the mailbox and continue occupying bytes until either the entire mailgram is in the mailbox or until all space allocated for the mailbox has been filled. Except for adherence to this convention, there is no standard format for a mailgram. There are, however, standard information units which must be associated with each mailbox. They are:

1. Length of the current mailgram in the mailbox,
2. Indication of a change in mailbox contents (discussed in Section II.2.3.), and
3. Some type of mailbox access control mechanism (discussed in Section II.2.)

The current level of the AMRF common memory architecture requires that these information units must be present either in the

```
Byte  1      2      3          . . .
  ----------------------------
  |      |      |      |
  |  Process-Dependent Text  . . .
  |      |      |      |
  ----------------------------
```

Figure II-1. Generic Mailbox Structure

mailgram or be maintained with the mailbox by the common memory manager. In general, if there is a common memory manager available in a particular common memory implementation, then the mailbox structure is shown in Figure II-1.

However, when no common memory manager is present and these entities are expressed in the mailbox area itself, then the mailbox has the structure shown in Figure II-2.

```
Byte  1       2       3       4       5       6       7       8
  -----------------------------------------------------------------
  ¦       ¦       ¦       ¦       ¦       ¦       ¦       ¦       ¦
  ¦  Write Lock   ¦  Read Lock    ¦   Sequence    ¦    Length     ¦
  ¦       ¦       ¦       ¦       ¦       ¦       ¦       ¦       ¦
  -----------------------------------------------------------------

          9       10      11      12     ...
         --------------------------------------
         ¦       ¦       ¦       ¦       ¦
         ¦  Process-Dependent Text . . .
         ¦       ¦       ¦       ¦       ¦
         --------------------------------------
```

Write Lock   Is a semaphore indicating current writer activity. (i.e., if the write lock is ON, then the mailbox is being written, and should not be read)

Read Lock   Is a semaphore indicating current reader activity. (i.e., if the read lock is ON, then the mailbox is being read, and should not be updated)

Sequence   Is a sequence number attached to the mailgram in the particular mailbox. Every time the text of the mailgram is changed, the sequence number is incremented. The update can be detected by examining only the sequence field.

Length   Is the length of the mailgram in bytes.

Text   Is the information portion of the mailgram that is defined entirely by the communicating processes.

Figure II-2. Special Mailbox Structure Used When A Common Memory Manager Is Not Present

With reference to Figure II-2, the read and write locks are considered to be part of the mailbox and not the mailgram. That is, when the mailgram is read or written, the lock bytes are manipulated in order to assure the integrity of the mailbox access, but are not transferred to or from a user data storage area. Furthermore, if a network interface process is to transfer the mailgram to another network interface process located on a different computer, the lock bytes are not transported.

```
Byte  1     2     3     4     5     6     7       . . .
  -----------------------------------------------------
 |     |     |     |     |     |     |     |     |
 |  Sequence |  Length   | Process-Dependent Text . . .
 |     |     |     |     |     |     |     |     |
 -----------------------------------------------------
```

Sequence — Is a sequence number attached to the mailgram in the particular mailbox. Every time the text of the mailgram is changed, the sequence number is incremented so that the change can be detected by examining only the sequence field.

Length — Is the length of the mailgram in bytes.

Text — Is the information portion of the mailgram that is defined entirely by the communicating processes.

Figure II-3. General Mailgram Format Used In A Mixed Common Memory Environment

In a mixed global common memory environment, where some systems have a common memory manager and others do not, the mailgram is assigned the specific format depicted in Figure II-3. The system with the common memory manager receives a mailgram with the sequence number and the length byte included in what it considers to be "process-dependent information". It is the responsibility of the application reading from the mailbox to know the mailgram format being used.

1.1.2. Mailbox and Mailgram Properties

(1) The mailbox must be created (or declared) before mailgrams can be deposited into it or an attempt can be made to read the mailbox contents.

(2) Every mailbox has a unique global name. The name is assigned to the mailbox at the time it is created and uniquely identifies the mailbox to all processes participating in the global common memory. Remote systems desiring a copy of the mailbox contents must have a mailbox with the same name declared in their local common memory. Network services perform the mailgram transfers [7].

(3) Every mailbox contains an initial value assigned by the creator when it is created. In some systems, this is a standard value (e.g., all zeros); in other systems, this value defaults to the contents of memory at the time of creation.

(4) Every mailbox contains exactly one mailgram at any given time. A mailgram stays in the mailbox, no matter how often it is read, until a new mailgram arrives for that mailbox. The new mailgram replaces the old one on arrival, whether or not the old mailgram has ever been read.

(5) The mailbox writer decides when to replace the mailgram. This may be performed independent of external information or may be influenced by "flow control" factors. (Section II.2.)

(6) In general, only one process is authorized to write into a mailbox at a time. In the case where more than one process may write into a specific mailbox, implicit or explicit "flow control" must be implemented. (Section II.2.)

(7) Any number of reader processes can retrieve the current mailgram in a mailbox.

(8) Any reader process can pick up the same mailgram several times if the writer does not change it in the interim. Likewise, any reader process may miss several mailgrams if the writer changes the mailgram more often than the reader picks it up. When it is important to assure that a particular recipient has read the mailgram before a new one is issued, the writer and reader must agree to a "flow control" protocol. (Section II.2.)

The mailbox management mechanism guarantees that a new mailgram will be distinct from its predecessors. However, the mailbox management mechanism does not guarantee that any particular receiver will have picked up a mailgram before it is replaced. If it is necessary to assure that a particular receiver has read the mailgram before it is replaced, the sender and that receiver must agree to a protocol by which the sender refrains from replacing the mailgram until it has an indication that the receiver has read it.

(9) Every mailbox has a fixed size which is defined when the mailbox is created. There is no predefined maximum on mailbox size. There may be a maximum mailbox size for individual systems as a consequence of hardware or

software limitations. Any given mailbox must be large enough to contain the largest mailgram agreed upon between the sender and receiver(s).

(10) Mailgrams can be of variable length. Each mailgram has associated with it information on how long it is. A mailgram may never be longer than the mailbox in which it is placed. If necessary, the mailgram will be truncated before it is transferred into the destination mailbox by the common memory service routines.

1.2. Common Memory Manager

Depending on the capabilities of the host operating system within any computer (or single-board computer in a Multibus chassis), it is possible to have an active common memory manager agent. However, an active common memory manager is not absolutely required. A passive manager is also possible.

1.2.1. Active Common Memory Manager

An active common manager is one that actively controls mailbox access for each common memory client using either of the following methods.

(1) Pass a token among the clients. The token might be passed sequentially from client to client. Alternately, the token could be held by the common memory manager; when a client requests common memory access, he is given the token. The token is returned to the manager when the client has completed common memory access.

It is possible for the token to get lost. That is, if a client that has been issued the token never returns it (due to a program abort or unexpected endless loop), all other clients are prohibited from further access. In order to minimize the consequences of such a deadlock situation, a token regeneration scheme must then be developed to recognize when the token has been unequivocally lost and a new token must be generated.

(2) Present a single interface to all common memory clients: the manager's access interface. The manager then coordinates, internally, the individual accesses of the clients. This is somewhat similar to token passing, where the token only passes between the manager and a client, and never between two clients. However, it does give the manager the flexibility of coordinating more than one concurrent access.

1.2.2. Passive Common Memory Manager

In the case of a passive common memory manager, there is no token or single interface. Instead, there is an access convention that is adhered to by all common memory clients. For example, the robot controller referenced previously (Section I.1.1.) uses a scheme involving repetitive common memory access cycles [8]. In this instance, there are three distinct divisions of the cycle:

(1) READ division: During the READ division all processes (each of which exists on its own single-board computer) compete for bus access in order to READ from common memory mailboxes. No WRITE actions are performed during this time period. The duration of this division is fixed. All READ accesses that are not completed during this time period are postponed until the next READ division.

(2) PROCESS division: During this division, all data reduction, equipment control and data acquisition occurs. No READ or WRITE access to the mailboxes is performed. Its duration is fixed.

(3) WRITE division: During the WRITE division, all processes once again compete for bus access in order to WRITE to common memory mailboxes. NO READ actions are performed during this time period. The duration of this division is fixed. All WRITE accesses that are not completed during this time period are postponed until the next WRITE division.

2. COORDINATING COMMON MEMORY ACCESS

Common memory environments can be susceptible to several problems related to coordinating access to these areas. The following subsections identify the potential problems and some possible solutions. Section III of this document specifies the solutions implemented in the PC common memory architecture.

2.1. Read While Write is Active

A process may be attempting to read information from a common memory mailbox at the same time that a second process is attempting to update that mailbox (or vice-versa). Consequently, the reading process may get inconsistent information (e.g., the current value of field A and the former value of field B).

To avoid this, one could:

(1) use a semaphore for each common memory buffer area (a mechanism that supports single-process access to the buffer). Some processors provide atomic "test and set" operations which can be used as hardware semaphores. Unfortunately, the PC does not. Software semaphores, using Dekker's algorithm [4], for example, can be extended to provide mutual exclusion between any number of processes.

(2) define a regular, recurring real-time interval and divide it into a write-only period and a read-only period. Any process not prepared to perform a write operation during the write-only period would have to wait for the next write-only period. The same restriction holds for read-only periods.

(3) pass a token among participating processes. The process that has the token can perform any read or write operation it wants. Fixed length or varying length time quanta can be employed. Token passing has an unfortunate drawback: if the process with the token halts (or appears to do so), passing of the token becomes impossible and all access to common memory is barred. In a fixed length time quantum implementation, the token can be reissued by some governing process after the expiration of the time quantum (plus some extra "safety margin"); in a varying length time quantum implementation, the recovery algorithm is much less obvious.

(4) utilize a hardware architecture that does not support interrupt processing. Once a processor has control of the bus (and consequent access to common memory), no other processor can interrupt. This assures that overlapped access does not occur.

2.2. Update Frequency Exceeds Read Frequency

A process may update the common memory area more often than a reading process is able to retrieve the information.

This may only be an "application-specific" problem. That is, if the reader process only wants the "current" information (as from a temperature sensor, for example), then the fact that any amount of older information may have been missed is a moot point. However, if it becomes important that the reader process have access to each information set before it gets updated, then some form of "flow-control" must be used.

For example, if the information set in a common memory mailbox includes a unique identifier (a time stamp or sequence number), then flow control could be implemented by defining a second mailbox in the common memory area into which the reader process could echo the unique identifier. When the writer of the original information sees the echoed identifier in this second common memory mailbox, it knows that it can proceed with the next update.

This method of flow control is feasible when the reader process can consume the information as fast as it is produced. However, if the reader process is too slow, it can have negative ramifications for the information writer. For example, if a temperature sensor at a nuclear power plant is attempting to report rapidly rising temperatures but is prohibited from reporting the current temperature because the mailgram reader has not acknowledged the previous temperature, undesirable side-effects can result.

2.3. Read Frequency Exceeds Update Frequency

A process may read the data in the common memory area more often than a writer process updates it. This can result in "old" information unintentionally being considered "new" information.

In the case where the information happens to be a command such as "hit nail on head with hammer", an undesirable number of duplicate executions could be performed.

A possible solution is to identify new information whenever it is placed into the common memory buffer by implementing a flag field within the mailbox. This flag field could take the form of a sequence number that gets incremented with each update of the mailbox or a time stamp that identifies when the information was placed into the mailbox. In each case, the reader process is looking for a change in the flag field to indicate that mailbox contents have been updated.

An alternative method is for the common memory manager to maintain a list containing the names of mailboxes that have been updated. A separate list is maintained for each common memory application. The application can then be given the update list upon request. By reading only the changed mailboxes, the application can minimize unproductive time spent examining unchanged mailboxes. However, this method is only available for systems with an active common memory manager.

2.4. Multiple Readers of a Common Mailbox

In the case where a single mailbox is being accessed by multiple readers, if it is important that each of the readers have the opportunity to retrieve the mailgram before it is overwritten, then a more elaborate form of flow control must be implemented.

One solution is to share a single "flow control" mailbox between all the readers. Each reader sets a specific "flag" in the mailbox indicating he has retrieved the message. When all flags have been set, the shared-read mailbox contents can be overwritten. This "solution" immediately introduces another problem: multiple writers to a single mailbox. (See Section II.2.5.)

A simpler, more reliable solution is to assign each reader process its own flow control mailbox.

2.5. Multiple Writers to a Common Mailbox

Unpredictable results can occur when more than one process is permitted to write into a single common memory buffer:

(1) predicting the sequence in which information is written to the common memory buffer may be impossible,

(2) guaranteeing that all reader clients have seen the contents of the common memory buffer before it is updated may be impossible, and

(3) identifying the intended reader client audience for any particular memory buffer update may be impossible.

A simple solution is to stipulate that any common memory buffer is permitted to have only a single process writing data into it, although it can have any number of reader clients.

More complex solutions that support the use of a single common memory buffer by more than one writing process are possible. In general, these solutions require the implementation of enhanced flow control and flag field techniques.

3. MAILBOX MANAGEMENT

Mailbox management, as discussed in the following subsections, applies only to implementations with an active common memory manager. There are four fundamental common memory functions used for mailbox management. They are: DECLARE, UNDECLARE, READ and WRITE. Other AMRF extensions exist and are discussed in [7].

Passive common memory management involves the static designation of specific memory areas for each mailbox. Although this might be considered a DECLARE action, there is no equivalent UNDECLARE action. Likewise, READ and WRITE access is uniquely different from passively managed common memories (Section II.1.2.2.).

3.1. DECLARE Access Requirements for Each Mailbox

Before an application can utilize a common memory mailbox for a read or write operation, it must DECLARE the mailbox to the common memory manager. Without this declaration, the common memory manager will not allow access to the mailbox.

A declaration must be issued for each mailbox that is to be accessed and must specify if the mailbox declaration is for a READ function or for a WRITE function. If the mailbox does not already exist in common memory, it is created and space is allocated dynamically. The user application can declare the same mailbox more than once.

There is no logical limit to the number of mailboxes that any application may declare or access. Available memory and/or memory addressing constraints impose the only limitation on the number of common memory participants (applications), number of mailboxes, and mailbox size.

3.2. Perform the READ or WRITE Action

Using the common memory READ and WRITE functions, an application can access the mailbox as often as desired. However, the application must previously have declared that mailbox for the respective access. The common memory manager will return a fatal error status indication if the application is not a client of the mailbox for the requested access.

3.3. UNDECLARE Mailbox Access

After an application has completed all desired accesses to a mailbox and before the application terminates, it should undeclare all previously-declared mailboxes.

When a user application undeclares a mailbox, the common memory manager removes that application from the client list of the mailbox. If the mailbox has other clients, no further action is taken. However, if the mailbox has no other clients, it is removed from common memory and the space it occupied is freed.

4. MAILBOX ACCESS METHODS

There are two possible mailbox access methods: implicit and explicit.

4.1. Explicit Mailbox Access

The word "explicit" is used to imply that there is no common memory manager present. Predesignated, static memory areas are assigned mailbox functions and are accessed directly by more than one application for the purpose of exchanging information.

Since there is no memory manager agent, mailboxes and their starting address and size are static and designated manually by a human agent. The mailbox specifications are loaded (or coded) into each participating application. Coordinating mailbox access (Section II.2.) is not a problem as long as interrupts are disabled when any application accesses a mailbox.

This method of common memory access can result if:

(1) the participating processes are actually different states within a single program,

(2) the host operating system does not enforce a memory protection scheme whereby a process is prohibited from accessing memory allocated to a second process,

(3) the host operating system supports the declaration of "common" memory regions that can be shared by multiple applications, or

(4) a Multibus implementation with multiple single-board computers (SBC) is used together with a separate memory board that maps into the address space of each SBC.

The mailgram format for explicit common memory would tend to approximate Figure II-2 in order to clearly and easily identify new information and coordinate mailbox access.

4.2. Implicit Mailbox Access

When the implicit method is used, each process associates an internal "logical unit number" (or numeric handle) with a common memory mailbox. This logical unit number is supplied to the process when it creates the mailbox through the respective common memory service. The process then references the logical unit number when it performs READ or WRITE operations in order to exchange mailgrams with the common memory.

The memory location of the mailbox is never accessed directly by any of the participating processes. Instead, the common memory manager has the responsibility of transferring data between the mailbox and the user data area. This activity is performed whenever the application requests a READ or WRITE.

4.3. Conversion of Implicit to Explicit Access and Explicit to Implicit Access

Programs designed for explicit access can be moved to an implicit access environment by inserting the necessary CM_READs, CM_WRITEs, and CM_DECLAREs to create the mailbox. CM_UNDECLAREs are recommended to discontinue mailboxes after their usefulness has expired.

Likewise, programs designed for an implicit mailbox access environment can be moved to an explicit environment by removing the CM_DECLARE, CM_READ, CM_WRITE and CM_UNDECLARE sections and replacing them with the necessary code to identify and access target mailbox memory areas.

5. EXISTING COMMON MEMORY IMPLEMENTATIONS

5.1. Common Memory Maps into the Process's Address Space

In order to reduce the time needed to access areas of common memory, the most desirable implementation is one where the common memory occupies memory in the addressable range of the process. Additional processes within the same "computer" can have access to this same common memory area as long as they have a means of directly accessing that same address space.

The word "computer" is placed in quotes in the preceding paragraph because the reference can be to a single computer such as the Digital Equipment Corporation VAX. It can also refer to a collection of single-board computers resident in a single Multibus chassis.

Within the VAX, the common memory areas can be included in memory space of multiple processes, all active concurrently, by linking to them as a shared READ or WRITE memory area.

For Multibus systems, the local common memory maps into the address space of each of the single-board computers sharing the same bus. Each process within its respective single-board computer sees that memory as its own, and is able to access it directly for READ or WRITE purposes.

5.2. Common Memory as a Separate Process

Some multitasking computer systems used within the AMRF are not immediately amenable to sharing memory space with other active processes. By altering the operating system, it is possible to make them amenable. However, it is actually easier (and safer) to create a separate common memory task.

The common memory information is then transferred between a subset of information maintained by the (user) application process and the actual common memory maintained by this separate task. The interprocess communication is performed using Transmission Control Protocol (TCP) routines [9].

5.3. The Global Common Memory

Information is exchanged with other common memory systems implemented on remote computer hosts via another process resident on the local host, called the network interface process (NIP) [7]. The NIPs have access to all of their local common memory. Connected to each other over a network, NIPs are able to transfer information between common memories. (Figure I-2)

Except for the time delays associated with the transfer of the information across the network, the processes accessing the common memory have no knowledge of what is actually happening to the information that they provide or access.

6. SUMMARY

6.1. Common Memory - An Application Interface

The result of this implemented architecture is that common memory is an application interface to communications with any other processes, both local and remote. It provides a uniform and portable interface for every application. If the application is later moved to a new location, no code changes need to be made for any of its correspondents in order to continue data exchanges. Some changes to the moved application may be necessary if the new location provides a different hardware or operating system architecture. The location changes are reported to the network service and the network service adjusts the mailgram delivery paths [7].

Providing a single application interface allows the application developers to concentrate on the application and frees them from the dependencies of the host-dependent interprocess communication, including network communication.

Further benefits of the common memory interface are listed in the following subsection. This is followed by a short discussion of some perceived drawbacks to common memory as an application interface.

6.1.1. Benefits of Common Memory

The benefits that common memory provides are listed below. Only benefit 3 relies on the availability of network services. However, all benefits are enhanced by the availability and use of network services.

(1) Asynchronous communications occur between processes. The application process is not interrupted by communications from other processes. It accesses the desired information whenever necessary (i.e., whenever it is ready for it).

(2) Information can be shared with additional processes with a minimum of effort. Additional processes can read from the same areas of common memory without any action on the part of the initial information provider to deliver it.

(3) Communication is independent of the location of related processes. The application process does not need to know the location of any other process with which it communicates. If the second process is within the same processor, it is directly connected to the same common

memory. If the second process is located remotely, it has its own common memory with which it communicates. Network services provide the connectivity between common memories.

(4) It supports coordinated activity between independent processes. Two processes can coordinate their activities by using common memory for command/status information, independent of their respective locations.

(5) It supports independent evolution of individual processes. With the structured interface between processes that common memory provides, individual processes may evolve in response to changing requirements without mandating equivalent changes in other processes interconnected through the common memory (including the network interface process.)

(6) It provides a consistent communications methodology for a diverse collection of computers and operating systems. Application processes are freed from machine-dependent communication primitives (e.g., subroutine calls) for both interprocess communications and network communications.

6.1.2. Drawbacks to Common Memory

No paper has yet been published discussing the drawbacks of common memory although one is in preparation [10]. The following are a few thoughts based upon personal experience and do not represent any consensus of opinion.

(1) The reader of a mailgram does not know the state of the writer of that mailgram:

(a) Is the writer on a local or remote host?
(b) Is the writer still active or has it terminated or been aborted?
(c) Is the logical network connection linking the applications, if applicable, still established?

(2) The writer of a mailgram does not know the state of the reader of that mailgram:

(a) Is the reader on a local or remote host?
(b) Is the reader still active or has it terminated or been aborted?
(c) Is the logical network connection linking the writer with the reader, if applicable, still established?
(d) Has the reader retrieved the mailgram yet?

(3) Why use common memory at all if the correspondents are all known to each other and the number of correspondents and their location will never change? Other communications techniques, such as network message passing, would be more efficient.

With the exception of item (3), these concerns are associated with the delivery and receipt of mailgrams. To some extent, they are answerable with the implementation of a mailbox to acknowledge the receipt of message similar to the flow control mailboxes discussed in Section II.2. However, they actually extend beyond that level of concern.

For example, if a control process is waiting for a data report that originates from a sensory process at irregular time intervals, it is important for the control process to know whether the sensory process is ever going to deliver the next data report. Simply knowing that the sensory process has not halted or been aborted may not be enough: it may be stuck in an unintentional/undesirable endless loop! Perhaps it is necessary for the sensory process to provide a "heartbeat" status in a mailbox?

On the other hand, a sensory process that reports status into a common memory mailbox may not have been programmed to care whether the deposited information is ever read.

This list of drawbacks identifies part of the next logical evolution (or extension) of the common memory architecture. Each drawback is only a minor obstacle that can easily be overcome by having the necessary information provided in common memory, either by the common memory manager or by one or more of the participating processes.

Item (3) raises a good question. The value of common memory as an application interface in a manufacturing or production environment where configuration changes are extremely infrequent has yet to be determined. However, in a research environment where applications evolve and shift from one host system to another and from one architecture to another, the flexibility of common memory has proven invaluable, for all the reasons listed in Section II.6.1.1. Some alternatives to common memory are identified in the following section.

6.2. <u>Alternatives to Common Memory</u>

Simple alternatives that encompass a single computer and networking architecture are easily identified and implemented. Significant difficulty arises when dissimilar architectures comprise the applications environment. In those environments, the user process is responsible for providing for all communications mechanisms, including gateway routing to devices connected to dissimilar networks.

The variety of hardware and software systems implemented in the AMRF preclude the use of any simple common solution to provide the same service as common memory. Communications among processes within a single computer system would have to use facilities provided by the operating system (if any), or new capabilities similar to those provided by common memory would have to be developed.

Attempts to use services provided by commercially-available network solutions would also be difficult. Commercial network solutions are not available for all computer systems in use in the AMRF. Although more applications (solutions) using the TCP/IP protocol are becoming available all the time, the current migration for networks in manufacturing environments is towards the Manufacturing Automation Protocol (MAP) and the Technical and Office Protocol (TOP). With network companies concentrating on major computer systems at this time, MAP and TOP products are not available for all computer systems.

Any alternative to common memory that must provide service to multiple applications located on several computer systems distributed across multiple network topologies is most likely a connection-oriented message passing solution. For example, although TCP/IP is connectionless, MAP and TOP are connection oriented.

In a connection-oriented network, application process 'A' establishes and maintains a connection to application process 'B' for the bi-directional exchange of messages. Any new application introduced into this environment will have to be accommodated through code changes in those applications with which it must communicate to provide for the additional connection and messages.

III. PC COMMON MEMORY ARCHITECTURE DESCRIPTION

This section describes the common memory architecture as it is implemented for the personal computer using the DOS operating system. Hardware and operating system limitations were the most influential factors affecting the architecture's development. Extensive effort was made to avoid operating system and hardware dependencies in order to maximize portability to other computer architectures and operating systems.

The presentation of specifications in the following sections assumes that the reader has previously become familiar with Section II and the general common memory architecture.

1. MAILBOXES AND MAILGRAMS

1.1 Coordinating Common Memory Access

This topic was originally discussed in Section II.2. The following subsections identify the solutions specifically adopted for the PC common memory without further reference to the problems or alternative solutions.

1.1.1. Read While Write is Active

Coordinating multiple accesses to a common memory mailbox in the PC architecture approximates token passing. DOS is a single-user operating system, so the common memory code is incorporated as "in-line" code to the user application. Likewise, if the user wishes to include other capabilities, such as a network interface program, they too must be included as inline code. (An example of this is shown in Appendix D.) This results in a single program or application when viewed from the DOS perspective.

This single application will not interrupt itself to access common memory. As incorporated in the preceding paragraph, neither will the network interface process interrupt either itself or the user application. Effectively, when any section of the program accesses the common memory, it can be considered to "have the token". The token can only be lost as a consequence of a program crash; appropriate recovery and application debugging steps must be taken following abnormal program terminations.

Other methods (i.e., local software development as well as possible commercial offerings) of implementing multitasking within DOS were considered. They were avoided because they were operating system specific and significantly compromised the portability of the common memory application to other hardware or operating systems.

1.1.2. Update Frequency Exceeds Read Frequency

The PC common memory takes a totally passive role in implementing flow control. The decision to avoid flow control by the common memory manager is intentional: common memory should function just like computer memory. An application can access a mailbox for read or write purposes as often as it desires. The READ operation will retrieve whatever was deposited last, all prior contents having been overwritten.

If flow control is important to processes exchanging information, then it is the responsibility of the respective application processes to provide for it. (Refer to Sections II.2.2 and II.2.5. for further discussion about flow control.)

1.1.3. Read Frequency Exceeds Update Frequency

The PC common memory manager maintains a separate linked list of updated mailboxes for each application process that is a client of common memory. The application process can then be given the update list upon request, thereby providing an indication of mailgram update without the overhead of a mailgram transfer. By reading only the changed mailboxes, the application processes minimize unproductive time spent examining unchanged mailboxes.

1.1.4. Multiple Readers of a Common Mailbox

As stated in Section III.1.1.2, the PC common memory takes a totally passive role regarding flow control. It is the responsibility of the application processes to coordinate the mailgram update procedure if it is desirable that each application have the opportunity to retrieve a mailgram before it is updated.

1.1.5. Multiple Writers to a Common Mailbox

The PC common memory allows more than one application to write to a mailbox. If only one application is to have access to a mailbox for WRITE access, then that application must request exclusive WRITE access when declaring the mailbox.

One potential future extension to PC common memory is to provide for an "access list": the application originally declaring the mailbox is considered to "own" it and may grant other applications READ/WRITE access to its mailbox.

```
Byte  1      2      3          . . .
  ---------------------------
  |      |      |      |
  |  Process-Dependent Text  . . .
  |      |      |      |
  ---------------------------
```

Figure III-1. Generic PC Mailbox Structure

Coordinating writer access in order to avoid a mailgram deposited by one writer from being overwritten by another writer before it has been retrieved by any reader is the responsibility of the user application. The PC common memory takes a totally passive role regarding mailbox flow control.

1.2. Mailgram Format

The PC common memory architecture assumes a mailbox structure as shown in Figure III-1.

However, since PC application processes may need to communicate with other applications resident on systems without a common memory manager, the assumed mailbox structure is as shown in Figure III-2. Since the PC common memory manager does not know which mailgram format is in use, it does not manipulate or monitor any of the fields of this alternate mailgram format.

It is the responsibility of the user application reading from the mailbox to be knowledgeable about the mailgram format and the mailgram contents. Furthermore, it is the responsibility of the user application to update and/or provide the information in the SEQUENCE and LENGTH fields in accordance with AMRF conventions. That is, the sequence number must change each time the mailgram is intended to be considered "new". Writing a mailgram with an unchanged sequence number but different process-dependent text may result in the mailgram not being read by an application on a host system that does not have an active common memory manager.

The LENGTH field specifies the number of bytes in the process-dependent section of this mailgram format. A process performing a READ request with PC common memory is returned the total length of the mailgram, including the SEQUENCE and LENGTH field sizes. This number is expected to be different from the value of the LENGTH field in the mailgram.

```
Byte  1       2       3       4       5       6       7       . . .
  -----------------------------------------------------------
  |       |       |       |       |       |       |       |       |
  |   Sequence    |    Length     | Process-dependent Text . . .
  |       |       |       |       |       |       |       |       |
  -----------------------------------------------------------
```

Sequence — is a sequence number attached to the mailgram in the particular mailbox. Every time the text of the mailgram is changed, the sequence number is incremented, so that the change can be detected by examining only the sequence field. The numeric representation is unsigned binary integer, and "wraps" back to zero when the maximum integer representation is incremented.

Length — is the length of the process-dependent text in the mailgram in bytes. The numeric representation is unsigned binary integer.

Text — is the information portion of the mailgram, defined entirely by the communicating processes.

Figure III-2. Underlying PC Mailgram Format

2. COMMON MEMORY ACCESS METHOD

The PC common memory is implemented using the implicit common memory access method. All mailbox access is performed indirectly through calls to common memory manager routines.

3. MAILBOX MANAGEMENT

The following subsections describe the software interfaces that support explicit common memory, as implemented on the PC. There are four fundamental common memory functions: DECLARE, UNDECLARE, READ and WRITE. Two additional functions, CKMAIL and DISC, have been provided in the PC version of common memory in order to expedite memory access and management, and are also discussed.

3.1. DECLARE Access Requirements for Each Mailbox

Before an application can utilize a common memory mailbox for a READ or WRITE operation, it must DECLARE the mailbox to the common memory manager. Without this declaration, the common memory manager will not allow access to the mailbox.

A declaration must be issued for each mailbox that is to be accessed. However, the application may issue multiple types of access for the declared mailbox within the same mailbox declaration (Appendix B).

If the mailbox does not already exist in common memory, it is created and space is allocated dynamically. The common memory manager maintains two separate lists of clients for each mailbox: a list of reader clients and a list of writer clients. If access is granted, the user application is registered on the appropriate list in accordance with the requested access.

The user application can declare a mailbox more than once. Subsequent declarations for the same mailbox are assumed to be requests to change the application's access rights. That is, these requests can ADD an access that was not previously requested, or CHANGE an existing access. A particular access can only be removed by undeclaring it.

For example, a "XREAD | WRITE" declaration for a mailbox followed by an "READ" declaration results in an access of "READ | WRITE". The character "|" is used to represent a bit-wise OR of the bit representations associated with the specific access function. (The result assumes that a non-fatal error status is returned from function cm_declare).

There is no logical limit to the number of mailboxes that any application may declare or access. Available memory and/or memory addressing constraints impose the only limitation on the number of common memory participants (applications) and mailboxes.

3.2. Perform the READ or WRITE Action

Using the common memory READ and WRITE functions, an application can access the mailbox as often as desired. However, the application must previously have declared that mailbox for the respective access. The common memory manager will return a fatal error status if the application is not on the client list of the mailbox or has not been granted the requested (READ or WRITE) access.

3.3. UNDECLARE Mailbox Access

After an application has completed all desired accesses to a mailbox and before the application terminates, it should undeclare all previously-declared mailboxes. This can be done individually for each mailbox, or all mailboxes can be undeclared through a DISConnect request.

When a user application undeclares a mailbox, the common memory manager removes that application from the client list of the mailbox. If the mailbox has other clients, no further action is taken. However, if the mailbox has no other clients, it is removed from common memory and the space it occupied is freed.

3.4. Mailbox Management Extensions

The PC common memory architecture provides two notable extensions to the standard mailbox management services. These are provided in order to meet mailbox access coordination criteria and to simplify the mailbox management process.

3.4.1. DISConnect the Application from Common Memory

If an application intends to terminate its involvement in the common memory, it should undeclare all of its mailboxes. The DISConnect function provided by the common memory manager reduces this to a single request. When an application issues a disconnect request, the common memory manager undeclares all mailboxes for that application.

This function is provided in order to simplify and expedite the departure of an application from the common memory environment. Well-behaved applications can be expected to undeclare their mailboxes either individually or via the DISConnect function. However, in some future multitasking environment, it is conceivable that some task participating in common memory may abruptly terminate. It will be the responsibility of the common memory manager to detect the abnormal termination of the process ("how" will depend on the specific operating system) and undeclare all of that process's mailboxes, thereby purging the common memory of unnecessary mailboxes (i.e., mailboxes to which no other process has declared access).

3.4.2. Check for Mail

Since common memory mailboxes serve as a communications interface, it is important to know when (or whether) new information has been deposited in any mailbox. As mentioned previously (Section III.1.1.3.), a mechanism exists within the PC common memory architecture to identify new information.

Rather than force every application participating in the common memory to examine each of its mailboxes to discover new data, the PC common memory manager maintains a list of mailboxes that have been updated. The application can simply check its mail and is presented with a list of mailboxes that have changed. The common memory manager then starts a new list. The application can now limit its READ requests to those mailboxes whose contents have changed, and thereby minimize the amount of CPU time spent looking at common memory mailboxes.

If the application performs a READ of a mailbox for which there is an entry on the update list (i.e., without checking for mail first), the update list entry for that mailbox is removed by the common memory manager.

IV. PROGRAMMER REFERENCE

The common memory library was developed using the C programming language, as specified by Kernighan and Ritchie [1]. The C programming language was selected in order to maximize portability to other computer architectures.

1. IMPLEMENTATION ISSUES

1.1. The Compiler

The Turbo C (version 1.0) small memory model compiler was used [2,3]. Users who wish to use a different compiler model may need to recompile the common memory library in order to have a consistent data structure definition for variables such as pointers. Defaults are used in the common memory library in order to facilitate such redefinition.

1.2. Memory Allocation and Usage

The common memory manager uses linked lists extensively to maintain information about its clients and their associated mailboxes. The memory space for these data structures is allocated and freed dynamically using functions MALLOC and FREE, respectively. The MALLOC function allocates memory space directly from the user static data area. In the small model, the user application is given 64 Kbytes of static data space.

The 64 Kbyte limit is an artificial one, however, since data space outside of that 64 kbyte can be dynamically allocated and freed using functions FARMALLOC and FARFREE. However, these functions are specific to MS-DOS, and it is desirable to avoid such dependencies.

The amount of data space available to the application process varies with the compiler memory model used. The compact and large models, for example, provide for up to 1 Mbyte of static data space. If the user prefers to use the small memory model (perhaps because compiling is faster), it is preferable that the user application perform the FARMALLOC and FARFREE in order to continue the current level of common memory library independence.

1.3. Interfaces to Languages Other Than C

The common memory library was developed using the C programming language. This does not preclude software developers, who wish to use another programming language compiler for their application, from linking with the library.

For example, the Turbo C user's guide [2] discusses how to link Turbo Pascal and Prolog programs with C object modules. Likewise, it may be feasible to link this C version of the common memory library to other languages, such as Lisp and Ada. This will depend on the compiler implementation. The interested reader must research the respective language reference manual(s).

1.4. Non-portable Common Memory Functions

Only one function referenced in the library is not immediately portable to other C language compilers. This is function "eprintf". It is located in file cm_utils.c, and is used by the common memory functions to print debugging statements. Its specificity is to Turbo C and is based on its use of a variable-length argument list. However, other (but not necessarily all) C compilers are known to support variable-length argument lists, so recoding this function to comply with another compiler should not be too difficult.

2. THE COMMON MEMORY INTERFACE LIBRARY

The common memory library provides three service categories: (1) functions that provide common memory access, (2) functions that provide information about common memory usage, and (3) convenience functions. These service categories are described in detail below.

The library distribution kit consists of two INCLUDE (source) files and two object files. The user application must include the source files during the application program compilation period (described in Section IV.5.2). The object files are linked during the application program linking period (described in Section IV.5.3). Section IV.5.1. lists files in the distribution kit.

The source files and the object files have an embedded variable that identifies the common memory library version number. It is critical that both version numbers be identical in order to insure the proper performance of the common memory interface. Section IV.4.3. details how to determine the respective version numbers.

The common memory interface descriptions shown below incorporate the C language convention that a function returns a value through a RETURN statement. These returned values are of type SHORT INT (2 bytes) and report a completion status for each routine. Appendix A lists all possible status codes and describes their significance.

The library routines utilize various data structures to contain and convey specific information to the user application. In the following pages, the data structure for each argument of the functions is identified. This identification is prefaced by the word "TYPE". The data structures are detailed in Appendix C.

2.1. Introduction to Common Memory Manipulations

The actual functions that provide access to common memory are presented in subsequent pages. Before using these functions, it is necessary to have an understanding of the relationship they have to each other and the sequence in which they must be accessed. Section III.3 describes this relationship.

2.2. Functions That Provide Common Memory Access

This section describes the functions that provide common memory access. Only four of them (cm_declare, cm_undeclare, cm_write, and cm_read) are actually necessary. The others provide expeditious extensions to these basic functions. (For example, cm_disc will undeclare all mailboxes previously declared by the user process, thereby relieving the user from having to submit a series of cm_undeclare's.)

The argument list for each function is described in detail. Each function returns an integer status value that correlates with what the function is to perform (hence the 'int' before each function name). The list of all potential status values that this family of functions can return and their significance is provided in Appendix A.

2.2.1. Function cm_declare

This function creates the necessary application, mailbox, and mailbox client structures within common memory to support future mailbox manipulations by the declaring application.

If an application calls cm_declare for an existing mailbox for READ or XREAD access, the common memory manager will NOT place an entry into its "update list" for that mailbox. The purpose of the update list is to indicate that the mailbox contents have been written SINCE the time of the cm_declare or cm_read. It is assumed that the user application will perform an initial cm_read as a matter of course.

```
int cm_declare(fsm                - INPUT
                                    TYPE char *fsm
                                    user application name string.
                                    String must be null-terminated
                                    and must be less than or equal to
                                    32 characters in length
                                    (excluding the trailing NULL).

               mbxname            - INPUT
                                    TYPE char *mbxname
                                    mailbox name string.  String must
                                    be null-terminated and must be
                                    less than or equal to  32
                                    characters in length (excluding
                                    the trailing NULL).

               mbxsize            - INPUT
                                    TYPE int mbxsize
                                    max size of mailbox to be
                                    created.

               mbxaccess          - INPUT
                                    TYPE int mbxaccess
                                    Potential vales are
                                    READ | WRITE | XREAD | XWRITE
                                    but not both of the same kind in
                                    the same declaration.  The
                                    associated constants are
                                    listed in cm_const.h.

               mbxhandle          - OUTPUT
                                    TYPE int *mbxhandle
                                    Value returned in the integer
                                    variable is used as a shorthand
                                    reference for mailbox for calls
                                    to all other cm routines.
               )
     RETURNS: status, as identified in cm_const.h
```

2.2.2. Function cm_undeclare

Function cm_undeclare is used to remove a user application from a particular mailbox's client list for the specified access. More than one access type may be specified at each call, subject to the access rules identified in the cm_declare section.

If the undeclare action results in a mailbox without any clients, that mailbox is deleted and the space returned to the operating system. Likewise, if the action results in a user application that has no other mailbox's declared, that user application is removed as a common memory client.

If the deleted mailbox is referenced on the undeclaring user application's update list, that specific update entry will also be deleted.

```
int cm_undeclare(fsm              - INPUT
                                    TYPE char *fsm
                                    user application name string.
                                    String must be null-terminated
                                    and must be less than or equal to
                                    32 characters in length
                                    (excluding the trailing NULL).

                 mbxaccess        - INPUT
                                    TYPE int mbxaccess
                                    Potential values are
                                    READ | WRITE | XREAD | XWRITE,
                                    but not both of the same kind in
                                    the same declaration.  The
                                    access constants are
                                    listed in cm_const.h.

                 mbxhandle        - INPUT
                                    TYPE int *mbxhandle
                                    Variable value was initially set
                                    by cm_declare and is used as a
                                    fast way to reference a specific
                                    mailbox.  Although this does
                                    not need to be a pointer, it is
                                    specified as such for
                                    compatibility with cm_declare
                                    (which requires it) and other
                                    common memory routines.
                )
     RETURNS: status, as identified in cm_const.h
```

2.2.3. Function cm_write

Function cm_write is used to transfer a specified number of bytes from the user data area to the common memory mailbox. As a consequence of the write operation, all user application's that have declared READ (or XREAD) access to this mailbox will have an entry made on their update list.

```
int cm_write (fsm               - INPUT
                                  TYPE char *fsm
                                  user application name string.
                                  String must be null-terminated
                                  and must be less than or equal to
                                  32 characters in length
                                  (excluding the trailing NULL).

              mbxhandle         - INPUT
                                  TYPE int *mbxhandle
                                  Variable value was initially set
                                  cm_declare and is used as a fast
                                  way to reference a specific
                                  mailbox. Although this does not
                                  need to be a pointer, it is
                                  specified as such for
                                  compatibility with cm_declare
                                  (which requires it) and other
                                  common memory routines.

              usr_data          - INPUT
                                  TYPE byte *usr_data
                                  points to user data area from
                                  which bytes are transferred.

              nr_bytes          - INPUT
                                  TYPE int *nr_bytes
                                  int variable contains nr of bytes
                                  to be transferred from user data
                                  area to common memory. Although
                                  this does not need to be a
                                  pointer, it is specified as such
                                  for compatibility with cm_read,
                                  which requires it.
              )
    RETURNS: status, as identified in cm_const.h
```

2.2.4. Function cm_read

Function cm_read is used to transfer a specified number of bytes to the user data area from the common memory mailbox. It is recommended that cm_ckmail be used together with cm_read to minimize unnecessary common memory accesses.

If an entry for this mailbox exists on the update list of this user application, it is removed at completion of the cm_read operation.

```
int cm_read  (fsm              - INPUT
                                 TYPE char *fsm
                                 user application name string.
                                 String must be null-terminated,
                                 and must be less than or equal to
                                 32 characters in length
                                 (excluding the trailing NULL).

              mbxhandle        - INPUT
                                 TYPE int *mbxhandle
                                 Variable value was initially set
                                 cm_declare and is used as a fast
                                 way to reference a specific
                                 mailbox. Although this does not
                                 need to be a pointer, it is
                                 specified as such for
                                 compatibility with cm_declare
                                 (which requires it) and other
                                 common memory routines.

              usr_data         - INPUT
                                 TYPE byte *usr_data
                                 points to user data area from
                                 which bytes are to be
                                 transferred.

              nr_bytes         - INPUT/OUTPUT
                                 TYPE int *nr_bytes
                                 When cm_read is called, if :
                                    (1) the int variable = 0, then
                                        all data bytes are
                                        transferred from the
                                        mailbox to the user's data
                                        area.
                                    (2) the int variable is not
                                        equal to 0, the nr of
                                        bytes transferred will be
                                        the minimum of (nr_bytes,
                                        nr_bytes_in_mbx).
```

Upon return, the variable pointed to by nr_bytes will contain the actual number of bytes transferred. If fewer bytes are transferred to the user data area than are available in the mailbox, an "information- only" status of I_CM_MOREDATA is returned to alert the user who may have inadvertently called cm_read without clearing the variable pointed to by nr_bytes.

It is the user's responsibility to make sure that the data area is large enough to contain the mailgram.

)

RETURNS: status, as identified in cm_const.h

2.2.5. Function cm_ckmail

For each user application that is a READer client of common memory, the common memory manager creates and maintains a list of those mailboxes that have changed since the last time the user application read them (i.e., an update list). Whenever a user application writes to a common memory mailbox, the common memory manager posts an entry on this "update" list. If an entry already exists for a changed mailbox, no additional entry is made. The list is maintained in first-in-first-out (FIFO) order.

Entries are removed from this list when a user application calls cm_read for the respective mailbox or when the application calls cm_ckmail. If an update list exists, cm_ckmail returns a pointer to the top of the list and releases the list to the application. If no update list exists, cm_ckmail returns NULL. (If a list is passed to the user application, the common memory manager will start a new list when the next mailbox update arrives.)

Once the list is released to the user application, it is the responsibility of the user application to FREE the memory allocated for the list.

Using the update list, the user application can now perform sequential cm_read operations and only access those mailboxes that have changed since the last read operation.

int cm_ckmail (fsm	- INPUT TYPE char *fsm user application name string. String must be null-terminated and must be less than or equal to 32 characters in length (excluding the trailing NULL).
list_ptr	- INPUT TYPE is struct update_list **list_ptr; If an update list exists for this user application, cm_ckmail will return a ptr to the top of the update list in this location. If none exists, cm_ckmail will return NULL.

(continued on next page)

```
        nr_entries        - INPUT
                            TYPE  int *nr_entries;
                            If an update list exists, the int
                            variable will contain the number
                            of entries in the update list;
                            else, it will contain ZERO.
        )
```

RETURNS: status, as identified in cm_const.h

2.2.6. Function cm_disc

Function cm_disc provides a shortcut method for a user application to undeclare all of its mailboxes at one time. All data structures within common memory that are associated with that user application are freed. The user application must issue a cm_declare before it can again access common memory variables.

```
int cm_disc (fsm                    - INPUT
                                      TYPE char *fsm
                                      user application name string.
                                      String must be null-terminated
                                      and must be less than or equal to
                                      32 characters in length
                                      (excluding the trailing NULL).
           )
```

RETURNS: status, as identified in cm_const.h

2.3. Functions That Provide Information About Common Memory Usage

The following functions provide a "window" into the common memory environment. They are available to any application. In fact, applications that are not participating in common memory (i.e., applications that do not have any mailboxes declared) can use these calls to determine common memory activity.

These functions were intended for use during the common memory development process. During that time, it was determined that they would be useful for reporting local common memory status to some supervisory and/or monitoring agent (located either across the network or on the local host).

2.3.1. Function cm_get_fsm_list

This function allows any user application to determine what user applications are currently active in common memory. Using the mbxname and list_type function arguments appropriately, the caller can retrieve the list of all user applications known to the common memory manager or only the list of clients (read or write) for a specific mailbox. The information that is returned on the list can be used to solicit other user application (and, indirectly, mailbox) statistics.

```
int cm_get_fsm_list(
            mbxname          - INPUT
                               TYPE  char *mbxname
                               If NULL, this routine will
                               return, through fsm_list_ptr, the
                               list of all user application
                               names known to the common memory
                               manager.  Argument "list_type"
                               has no effect.  If not NULL, it
                               must point to a mailbox name.
                               This routine will return a list
                               of all user applications that are
                               a client of that specific
                               mailbox. The argument "list_type"
                               is used to qualify whether the
                               caller wants the list of READer
                               clients or the list of WRITEr
                               clients.

            list_type        - INPUT
                               TYPE  char
                               May only have the values 'R'
                               (READ) and 'W' (WRITE) when
                               *mbxname is non-NULL.  If
                               *mbxname is NULL, list_type is
                               ignored.

            fsm_list_ptr     - OUTPUT
                               TYPE  struct fsm_list_type
                                     **fsm_list_ptr
                               This routine will create a linked
                               list of user application names
                               and return a ptr to the top of
                               the list if any user applications
                               exists or NULL if none exist.  It
                               is the user's responsibility to
                               free this list when it is no
                               longer needed.
```

```
int_ptr             - OUTPUT
                      TYPE int *int_ptr
                      Upon return, the int variable
                      will equal the number of entries
                      in the list.
)
```

RETURNS: status, as identified in cm_const.h

2.3.2. Function cm_get_mbx_list

This function allows any user application to determine what mailboxes (mbx) are currently active in common memory. The information that is returned on the list can be used to solicit other common memory user application (and, indirectly, mailbox) statistics. Using the fsm and list_type function arguments appropriately, the caller can retrieve the list of all mailboxes in common memory, or only the list of mailboxes (read or write) for a specific user application.

```
int cm_get_mbx_list(
               fsm               -  INPUT
                                    TYPE  char *fsm
                                    user application name string.
                                    String must be null-terminated
                                    and must be less than or equal to
                                    32 characters in length
                                    (excluding the trailing NULL).
                                    If NULL, this routine will
                                    return, through mbx_list_ptr, the
                                    list of all mailbox names known
                                    to the common memory manager.
                                    Argument "list_type" has no
                                    effect.  If not NULL, it must
                                    point to a user application name.
                                    This routine will return a list
                                    of all mailboxes that are a
                                    declared by that user
                                    application.  The argument
                                    "list_type" is used to qualify
                                    whether the caller wants the list
                                    of READer mailboxes or the list
                                    of WRITEr mailboxes.

               list_type         -  INPUT
                                    TYPE  char
                                    May only have the values 'R'
                                    (READ) and 'W' (WRITE) when *fsm
                                    is non-NULL.  If *fsm is NULL,
                                    list_type is ignored.

               mbx_list_ptr      -  OUTPUT
                                    TYPE  struct mbx_list_type
                                          **mbx_list_ptr
                                    This routine will create a linked
                                    list of mailbox names and return
```

```
                                a ptr to the top of the list if
                                any mailboxes exist, or NULL if
                                none exist.  It is the user's
                                responsibility to free this list
                                when it is no longer needed.

         int_ptr              - INPUT
                                TYPE  int *int_ptr
                                Upon return, the int variable
                                will equal the number of entries
                                in the list.
         )
```

RETURNS: status, as identified in cm_const.h

2.3.3. Function cm_get_cm_stats

This function provides common memory operating statistics. Since the size of the statistics areas is static, the user application must provide a pointer to space in the user data area into which the statistics will be copied. This avoids the overhead associated with dynamic memory allocation in case the user application calls this routine multiple times.

```
int cm_get_cm_stats(
              activity_ptr    - INPUT
                                TYPE  cm_activity_stats
                                      *activity_ptr
                                Points to user-allocated data
                                area of appropriate size.  This
                                routine will copy the activity
                                statistics into that data area.
                                It has been implemented in this
                                fashion to minimize malloc and
                                free operations, since it is
                                assumed the user will want this
                                information more than once.

              client_ptr      - INPUT
                                TYPE  cm_client_stats
                                      *client_ptr;
                                Points to user-allocated data
                                area of appropriate size.  This
                                routine will copy the client
                                statistics into that data area.
                                It has been implemented in this
                                fashion to minimize malloc and
                                free operations, since it is
                                assumed the user will want this
                                information more than once.
              )

      RETURNS: status, as identified in cm_const.h
```

2.3.4. Function cm_get_fsm_stats

This function returns the common memory statistics for the specified user application, as identified in structure cm_fsm_stats_rec (Appendix C). Since the size of the statistics area is static, the user application must provide a pointer to space in the user data area into which the statistics will be copied. This avoids the overhead associated with dynamic memory allocation in case the user application calls this routine multiple times.

```
int cm_get_fsm_stats(
            fsm             - INPUT
                              TYPE  char *fsm
                              user application name string.
                              String must be null-terminated
                              and must be less than or equal to
                              32 characters in length
                              (excluding the trailing NULL).
            ptr             - INPUT
                              TYPE  cm_fsm_stats_rec *ptr
            )
     RETURNS: status, as identified in cm_const.h
```

2.3.5. Function cm_get_mbx_stats

This function returns the common memory statistics for the specified mailbox, as identified in structure cm_mbx_stats_rec (Appendix C). Since the size of the statistics area is static, the user application must provide a pointer to space in the user data area into which the statistics will be copied. This avoids the overhead associated with dynamic memory allocation in case the user application calls this routine multiple times.

```
int cm_get_mbx_stats(
            mbx               - INPUT
                                TYPE  char *mbx
                                mailbox name string.  String must
                                be null-terminated and must be
                                less than or equal to  32
                                characters in length (excluding
                                the trailing NULL).

            ptr               - INPUT
                                TYPE  cm_mbx_stats_rec *ptr
            )
      RETURNS: status, as identified in cm_const.h
```

3. CONVENIENCE FUNCTIONS

During the development of the PC common memory library, several utility functions were developed that might be useful to an application programmer developing a program that uses PC common memory. These functions provide information or functionality directly related to PC common memory usage and are documented below. (Other functions that might be more generally useful are documented in the source listing of file "sfuncs.c". However, they are not specific only to the use of PC common memory. The interested reader is encouraged to examine the "sfuncs.c" file.)

3.1. Function cm_free_update_list

This function frees the memory allocated to an update list. The update list is passed to the user by function cm_ckmail. It is the user's responsibility to free the memory allocated to that linked list. If the user does not free the memory associated with that linked list, it is conceivable that the application may run out of dynamically allocable memory and the common memory manager will be unable to function.

The user may free the blocks of the linked list with calls to the C function "free" or may use this function.

```
void cm_free_update_list(
              list              - INPUT
                                  TYPE struct update_list *list
                                  This is a pointer to the top of
                                  the update list.  The function
                                  will free each block of the
                                  structure, advancing to the next
                                  block, until NULL is reached.
              )
      RETURNS: nothing
```

3.2. Function cm_get_statusname

This function converts the status code from its numeric representation into a string containing the corresponding status mnemonic. The conversion is based on the status codes maintained in file cm_const.h and listed in Appendix A. All unrecognizable status code is converted into the string "unknown code".

```
char *cm_get_statusname (
             code              - INPUT
                                 TYPE int code
                                 This is presumed to be the status
                                 code returned by one of the
                                 common memory interface routines.
             )

      RETURNS: pointer to a string containing the status code
               associated with the code passed to it. If the
               code is not recognizable, the function returns
               the string "unknown code".
```

3.3. Function cm_ini

This function can be used to (1) perform common memory initialization at a known point in the user application program and (2) display the version of the common memory object library.

The user application does not have to call this function. Each common memory interface function, when it is called, checks to see if common memory has been initialized. If common memory has not been initialized, the interface function makes a call to cm_ini. Conversely, if common memory has been initialized, no call to cm_ini is made.

Function cm_ini is responsible for establishing the data structures for the use and management of common memory. During cm_ini execution, the value of global variable CM_DEBUG_LEVEL is examined. If the value of this variable is greater than zero, cm_ini will display the common memory object library version number and will leave the value of CM_DEBUG_LEVEL unchanged. If it is less than zero, cm_ini sets CM_DEBUG_LEVEL to zero to turn off common memory debugging statements.

The interested reader is referred to Section IV.4 for further discussions about global common memory variables CM_DEBUG_LEVEL, CM_GET_STATS, and CM_VERSION.

```
void cm_ini()

      RETURNS: nothing
```

4. GLOBAL COMMON MEMORY VARIABLES

The variables listed in the following subsections are declared by the common memory manager and made available to the user application program. By manipulating these variables, the application program can affect the operation of the local common memory manager. CM_VERSION, as the exception to the previous statement, has no control function. It is only used to provide information about the common memory library.

4.1. CM_DEBUG_LEVEL - Control the Amount of Common Memory Debugging Information Displayed

This variable is declared and initialized in the common memory object library and is located in file cm_globa.h. It is an integer variable whose assigned value determines the amount of diagnostic and/or debugging information the common memory manager will display at the user console. The debugging levels "build" upon each other. That is, selecting a debug level also selects those levels below it (i.e., those with a lower debug level number are also included). The current CM_DEBUG_LEVEL values and their effect are:

- 0 - no debugging data is displayed.
- 1 - display the common memory library version. (This value has been compiled into the object library and cannot be easily accessed or changed by the user application program. Section IV.4.3 provides more information.)
- 2 - 4 <reserved for future use>
- 5 - display identifying messages whenever a mailbox is written or read.
- 6 - display identifying messages whenever an application or mailbox entry is added or deleted to the list maintained by the common memory manager or whenever a cm_declare fails.
- 7 - display identifying messages whenever a mailbox client is added or deleted.
- 8 - display identifying messages whenever a mailbox update notification is posted or removed from the list maintained by the common memory manager.
- 9 - everything (includes 0-8 and more)

The value of CM_DEBUG_LEVEL is initially set to -1 in file cm_globa.h. If the application program changes the value to be greater than or equal to 0, then the common memory manger will not change it. Function cm_ini is responsible for initializing the value of CM_DEBUG_LEVEL to zero if it has a negative value when common memory is initialized.

The only time this variable is examined and (potentially) changed is in function cm_ini. This function is only called once by the common memory manager although the user application can call it as frequently as desired. Consequently, the user application can manipulate the value of CM_DEBUG_VALUE in order to change or halt the amount of common memory debugging information provided by the common memory manager.

If the user application intends to manipulate or monitor the value of CM_DEBUG_LEVEL, it must include the following line in the application program:

```
extern int CM_DEBUG_LEVEL;
```

4.2. CM_GET_STATS - Control the Acquisition of Common Memory Statistics

The PC common memory manager includes the capability of gathering common memory usage statistics. These statistics are available to any application that has access to the common memory manager.

Gathering these statistics takes CPU time away from the application process. If the host processor is slow and cannot provide the level of response necessary for the application, it may be necessary to analyze where the bottleneck is located. If it occurs at the interface between the application and common memory, turning OFF statistics gathering is one way to improve responsiveness. If it occurs within the user application, then changing the value associated with CM_GET_STATS will have no affect.

This variable is declared and initialized in the common memory object library, in file cm_globa.h, and is of type "boolean". Its initial value is set to TRUE to enable the gathering of statistics.

If the user application intends to manipulate or monitor the value of CM_GET_STATS, it must include the following line in the application program:

```
extern boolean CM_DEBUG_LEVEL;
```

In order to assure the proper definition of structure "boolean", this line should appear in the user application source code after the line containing:

```
INCLUDE "cm_types.h"
```

4.3. CM_VERSION - Determine Version Numbers of the Common Memory Distribution Components

The PC common memory distribution kit contents are listed in Section IV.5.1. Without some sort of tag or label, it becomes a nearly impossible task to make sure that the components of the distribution kit are all at the same version number.

No matter how much care is exerted, it is always feasible that the common memory version of the two INCLUDE files can get out of synchronization with the version of the common memory object library or even with each other. Therefore, it is the application developer's responsibility to verify that distribution components are at the same version number. If this verification is not done, unintentional, undesirable and unexplainable errors may appear during the execution of the application program. These errors may be a direct result of differences in constant definitions or data structures that may have been introduced in subsequent versions of the common memory distribution.

CM_VERSION is defined and initialized in file cm_const.h via:

```
#define CM_VERSION     "1.0a"
```

However, two logical variables called CM_VERSION actually exist: one that is easily accessible to the application program and one that is not so easily accessible. It is neither intended nor desirable for the user application to change the common memory version identifiers.

To determine and verify that all common memory distribution components are at the same version, perform the following steps:

(1) Manually inspect the source listings of files cm_const.h and cm_types.h. Both files will have an indication of their common memory version number. File cm_const.h will have it as part of a #DEFINE statement and file cm_types.h will have it as part of a comment area near the beginning of the file. Optionally, you can place a "printf" at the beginning of your program to remind you what the INCLUDE file versions are once you have verified that both cm_const.h and cm_types.h are at the same version level. For example:

```
printf("file cm_const.h is at version %s\n",CM_VERSION);
```

(2) Display the common memory version number in file cm_funcs.obj by compiling and linking a (simple) program that sets the CM_DEBUG_LEVEL equal to an integer value of 1 (or greater) and then calls function cm_ini. Compare this version number with the value determined using step (1).

If the common memory components do not have the same version number, it will be necessary to locate the (most recent) matching set of files.

5. APPLICATION PROGRAM DEVELOPMENT

5.1. The PC Common Memory Distribution Kit

The PC common memory distribution kit consists of this documentation set and the following files:

- cm_const.h - contains the definition of constants.
- cm_types.h - contains the data structure definitions.
- cm_funcs.obj - contains the common memory interface functions. This is distributed in its object file form in order to minimize the potential for user-initiated changes that may later result in inexplicable common memory behavior and to control the possible divergence of common memory interfaces from that identified in this documentation.
- sfuncs.obj - contains utilities used by the common memory library, some of which may be useful to the application process, too. It must be included during application program linking.

5.2. Compiling Programs That Use the Common Memory Library

Two of the common memory source files must be included during the user application compilation. They are files cm_types.h and cm_const.h. File cm_types.h references variables defined in cm_const.h, so cm_const.h must be included before cm_types.h. For the C language compiler, the directive is

```
#include "cm_const.h"
#include "cm_types.h"
```

If the user application intends to access or manipulate variables CM_DEBUG_LEVEL or CM_GET_STATS, the respective "extern" statement should be inserted following the above two lines, as:

```
extern int     CM_DEBUG_LEVEL;
extern boolean CM_GET_STATS;
```

File cm_types.h defines all the common memory data structures.

File cm_const.h defines all the status return constants. This file is necessary if the user application will be testing the status value returned by each common memory function.

Applications that are written in programming languages other than C and that cannot import these declarations must provide equivalent data structures and constant declarations.

5.3. Linking Programs That Use the Common Memory Library

Two object files must be included in the link process. They are files cm_funcs.obj and sfuncs.obj.

File cm_funcs.obj contains all the common memory interface routines.

File sfuncs.obj contains some general support utilities required by the common memory interface routines. They are maintained in a separate file because they are useful to applications that do not need to use the common memory library.

APPENDIX A

STATUS REPORT CODES FOR THE COMMON MEMORY INTERFACE FUNCTIONS

The following list identifies the status report codes that the common memory interface routines can return. They are found in file CM_CONST.H.

Although each status is associated with a numeric value, it is strongly recommended that the user application use only the status name (e.g., I_CM_OK) when testing completion status. It is conceivable that the values associated with the status may change in future versions of the common memory interface, whereas status variable names will not change.

The status values are divided into two groups: informational (i.e., non-fatal) and fatal. The boundary is established at variable E_CM_FATALERR. Status values less than E_CM_FATALERR are informational. Status values greater than or equal to E_CM_FATALERR report fatal errors.

Informational status values are used to indicate that the call to the common memory interface function was successfully completed while concurrently providing some additional information affecting that call. Fatal status returns are used to indicate that the call to the common memory interface function was aborted and the cause of the abort.

CONSTANT	HEX VALUE	DESCRIPTION
I_CM_OK	0x0	normal success indicator
I_CM_MBXACTV	0x2	mailbox successfully undeclared, but other applications are still connected. In the case of cm_disc, this refers to the status of one or more mailboxes.
I_CM_DUPMBX	0x4	mailbox was previously declared by this application and for similar access (READ/XREAD or WRITE/XWRITE). If DECLARE was used to change access from exclusive to non-exclusive (or vice-versa), then the change was made. Additional client entries are not made in the client list of the respective mailbox.

CONSTANT	HEX VALUE	DESCRIPTION
I_CM_MOREDATA	0x5	cm_read was successfully performed. However the int variable pointed to by "nr_bytes" was non-zero when the call was made and specified a number of bytes that was less than the number of bytes actually available in the mailbox. Only the number of bytes specified in the int variable pointed to by "nr_bytes" was transferred to the user data are. More data is actually available.

Errors greater than 0fx are FATAL errors. This means that the requested action was NOT performed.

E_CM_FATALERR	0x10	defines the start of FATAL ERROR range
E_CM_INSUFFMEM	0x10	insufficient memory. malloc failed
E_CM_MBXERR	0x20	base value for mbx errors
E_CM_MBXNOREAD	0x21	can't have READ - another has XREAD
E_CM_MBXNOXREAD	0x22	can't have XREAD - another fsm has either READ or XREAD
E_CM_MBXNOWRITE	0x24	can't have WRITE - another has XWRITE
E_CM_MBXNOXWRITE	0x28	can't have XWRITE - another fsm has either WRITE or XWRITE
E_CM_MBXSIZE	0x29	invalid size, returned if: (1) negative size in cm_declare, or (2) byte count < 1 for cm_write, or (3) size doesn't match previously-declared size (for cm_declare), or (4) attempt to write nr_bytes greater than declared size
E_CM_MBXACCESS	0x2a	invalid mbx access, returned if: (1) unrecognizable mbx access code supplied for a cm_declare or cm_undeclare, or (2) invalid list_type supplied to cm_get_fsm_list

CONSTANT	HEX VALUE	DESCRIPTION
E_CM_MBXACCBOTH	0x2b	can't declare both READ and XREAD or WRITE and XWRITE in the same mbx declare. However, you can later declare a mbx to be READ after having previously declared it XREAD ... this changes your access and lets other fsm's have access.
E_CM_MBXNAME	0x2c	invalid mbx name - too long or none given
E_CM_MBXNOTDECL	0x2d	returned if : (1) fsm attempts to undeclare a mbx for which it is not a client for the respective access, or (2) fsm attempts to perform read or write actions w/ a mbx for which it is not a client for the respective access, or (3) fsm requests any cm action without being a client of cm, or (4) attempt to get info for a mbx that is not in common memory
E_CM_FSMERR	0x30	base value for fsm errors
E_CM_FSMNAME	0x31	invalid name - too long or none given
E_CM_FSMNOTINCM	0x32	fsm not a common memory client

APPENDIX B

COMMON MEMORY MAILBOX ACCESS CODES

The following list identifies the mailbox access codes that the common memory interface routines will accept. They are found in file CM_CONST.H.

Although each access type is associated with a numeric value, it is strongly recommended that the user application use only the access name (e.g., CM_READ_ACCESS). It is conceivable that the values associated with the access names may change in future versions of the common memory interface, whereas access names will not change.

CONSTANT	HEX VALUE	DESCRIPTION
CM_READ_ACCESS	0x1	The declaring application is requesting shared mailbox read access. This will be granted as long as no other application has previously declared exclusive read access. Other applications may declare shared read access for the same mailbox.
CM_XREAD_ACCESS	0x2	The declaring application is requesting exclusive mailbox read access. This will be granted as long as no other application has read or exclusive read access. Use this option with caution, since it also precludes the network from accessing this mailbox. A future extension will allow the mailbox declarer to generate a mailbox access list.
CM_WRITE_ACCESS	0x4	The declaring application is requesting shared mailbox write access. This will be granted as long as no other application has previously declared exclusive write access. Other applications may declare shared write access for the same mailbox.
CM_XWRITE_ACCESS	0x8	The declaring application is requesting exclusive mailbox write access. This will be granted as long as no other application has write or exclusive write access. Use this option with caution, since it also

precludes the network from accessing this mbx. A future extension will allow mbx declarer to generate a mbx access list.

Mailbox access can be specified by using the appropriate access constant individually or by specifying the bit-wise OR of two or more access constants. For example, using the C language syntax, both READ and WRITE access can be specified via

CM_READ_ACCESS | CM_WRITE_ACCESS

The common memory manager checks the validity of the access request. Any access code combination can be submitted except one where the user process is requesting both exclusive and non-exclusive access for the same purpose (e.g., READ or WRITE). The following combinations are illegal:

CM_READ_ACCESS | CM_XREAD_ACCESS | <anything else>

CM_WRITE_ACCESS | CM_XWRITE_ACCESS | <anything else>

APPENDIX C

DATA STRUCTURES USED IN THE COMMON MEMORY INTERFACE FUNCTIONS

The data structures specific to the common memory functions available to the user application are detailed below. They are a subset of the complete set of data structures used by the common memory library. They can be found in file cm_types.h.

In some instances, constants are referenced in the type definitions. When referenced, they are displayed in uppercase characters. The values for these constants are found in file cm_const.h.

C.1. STANDARDIZED DEFINITIONS

The following are some standardized definitions used in subsequent type declarations.

```
typedef   unsigned char    byte;
typedef   long int         timestamp;     /* nr of seconds since
                                             01 Jan 1970 */
```

C.2. UPDATE LIST STRUCTURE

When a mailbox is written to, the common memory manager checks to see if an entry identifying this mailbox already exists on the update list maintained by each reader client of that mailbox. If an entry already exists, no further action is taken. If an entry does not exist, an entry identifying this mailbox is appended to the update list. The list is maintained in FIFO (first-in-first-out) order.

```
typedef struct update_list {
       int mbxhandle;              /* identifies the mbx that has
                                      been changed */
       struct update_list *next;   /* points to next list entry */
       };
```

C.3. COMMON MEMORY STATISTICS

The following structures show how the common memory manager returns common memory utilization statistics.

C.3.1. Function Call Statistics

When a user application requests common memory statistics for this category, the common memory manager places a copy of the statistics into the user-specified data area. The data area must be large enough to contain the data or unpredictable and undesirable side effects may result.

The following structure is used to return usage statistics for each of the user-callable functions of common memory. Its primary use is expected to be as a diagnostic tool.

```
typedef struct {
        char fsm[CM_MAXFSMNAMELENGTH];      /* name of fsm */
        char mbx[CM_MAXMBXNAMELENGTH];      /* name of mbx */
        timestamp when;
        unsigned int nr_times;              /* nr times this
                                               service called */
        } base_stats;

typedef struct {
        base_stats success;                 /* for successes */
        base_stats failure;                 /* for failures  */
        } group_stats;

typedef struct {
        group_stats cm_declare,             /* id the various */
                    cm_undeclare,           /* routines */
                    cm_write,
                    cm_read,
                    cm_ckmail,
                    cm_disc,
                    cm_get_mbx_list,
                    cm_get_fsm_list,
                    cm_get_cm_stats,
                    cm_get_fsm_stats,
                    cm_get_mbx_stats;
        } cm_activity_stats;                /* for cm calls */

typedef struct {
        unsigned int
            mbx_ttl,          /* total nr of mbx's declared */
            mbx_active,       /* nr of mbx's currently active */
            fsm_ttl,          /* total nr fsm's declared */
            fsm_active;       /* nr fsm's currently active */
        } cm_client_stats;
```

C.3.2. Mailbox and Client Lists

Two functions (cm_get_mbx_list and cm_get_fsm_list) return a pointer to a linked list containing the requested information in one of the arguments of the function call. In this case, it is the user's responsibility to FREE the linked list after the list's usefulness has been completed. As mentioned elsewhere (Section IV.1.2), UNIX-portable functions MALLOC and FREE are used within the common memory manager so the user must use function FREE to release the space allocated for this list.

The following structure is used to return a copy of the mailbox list to the common memory client:

```
typedef struct mbx_list_type {
      mbxname[CM_MAXMBXNAMELENGTH];   /* contains name of mbx */
      struct mbx_list_type *next;     /* next block, or NULL */
      };
```

The following structure is used to return a copy of the common memory client list to the user application:

```
typedef struct fsm_list_type {
      fsmname[CM_MAXFSMNAMELENGTH];   /* contains name of fsm */
      struct fsm_list_type *next;     /* next block, or NULL */
      };
```

C.3.3. Mailbox and Client Statistics

When a user application requests common memory statistics for this category, the common memory manager places a copy of the statistics into the user-specified data area. The data area must be large enough to contain the data or unpredictable and undesirable side effects may result.

The following is returned whenever common memory client information is requested via function cm_get_fsm_stats:

```
typedef struct {
       int  nr_read_mbx;             /* nr of read mbx declared */
       unsigned int  nr_reads;       /* nr of mbx reads performed */
       timestamp read_time;          /* time of last read */
       char mbx_read[CM_MAXMBXNAMELENGTH]; /* name of last mbx
                                        read, null-terminated */
       int  nr_write_mbx;            /* nr of write mbx declared */
       unsigned int  nr_writes;      /* nr of writes performed */
       timestamp write_time;         /* time of last write */
       char mbx_write[CM_MAXMBXNAMELENGTH]; /* name of last mbx
                                        written, null-terminated */
       int nr_updates;               /* nr entries on update_list */
       }cm_fsm_stats_rec;
```

The following is returned whenever mailbox information is requested via function cm_get_mbx_stats:

```
typedef struct {
       int  handle;                  /* mbxhandle for this mbx */
       int  declaredlength;          /* nr bytes declared */
       int  msglength;               /* nr bytes currently stored */
       int  read_fsms;               /* nr of READ subscribers */
       unsigned int  read_accesses;  /* nr of READ accesses */
       timestamp read_time;          /* time of last read */
       char reader[CM_MAXFSMNAMELENGTH];  /* name of last reader
                                        fsm, null-terminated */
       int  write_fsms;              /* nr of WRITE subscribers */
       unsigned int  write_accesses; /* nr of WRITE accesses */
       timestamp write_time;         /* time of last write */
       char writer[CM_MAXFSMNAMELENGTH];  /* name of last writer
                                        fsm, null-terminated */
       char xreader[CM_MAXFSMNAMELENGTH]; /* name of exclusive
                                        reader, null-terminated */
       char xwriter[CM_MAXFSMNAMELENGTH]; /* name of exclusive
                                        writer, null-terminated */
       }cm_mbx_stats_rec;
```

APPENDIX D

SAMPLE PROGRAM
DEMONSTRATING A COMMON MEMORY MAILBOX INTERACTION
BETWEEN
TWO LOGICALLY SEPARATE APPLICATIONS

D.1. PURPOSE OF THE PROGRAM

This program serves as a coding example to programmers wishing to develop a PC common memory application. It incorporates all available common memory function calls and is heavily commented in order to document what the program is attempting to do.

The reader is refered to the program listing for further comments about the program.

D.2. PROGRAM LISTING

The program name is "cm_sampl.c" and begins on the following page.

```
/*  cm_sampl.c   -   sample program to show usage of the common memory routines.
                     Some procedural and explanatory notes are listed below.


1)  When compiling:   Two common memory files must be available during the
                      compile process, cm_types.h and cm_const.h, in order to
                      bring in the common memory data structure definitions.
                      These files do not reference any others, so they may
                      be placed into a library directory, if desired.

                      File cm_const.h must be included before cm_types.h.


2)  When linking:     Include files cm_funcs.obj and sfuncs.obj in the
                      linking process.

************************************

In this sample program, the following is expected to happen:

3)  Since the common memory version of the two INCLUDE files can get out of
    sync with the version of the common memory object library, it is a good
    idea to check the respective versions.  To avoid cryptic errors, these
    version numbers MUST be identical.  If they do not match, it will be
    necessary to locate the (most recent) matching set of files.

    Check the respective versions via the following (demonstrated below):
           1)  "printf" the CM_VERSION from the common memory INCLUDE files,
               or check the include file's source code manually, and
           2)  set CM_DEBUG_LEVEL = 1 to generate a "printf"from the common
               memory initialization routine that is internal to the (object)
               library.  The initialization routine is called the very first
               time you attempt to perform a DECLARE, UNDECLARE, READ, WRITE,
               CKMAIL, or DISCONNECT action.  Set CM_DEBUG_LEVEL to zero for
               those applications where you don't need to monitor the
               common memory library version level.


4)  A common memory interaction occurs between two application processes:
        (1) process_1 declares a mailbox for READ and WRITE access and
            writes something into it.
        (2) process_2 declares the same mailbox for READ access and checks
            if any of its READ mailboxes have been updated.  Since it has
            only one mailbox, and it was just declared, no updates have yet
            been posted.  process_1 does not perform any read function until
```

after the next time the mailbox is written to. Thus process_1 misses the first mailgram. The next time process_1 writes into the mailbox, process_2 expects to see an update record (since they are called sequentially from the main program). process_2 will return FALSE until it receives its first mailgram. Thereafter it will always return TRUE.

(3) process_1 now enters a state where it writes a new mailgram to the mailbox every delta_time seconds. If delta_time seconds have not elapsed since the last time it was called, process_1 will return FALSE. Only after process_1 has written the specified nr_times to the mailbox will it return TRUE.

(4) Since process_2 is called immediately after each call to process_1, process_2 should display the contents of the newly-written mailbox (almost) immediately after it was written by process_1.

(5) only when both process_1 and process_2 return TRUE does the main program exit the "when" loop.

5) Get and display the list of all applications active in common memory.

6) Get and display the names of all mailboxes currently in common memory.

7) Get and display the list of READ/XREAD clients for mailbox proc_1_mbx.

8) Get and display the list of mailboxes declared for READ/XREAD access by process_1.

9) Get and display the available common memory statistics for an application that is not in common memory to see the error generated. Use "process_3".

10) Get and display the available common memory statistics for an application that is in common memory. Use "process_1".

11) Get and display the available common memory statistics for a mailbox that is not in common memory to see the error generated. Use "proc_3_mbx".

12) Get and display the available common memory statistics for a mailbox that is in common memory. Use "proc_1_mbx".

13) Undeclare the process_2 mailbox connections by having it call cm_undeclare. Once all of it's mailboxes are undeclared, the common memory manager will remove it from it's client list. Demonstrate this by displaying the list of applications still in common memory. Only 1 should be left.

14) Undeclare the process_1 mailbox connections by having it call cm_disc Once all of it's mailboxes are undeclared, the common memory manager will remove it from it's client list. Demonstrate this by displaying the list of applications still in common memory. The common memory client list should be empty.

15) Retrieve and display the common memory statistics.

```
 */

#include <stdio.h>
#include "cm_const.h"                         /* include before cm_types.h */
#include "cm_types.h"
extern int CM_DEBUG_LEVEL;                    /* declared in the cm library */

enum states {INIT, NORMAL, SHUTDOWN};

/* ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
   ¦    main    ¦
   ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
*/
main ()
{ boolean DONE1 = FALSE;
  boolean DONE2 = FALSE;
  char *mbxname, *fsmname;
  struct mbx_list_type *mbx_list;
  struct fsm_list_type *fsm_list, *tptr;
  cm_fsm_stats_rec fsm_stats;
  cm_mbx_stats_rec mbx_stats;
  cm_activity_stats cm_activity;
  cm_client_stats cm_clients;
  int status, nr;
  enum states p1_state = INIT,
              p2_state = INIT;


  printf("\n\n\t\t   ¦--------------------------------¦\n"
           "\t\t   ¦ section corresponding to       ¦\n"
           "\t\t   ¦                                ¦\n"
           "\t\t   ¦         NOTE  # 3              ¦\n"
           "\t\t   ¦                                ¦\n"
           "\t\t   ¦ Display the common memory      ¦\n"
           "\t\t   ¦ library version numbers for both ¦\n"
           "\t\t   ¦ the INCLUDE files and the      ¦\n"
           "\t\t   ¦ linked object file.            ¦\n"
           "\t\t   ¦                                ¦\n"
           "\t\t    --------------------------------  \n");
  /* the first common memory version number comes from the common memory
     INCLUDE file, cm_const.h   */
  printf("\nThe common memory include file is version %s\n",CM_VERSION);
```

```
CM_DEBUG_LEVEL = 1;
/* a common memory debug level of '1' means: display the common memory
   object library version nr.  The version nr will be displayed by function
   cm_ini (located in file cm_utils.c, and thus part of the object file
   cm_funcs.obj) the first time a common memory interface function is
   called.   This can be forced to occur NOW by inserting a call to
   function cm_ini here, in order to perform common memory initialization
   NOW.  The CM_DEBUG_LEVEL must be set to 1 (or greater) before you make
   the call to cm_ini in order to display the version number.  */
cm_ini();

printf("\n\n\t\t   |-----------------------------------|\n"
              "\t\t   | section corresponding to          |\n"
              "\t\t   |                                   |\n"
              "\t\t   |           NOTE  # 4               |\n"
              "\t\t   |                                   |\n"
              "\t\t   | Two processes access a common     |\n"
              "\t\t   | mailbox, transfering information  |\n"
              "\t\t   | from one to the other.            |\n"
              "\t\t   |                                   |\n"
              "\t\t    -----------------------------------  \n");
while ((! DONE1) || (! DONE2))  {          /* continue until both are done */
   DONE1 = process_1(&p1_state);
   DONE2 = process_2(&p2_state);
}


printf("\n\n\t\t   |-----------------------------------|\n"
              "\t\t   | section corresponding to          |\n"
              "\t\t   |                                   |\n"
              "\t\t   |           NOTE  # 5               |\n"
              "\t\t   |                                   |\n"
              "\t\t   | Get and display the list of ALL   |\n"
              "\t\t   | applications participating in     |\n"
              "\t\t   | common memory.                    |\n"
              "\t\t   |                                   |\n"
              "\t\t    -----------------------------------  \n");
printf("\n\nmain: Get and display the list of ALL applications currently"
        "\n      participating in common memory\n");
status = cm_get_fsm_list(NULL,'',&fsm_list,&nr);
if (status >= E_CM_FATALERR)               /* was there a fatal error */
   printf("main: FATAL ERROR %s returned from cm_get_fsm_list\n",
          cm_get_statusname(status));
```

```
else
   if (fsm_list) {
      printf("main: following %d applications are active in common memory...\n", nr);
      while (fsm_list)  {                       /* display the application names */
         printf("\t%s\n",fsm_list->fsmname);
         tptr = fsm_list->next;
         free(fsm_list);                        /* free the blocks when no longer needed */
         fsm_list = tptr;
      }
   }else printf("main: no applications active in common memory\n");


printf("\n\n\t\t    |-----------------------------------|\n"
            "\t\t    | section corresponding to          |\n"
            "\t\t    |                                   |\n"
            "\t\t    |           NOTE  # 6               |\n"
            "\t\t    |                                   |\n"
            "\t\t    | Get and display the list of ALL   |\n"
            "\t\t    | mailboxes in common memory.       |\n"
            "\t\t    |                                   |\n"
            "\t\t     -----------------------------------  \n");
printf("\n\nmain: Get and display the list of ALL mailboxes in common memory\n");
status = cm_get_mbx_list(NULL,'',&mbx_list,&nr);
if (status >= E_CM_FATALERR)                /* was there a fatal error */
   printf("main: FATAL ERROR %s returned from cm_get_mbx_list\n",
          cm_get_statusname(status));
else
   if (mbx_list) {
      printf("main: following %d mailboxes are declared in common memory...\n",nr);
      while (mbx_list)  {                       /* display the application names */
         printf("\t%s\n",mbx_list->mbxname);
         tptr = mbx_list->next;
         free(mbx_list);                        /* free the blocks when no longer needed */
         mbx_list = tptr;
      }
   }else printf("main: no mailboxes declared in common memory\n");


printf("\n\n\t\t    |-----------------------------------|\n"
            "\t\t    | section corresponding to          |\n"
            "\t\t    |                                   |\n"
            "\t\t    |           NOTE  # 7               |\n"
            "\t\t    |                                   |\n"
            "\t\t    | Get and display the list or       |\n"
            "\t\t    | READ/XREAD clients for mailbox    |\n"
            "\t\t    | 'proc_1_mbx'.                     |\n"
            "\t\t    |                                   |\n"
            "\t\t     -----------------------------------  \n");
printf("\n\nmain: Get and display the list of READ/XREAD clients for mailbox"
        "\n      proc_1_mbx.\n");
```

```
mbxname = "proc_1_mbx";
status = cm_get_fsm_list(mbxname,'r',&fsm_list,&nr);
if (status >= E_CM_FATALERR)                    /* was there a fatal error */
   printf("main: FATAL ERROR %s returned from cm_get_fsm_list\n",
          cm_get_statusname(status));
else
   if (fsm_list) {
      printf("main: %s has following %d READ/XREAD clients ...\n", mbxname,nr);
      while (fsm_list)  {                       /* display the application names */
         printf("\t%s\n",fsm_list->fsmname);
         tptr = fsm_list->next;
         free(fsm_list);                        /* free the blocks when no longer needed */
         fsm_list = tptr;
      }
   }else printf("main: no READ/XREAD applications declared mailbox %s\n",mbxname);


printf("\n\n\t\t     |-----------------------------------|\n"
            "\t\t    | section corresponding to           |\n"
            "\t\t    |                                    |\n"
            "\t\t    |            NOTE  # 8               |\n"
            "\t\t    |                                    |\n"
            "\t\t    | Get and display the list of        |\n"
            "\t\t    | mailboxes declared for READ/       |\n"
            "\t\t    | XREAD access by 'process_1'.       |\n"
            "\t\t    |                                    |\n"
            "\t\t     -----------------------------------  \n");
printf("\n\nmain: Get and display the list of mailboxes declared for READ/XREAD"
        "\n      by process_1.\n");
fsmname = "process_1";
status = cm_get_mbx_list(fsmname,'r',&mbx_list,&nr);
if (status >= E_CM_FATALERR)                    /* was there a fatal error */
   printf("main: FATAL ERROR %s returned from cm_get_mbx_list\n",
          cm_get_statusname(status));
else
   if (mbx_list) {
      printf("main: %s has following %d READ mailboxes ...\n", fsmname,nr);
      while (mbx_list)  {                       /* display the application names */
         printf("\t%s\n",mbx_list->mbxname);
         tptr = mbx_list->next;
         free(mbx_list);                        /* free the blocks when no longer needed */
         mbx_list = tptr;
      }
   }else printf("main: no READ/XREAD mailboxes declared by %s\n",fsmname);
```

```
printf("\n\n\t\t    |-----------------------------------|\n"
          "\t\t    | section corresponding to          |\n"
          "\t\t    |                                   |\n"
          "\t\t    |          NOTE  # 9                |\n"
          "\t\t    |                                   |\n"
          "\t\t    | Get and display the available     |\n"
          "\t\t    | common memory statistics for an   |\n"
          "\t\t    | application that is not in        |\n"
          "\t\t    | common memory.  Error will be     |\n"
          "\t\t    | generated.  Use 'process_3'.      |\n"
          "\t\t    |                                   |\n"
          "\t\t     -----------------------------------  \n");
printf("\n\nmain: Get and display the available common memory statistics for"
        "\n      an application that doesn't exist.  Expect an error report.\n");
fsmname = "process_3";
status = cm_get_fsm_stats(fsmname,&fsm_stats);
if (status >= E_CM_FATALERR)                 /* was there a fatal error */
   printf("main: FATAL ERROR %s returned from cm_get_fsm_stats\n",
          cm_get_statusname(status));
else  {
   printf("\t%d read mbx's\n\t%u reads performed\n\tlast one at ",
          fsm_stats.nr_read_mbx,
          fsm_stats.nr_reads);
   printf("%s\tmbx: %s\n\t%d write mbx's\n\t%u writes performed\n\t"
          "last one at ",
          (fsm_stats.read_time) ? ctime(&fsm_stats.read_time) : "<none>\n",
          fsm_stats.mbx_read,
          fsm_stats.nr_write_mbx,
          fsm_stats.nr_writes);
   printf("%s\tmbx: %s\n\t%d updates on the update list\n",
          (fsm_stats.write_time) ? ctime(&fsm_stats.write_time) : "<none>\n",
          fsm_stats.mbx_write,
          fsm_stats.nr_updates);
}


printf("\n\n\t\t    |-----------------------------------|\n"
          "\t\t    | section corresponding to          |\n"
          "\t\t    |                                   |\n"
          "\t\t    |          NOTE  # 10               |\n"
          "\t\t    |                                   |\n"
          "\t\t    | Get and display the available     |\n"
          "\t\t    | common memory statistics for      |\n"
          "\t\t    | process_1.                        |\n"
          "\t\t    |                                   |\n"
          "\t\t     -----------------------------------  \n");
printf("\n\nmain: Get and display the available common memory statistics for"
        "\n      an application that does exist.\n");
fsmname = "process_1";
status = cm_get_fsm_stats(fsmname,&fsm_stats);
```

```
if (status >= E_CM_FATALERR)                /* was there a fatal error */
   printf("main: FATAL ERROR %s returned from cm_get_fsm_stats\n",
          cm_get_statusname(status));
else  {
   printf("\t%d read mbx's\n\t%u reads performed\n\tlast one at ",
          fsm_stats.nr_read_mbx,
          fsm_stats.nr_reads);
   printf("%s\tmbx: %s\n\t%d write mbx's\n\t%u writes performed\n\t"
          "last one at ",
          (fsm_stats.read_time) ? ctime(&fsm_stats.read_time) : "<none>\n",
          fsm_stats.mbx_read,
          fsm_stats.nr_write_mbx,
          fsm_stats.nr_writes);
   printf("%s\tmbx: %s\n\t%d updates on the update list\n",
          (fsm_stats.write_time) ? ctime(&fsm_stats.write_time) : "<none>\n",
          fsm_stats.mbx_write,
          fsm_stats.nr_updates);
}

printf("\n\n\t\t    |----------------------------------|\n"
          "\t\t    | section corresponding to         |\n"
          "\t\t    |                                  |\n"
          "\t\t    |          NOTE  # 11              |\n"
          "\t\t    |                                  |\n"
          "\t\t    | Get and display the available    |\n"
          "\t\t    | common memory statistics for a   |\n"
          "\t\t    | nonexistent mailbox.             |\n"
          "\t\t    |                                  |\n"
          "\t\t     ---------------------------------- \n");
printf("\n\nmain: Get and display the available common memory statistics for"
        "\n      a mailbox that doesn't exist.\n");
mbxname = "proc_3_mbx";
status = cm_get_mbx_stats(mbxname,&mbx_stats);
if (status >= E_CM_FATALERR)                /* was there a fatal error */
   printf("main: FATAL ERROR %s returned from cm_get_mbx_stats\n",
          cm_get_statusname(status));
else  {
   printf("\tmbxhandle %d, declared length %d, msg length %d\n\t"
          "%d readers w/ %u read actions\n\t   last by %s @",
          mbx_stats.handle, mbx_stats.declaredlength,mbx_stats.msglength,
          mbx_stats.read_fsms, mbx_stats.read_accesses,
          mbx_stats.reader);
   printf(" %s\t%d writers w/ %u write actions\n\t   last by %s @",
          (mbx_stats.read_time) ? ctime(&mbx_stats.read_time) : "<none>\n",
          mbx_stats.write_fsms, mbx_stats.write_accesses,
          mbx_stats.writer);
```

```
   printf(" %s\txreader: %s\n\txwriter: %s\n",
          (mbx_stats.write_time) ? ctime(&mbx_stats.write_time) : "<none>\n",
          mbx_stats.xreader,
          mbx_stats.xwriter);
}


printf("\n\n\t\t    ¦-----------------------------------¦\n"
          "\t\t    ¦ section corresponding to          ¦\n"
          "\t\t    ¦                                   ¦\n"
          "\t\t    ¦         NOTE  # 12                ¦\n"
          "\t\t    ¦                                   ¦\n"
          "\t\t    ¦ Get and display the available     ¦\n"
          "\t\t    ¦ common memory statistics for      ¦\n"
          "\t\t    ¦ 'proc_1_mbx'.                     ¦\n"
          "\t\t    ¦                                   ¦\n"
          "\t\t     -----------------------------------  \n");
printf("\n\nmain: Get and display the available common memory statistics for"
        "\n      a mailbox that does exist.\n");
mbxname = "proc_1_mbx";
status = cm_get_mbx_stats(mbxname,&mbx_stats);
if (status >= E_CM_FATALERR)                  /* was there a fatal error */
   printf("main: FATAL ERROR %s returned from cm_get_mbx_stats\n",
          cm_get_statusname(status));
else  {
   printf("\tmbxhandle %d, declared length %d, msg length %d\n\t"
          "%d readers w/ %u read actions\n\t   last by %s @",
          mbx_stats.handle, mbx_stats.declaredlength,mbx_stats.msglength,
          mbx_stats.read_fsms, mbx_stats.read_accesses,
          mbx_stats.reader);
   printf(" %s\t%d writers w/ %u write actions\n\t   last by %s @",
          (mbx_stats.read_time) ? ctime(&mbx_stats.read_time) : "<none>\n",
          mbx_stats.write_fsms, mbx_stats.write_accesses,
          mbx_stats.writer);
   printf(" %s\txreader: %s\n\txwriter: %s\n",
          (mbx_stats.write_time) ? ctime(&mbx_stats.write_time) : "<none>\n",
          mbx_stats.xreader,
          mbx_stats.xwriter);
}
```

```
printf("\n\n\t\t     |-----------------------------------|\n"
           "\t\t     | section corresponding to          |\n"
           "\t\t     |                                   |\n"
           "\t\t     |          NOTE  # 13               |\n"
           "\t\t     |                                   |\n"
           "\t\t     | Cause process_2 to undeclare all  |\n"
           "\t\t     | its common memory mailbox         |\n"
           "\t\t     | connections.  Display the list    |\n"
           "\t\t     | or remaining common memory        |\n"
           "\t\t     | clients.  Only one client should  |\n"
           "\t\t     | be left.                          |\n"
           "\t\t     |                                   |\n"
           "\t\t     | Process_2 will use cm_undeclare   |\n"
           "\t\t     | to disconnect from the mailboxes  |\n"
           "\t\t     |                                   |\n"
           "\t\t      -----------------------------------  \n");
printf("\n\nmain: Cause process_2 to undeclare its common memory mailbox connections."
        "\n      Then display the common memory client list.  Expect only one client.\n");
p2_state = SHUTDOWN;
DONE2 = process_2(&p2_state);
status = cm_get_fsm_list(NULL,'',&fsm_list,&nr);
if (status >= E_CM_FATALERR)                /* was there a fatal error */
   printf("main: FATAL ERROR %s returned from cm_get_fsm_list\n",
          cm_get_statusname(status));
else
   if (fsm_list) {
      printf("main: following %d applications are active in common memory...\n", nr);
      while (fsm_list)  {                   /* display the application names */
         printf("\t%s\n",fsm_list->fsmname);
         tptr = fsm_list->next;
         free(fsm_list);                    /* free the blocks when no longer needed */
         fsm_list = tptr;
      }
   }else printf("main: no applications active in common memory\n");
```

```
printf("\n\n\t\t     |-----------------------------------|\n"
             "\t\t     | section corresponding to          |\n"
             "\t\t     |                                   |\n"
             "\t\t     |          NOTE  # 14               |\n"
             "\t\t     |                                   |\n"
             "\t\t     | Cause process_1 to undeclare all  |\n"
             "\t\t     | its common memory mailbox         |\n"
             "\t\t     | connections.  Display the list    |\n"
             "\t\t     | or remaining common memory        |\n"
             "\t\t     | clients.  Only one client should  |\n"
             "\t\t     | be left.                          |\n"
             "\t\t     |                                   |\n"
             "\t\t     | Process_1 will use cm_disc to     |\n"
             "\t\t     | disconnect from the mailboxes.    |\n"
             "\t\t     |                                   |\n"
             "\t\t      -----------------------------------  \n");
printf("\n\nmain: Cause process_1 to undeclare its common memory mailbox connections."
         "\n     Then display the common memory client list.  Expect zero clients.\n");
p1_state = SHUTDOWN;
DONE1 = process_1(&p1_state);
status = cm_get_fsm_list(NULL,'',&fsm_list,&nr);
if (status >= E_CM_FATALERR)                  /* was there a fatal error */
   printf("main: FATAL ERROR %s returned from cm_get_fsm_list\n",
          cm_get_statusname(status));
else
   if (fsm_list) {
      printf("main: following %d applications are active in common memory...\n", nr);
      while (fsm_list)  {                       /* display the application names */
         printf("\t%s\n",fsm_list->fsmname);
         tptr = fsm_list->next;
         free(fsm_list);                        /* free the blocks when no longer needed */
         fsm_list = tptr;
      }
   }else printf("main: no applications active in common memory\n");

printf("\n\n\t\t     |-----------------------------------|\n"
             "\t\t     | section corresponding to          |\n"
             "\t\t     |                                   |\n"
             "\t\t     |          NOTE  # 15               |\n"
             "\t\t     |                                   |\n"
             "\t\t     | Retrieve and display the common   |\n"
             "\t\t     | memory usage statistics.          |\n"
             "\t\t     |                                   |\n"
             "\t\t      -----------------------------------  \n");
status = cm_get_cm_stats(&cm_activity, &cm_clients);
if (status >= E_CM_FATALERR)                  /* was there a fatal error */
   printf("main: FATAL ERROR %s returned from cm_get_cm_stats\n",
          cm_get_statusname(status));
```

```
    else {
       printf("A total of %u mbx's were created.  %u still active.\n"
              "A total of %u fsm's were clients of cm.  %u still active.\n",
              cm_clients.mbx_ttl, cm_clients.mbx_active,
              cm_clients.fsm_ttl, cm_clients.fsm_active);
       dmp_base_stats(cm_activity.cm_declare.failure,"cm_declare failures");
       dmp_base_stats(cm_activity.cm_declare.success,"cm_declare successes");
       dmp_base_stats(cm_activity.cm_undeclare.failure,"cm_undeclare failures");
       dmp_base_stats(cm_activity.cm_undeclare.success,"cm_undeclare successes");
       dmp_base_stats(cm_activity.cm_write.failure,"cm_write failures");
       dmp_base_stats(cm_activity.cm_write.success,"cm_write successes");
       dmp_base_stats(cm_activity.cm_read.failure,"cm_read failures");
       dmp_base_stats(cm_activity.cm_read.success,"cm_read successes");
       dmp_base_stats(cm_activity.cm_ckmail.failure,"cm_ckmail failures");
       dmp_base_stats(cm_activity.cm_ckmail.success,"cm_ckmail successes");
       dmp_base_stats(cm_activity.cm_disc.failure,"cm_disc failures");
       dmp_base_stats(cm_activity.cm_disc.success,"cm_disc successes");
       dmp_base_stats(cm_activity.cm_get_mbx_list.failure,"cm_get_mbx_list failures");
       dmp_base_stats(cm_activity.cm_get_mbx_list.success,"cm_get_mbx_list successes");
       dmp_base_stats(cm_activity.cm_get_fsm_list.failure,"cm_get_fsm_list failures");
       dmp_base_stats(cm_activity.cm_get_fsm_list.success,"cm_get_fsm_list successes");
       dmp_base_stats(cm_activity.cm_get_cm_stats.failure,"cm_get_cm_stats failures");
       dmp_base_stats(cm_activity.cm_get_cm_stats.success,"cm_get_cm_stats successes");
       dmp_base_stats(cm_activity.cm_get_fsm_stats.failure,"cm_get_fsm_stats failures");
       dmp_base_stats(cm_activity.cm_get_fsm_stats.success,"cm_get_fsm_stats successes");
       dmp_base_stats(cm_activity.cm_get_mbx_stats.failure,"cm_get_mbx_stats failures");
       dmp_base_stats(cm_activity.cm_get_mbx_stats.success,"cm_get_mbx_stats successes");
    }
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 *  ¦ dmp_base_stats ¦
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
int dmp_base_stats(func,msg)
base_stats func;
char *msg;
{
    printf("%s: %u\n\tlast by fsm: %s\n\tfor mbx: %s\n\tat: %s\n",
           msg,func.nr_times,
           (func.fsm) ? func.fsm : " ",
           (func.mbx) ? func.mbx : " ",
           (func.when) ? ctime(&func.when) : "\n");
}
```

```
/* ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦     This process initializes itself the first time it is called.
   ¦  process_1  ¦     Next, it writes into a mailbox, records the time of write,
   ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦     and decrements the number of times that it is supposed to
                       write into the mailbox.  If the result is zero, it
                       returns TRUE, else it returns FALSE.

                       The next time it is called, it checks if delta_time has
                       elapsed.  If not, it returns FALSE.  If so, it once again
                       writes into the mailbox, records the time of the write,
                       decrements the counter, and returns either TRUE or FALSE
                       according to the algorithm described before.

                       At shutdown, the mailboxes are disconnected via cm_disc.
*/
boolean process_1(state)
enum states *state;
{ char data[200];
  int mbxsize, mbxaccess;
  timestamp time_now;
  struct update_list *update_ptr, *tptr;
  int nrbytes, status;

  static timestamp last_write,                /* timestamp for last cm_write */
                   delta_time = 10;           /* nr seconds between cm_write */
  static int nr_writes =  5;                  /* total nr of writes to perform */
  static int mbxhandle;
  static char *fsm = "process_1";
  static char *mbxname = "proc_1_mbx";

  switch (*state)   {
    case INIT  :
             *state = NORMAL;                 /* for next call */
             time(&last_write);               /* fake a last_write time */
             last_write -= 10;                /* take off 10 seconds */
             mbxsize = 150;
             mbxaccess = CM_READ_ACCESS | CM_WRITE_ACCESS;
             status = cm_declare(fsm, mbxname, mbxsize, mbxaccess, &mbxhandle);
             if (status >= E_CM_FATALERR)
                printf("process_1: FATAL ERROR %s returned from cm_declare\n",
                       cm_get_statusname(status));
             break;
```

```
    case NORMAL :
              time(&time_now);
              if ((time_now - last_write) >= delta_time)  {
                 last_write = time_now;
                 sprintf(data,"process_1 wrote mbx (%d writes remaining) on %s",
                        nr_writes-1, ctime(&time_now));
                 nrbytes = strlen(data);
                 status = cm_write(fsm, &mbxhandle, data, &nrbytes);
                 if (status >= E_CM_FATALERR)
                    printf("process_1: FATAL ERROR %s returned from cm_write\n",
                          cm_get_statusname(status));
                 return ((--nr_writes) == 0);
              }else return(FALSE);          /* insufficient time elapsed */
              break;
    case SHUTDOWN :
              status = cm_disc(fsm);
              if (status >= E_CM_FATALERR) {
                 printf("process_1: FATAL ERROR %s returned from cm_disc\n",
                       cm_get_statusname(status));
                 return (TRUE);             /* give MAIN the option for clean getaway */
              }
              break;
  }
}


/* |||||||||||||||     This process initializes itself the first time it is called.
   |  process_2  |     Next, and every time it is called thereafter, it checks its
   |||||||||||||||     mail to see if any of the mailboxes that it has declared as
                       READ or XREAD access have been updated.  If none, it returns
                       a boolean value indicating whether it has read any of its
                       declared mailboxes since they were declared.  If updated
                       mailboxes exist, they are sequentially displayed, the
                       update list freed, and a boolean value is (set and) returned
                       to indicate that a mailbox has been read.

                       At shutdown, the mailboxes are disconnected via cm_undeclare.
*/
boolean process_2(state)
enum states *state;
{ char data[200];
  int mbxsize, mbxaccess;
  timestamp time_now;
  struct update_list *update_ptr, *tptr;
  int nrbytes, nr_updates, status;
```

```
static boolean read_one = FALSE;
static int mbxhandle;
static char *fsm = "process_2";
static char *mbxname = "proc_1_mbx";

switch (*state)  {
  case INIT   :
              mbxsize = 150;
              mbxaccess = CM_READ_ACCESS;
              status = cm_declare(fsm, mbxname, mbxsize, mbxaccess, &mbxhandle);
              if (status >= E_CM_FATALERR) {
                 printf("process_2: FATAL ERROR %s returned from cm_declare\n",
                        cm_get_statusname(status));
                 return (TRUE);
              }
              *state = NORMAL;
              break;
  case NORMAL :
              status = cm_ckmail(fsm,&update_ptr,&nr_updates);
              if (status >= E_CM_FATALERR) {
                 printf("process_2: FATAL ERROR %s returned from cm_ckmail\n",
                        cm_get_statusname(status));
                 return (TRUE);                /* try to give MAIN a graceful exit */
              }
              if (update_ptr)  {               /* read updated mailboxes */
                 read_one = TRUE;              /* remember that you read at least one */
                 tptr = update_ptr;            /* remember start of list for later FREE */
                 time(&time_now);
                 printf("\nprocess_2: it is now  %s",ctime(&time_now));
                 printf("process_2: %d mailbox update notifications\n",nr_updates);
                 while (update_ptr) {
                    nrbytes = 0;               /* to insure we get all of the mailgram */
                    status = cm_read(fsm, &update_ptr->mbxhandle, data, &nrbytes);
                    if (status >= E_CM_FATALERR) {
                       printf("process_2: FATAL ERROR %s returned from cm_read\n",
                              cm_get_statusname(status));
                       exit();
                    }
                    data[nrbytes] = 0;     /* trailing NULL so we can treat as string w/ printf *
                    printf("\tmbxhandle: %d, msg is %d bytes long\n"
                           "\tstatus returned is %s, or %x (hex)\n\tCONTENTS: %s",
                           update_ptr->mbxhandle,nrbytes,cm_get_statusname(status),status,data);
                    update_ptr = update_ptr->next;
                 }
                 cm_free_update_list(tptr);              /* FREE the update list */
                 return (read_one);
              }
              break;
```

```
    case SHUTDOWN :
                mbxaccess = CM_READ_ACCESS;
                status = cm_undeclare(fsm, mbxaccess, &mbxhandle);
                if (status >= E_CM_FATALERR) {
                   printf("process_1: FATAL ERROR %s returned from cm_undeclare\n",
                          cm_get_statusname(status));
                   return (TRUE);            /* give MAIN the option for clean getaway */
                }
                break;
  }
  return (read_one);
}
```

D.3. SAMPLE PROGRAM OUTPUT

The output generated by the sample program is shown below. The correlation between sections of the output and notations made in the program listing are clearly identified by in-line comments. In some cases, the sample program generates a line that is wider than can be represented on these typewriten pages. In these cases, the line is broken into two parts and the information is displayed on two sequential lines.

The output begins on the following page.

```
 ----------------------------------
| section corresponding to         |
|                                  |
|          NOTE  # 3               |
|                                  |
| Display the common memory        |
| library version numbers for both |
| the INCLUDE files and the        |
| linked object file.              |
|                                  |
 ----------------------------------

The common memory include file is version 1.0a
cm_ini: using common memory library version 1.0a


 ----------------------------------
| section corresponding to         |
|                                  |
|          NOTE  # 4               |
|                                  |
| Two processes access a common    |
| mailbox, transfering information |
| from one to the other.           |
|                                  |
 ----------------------------------

process_2: it is now  Sat Feb 27 19:50:55 1988
process_2: 1 mailbox update notifications
    mbxhandle: 1, msg is 69 bytes long
    status returned is i_CM_OK, or 0 (hex)
    CONTENTS: process_1 wrote mbx (4 writes remaining)
                  on Sat Feb 27 19:50:55 1988

process_2: it is now  Sat Feb 27 19:51:05 1988
process_2: 1 mailbox update notifications
    mbxhandle: 1, msg is 69 bytes long
    status returned is i_CM_OK, or 0 (hex)
    CONTENTS: process_1 wrote mbx (3 writes remaining)
                  on Sat Feb 27 19:51:05 1988
```

```
process_2: it is now  Sat Feb 27 19:51:15 1988
process_2: 1 mailbox update notifications
    mbxhandle: 1, msg is 69 bytes long
    status returned is I_CM_OK, or 0 (hex)
    CONTENTS: process_1 wrote mbx (2 writes remaining)
                  on Sat Feb 27 19:51:15 1988

process_2: it is now  Sat Feb 27 19:51:25 1988
process_2: 1 mailbox update notifications
    mbxhandle: 1, msg is 69 bytes long
    status returned is I_CM_OK, or 0 (hex)
    CONTENTS: process_1 wrote mbx (1 writes remaining)
                  on Sat Feb 27 19:51:25 1988

process_2: it is now  Sat Feb 27 19:51:35 1988
process_2: 1 mailbox update notifications
    mbxhandle: 1, msg is 69 bytes long
    status returned is I_CM_OK, or 0 (hex)
    CONTENTS: process_1 wrote mbx (0 writes remaining)
                  on Sat Feb 27 19:51:35 1988


          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
         | section corresponding to          |
         |                                   |
         |          NOTE  # 5                |
         |                                   |
         | Get and display the list of ALL   |
         | applications participating in     |
         | common memory.                    |
         |                                   |
          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


main: Get and display the list of ALL applications currently
      participating in common memory
main: following 2 applications are active in common memory...
    process_1
    process_2


          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
         | section corresponding to          |
         |                                   |
         |          NOTE  # 6                |
         |                                   |
         | Get and display the list of ALL   |
         | mailboxes in common memory.       |
         |                                   |
          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
main: Get and display the list of ALL mailboxes in common memory
main: following 1 mailboxes are declared in common memory...
    proc_1_mbx


         ----------------------------------
        ¦ section corresponding to          ¦
        ¦                                   ¦
        ¦         NOTE  # 7                 ¦
        ¦                                   ¦
        ¦ Get and display the list or       ¦
        ¦ READ/XREAD clients for mailbox    ¦
        ¦ 'proc_1_mbx'.                     ¦
        ¦                                   ¦
         ----------------------------------


main: Get and display the list of READ/XREAD clients for mailbox
      proc_1_mbx.
main: proc_1_mbx has following 2 READ/XREAD clients ...
    process_1
    process_2


         ----------------------------------
        ¦ section corresponding to          ¦
        ¦                                   ¦
        ¦         NOTE  # 8                 ¦
        ¦                                   ¦
        ¦ Get and display the list of       ¦
        ¦ mailboxes declared for READ/      ¦
        ¦ XREAD access by 'process_1'.      ¦
        ¦                                   ¦
         ----------------------------------


main: Get and display the list of mailboxes declared for
      READ/XREAD by process_1.
main: process_1 has following 1 READ mailboxes ...
    proc_1_mbx
```

```
 -----------------------------------
| section corresponding to          |
|                                   |
|         NOTE  # 9                 |
|                                   |
| Get and display the available     |
| common memory statistics for an   |
| application that is not in        |
| common memory.  Error will be     |
| generated.  Use 'process_3'.      |
|                                   |
 -----------------------------------


main: Get and display the available common memory statistics for
      an application that doesn't exist.  Expect an error report.
main: FATAL ERROR E_CM_FSMNOTINCM returned from cm_get_fsm_stats


 -----------------------------------
| section corresponding to          |
|                                   |
|         NOTE  # 10                |
|                                   |
| Get and display the available     |
| common memory statistics for      |
| process_1.                        |
|                                   |
 -----------------------------------


main: Get and display the available common memory statistics for
      an application that does exist.
    1 read mbx's
    0 reads performed
    last one at <none>
    mbx: <none>
    1 write mbx's
    5 writes performed
    last one at Sat Feb 27 19:51:35 1988
    mbx: proc_1_mbx
    1 updates on the update list
```

```
 ----------------------------------
| section corresponding to         |
|                                  |
|          NOTE  # 11              |
|                                  |
| Get and display the available    |
| common memory statistics for a   |
| nonexistent mailbox.             |
|                                  |
 ----------------------------------
```

```
main: Get and display the available common memory statistics for
      a mailbox that doesn't exist.
main: FATAL ERROR E_CM_MBXNOTDECL returned from cm_get_mbx_stats
```

```
 ----------------------------------
| section corresponding to         |
|                                  |
|          NOTE  # 12              |
|                                  |
| Get and display the available    |
| common memory statistics for     |
| 'proc_1_mbx'.                    |
|                                  |
 ----------------------------------
```

```
main: Get and display the available common memory statistics for
      a mailbox that does exist.
    mbxhandle 1, declared length 150, msg length 69
    2 readers w/ 5 read actions
       last by process_2 @ Sat Feb 27 19:51:35 1988
    1 writers w/ 5 write actions
       last by process_1 @ Sat Feb 27 19:51:35 1988
    xreader: <none>
    xwriter: <none>
```

```
 ----------------------------------
¦ section corresponding to         ¦
¦                                  ¦
¦         NOTE  # 13               ¦
¦                                  ¦
¦ Cause process_2 to undeclare all ¦
¦ its common memory mailbox        ¦
¦ connections.  Display the list   ¦
¦ or remaining common memory       ¦
¦ clients.  Only one client should ¦
¦ be left.                         ¦
¦                                  ¦
¦ Process_2 will use cm_undeclare  ¦
¦ to disconnect from the mailboxes ¦
¦                                  ¦
 ----------------------------------


main: Cause process_2 to undeclare its common memory mailbox
      connections.  Then display the common memory client list.
      Expect only one client.
main: following 1 applications are active in common memory...
    process_1


 ----------------------------------
¦ section corresponding to         ¦
¦                                  ¦
¦         NOTE  # 14               ¦
¦                                  ¦
¦ Cause process_1 to undeclare all ¦
¦ its common memory mailbox        ¦
¦ connections.  Display the list   ¦
¦ or remaining common memory       ¦
¦ clients.  Only one client should ¦
¦ be left.                         ¦
¦                                  ¦
¦ Process_1 will use cm_disc to    ¦
¦ disconnect from the mailboxes.   ¦
¦                                  ¦
 ----------------------------------


main: Cause process_1 to undeclare its common memory mailbox
      connections.  Then display the common memory client list.
      Expect zero clients.
main: no applications active in common memory
```

```
 ---------------------------------
| section corresponding to        |
|                                 |
|          NOTE # 15              |
|                                 |
| Retrieve and display the common |
| memory usage statistics.        |
|                                 |
 ---------------------------------
A total of 1 mbx's were created.  0 still active.
A total of 2 fsm's were clients of cm.  0 still active.
cm_declare failures: 0
    last by fsm:
    for mbx:
    at:

cm_declare successes: 2
    last by fsm: process_2
    for mbx: proc_1_mbx
    at: Sat Feb 27 19:50:55 1988

cm_undeclare failures: 0
    last by fsm:
    for mbx:
    at:

cm_undeclare successes: 1
    last by fsm: process_2
    for mbx: proc_1_mbx
    at: Sat Feb 27 19:51:35 1988

cm_write failures: 0
    last by fsm:
    for mbx:
    at:

cm_write successes: 5
    last by fsm: process_1
    for mbx: proc_1_mbx
    at: Sat Feb 27 19:51:35 1988

cm_read failures: 0
    last by fsm:
    for mbx:
    at:
```

```
cm_read successes: 5
    last by fsm: process_2
    for mbx: proc_1_mbx
    at: Sat Feb 27 19:51:35 1988

cm_ckmail failures: 0
    last by fsm:
    for mbx:
    at:

cm_ckmail successes: 9002
    last by fsm: process_2
    for mbx:
    at: Sat Feb 27 19:51:35 1988

cm_disc failures: 0
    last by fsm:
    for mbx:
    at:

cm_disc successes: 1
    last by fsm: process_1
    for mbx:
    at: Sat Feb 27 19:51:36 1988

cm_get_mbx_list failures: 0
    last by fsm:
    for mbx:
    at:

cm_get_mbx_list successes: 2
    last by fsm: process_1
    for mbx:
    at: Sat Feb 27 19:51:35 1988

cm_get_fsm_list failures: 0
    last by fsm:
    for mbx:
    at:

cm_get_fsm_list successes: 3
    last by fsm:
    for mbx:
    at: Sat Feb 27 19:51:35 1988

cm_get_cm_stats failures: 0
    last by fsm:
    for mbx:
    at:
```

```
cm_get_cm_stats successes: 0
    last by fsm:
    for mbx:
    at:

cm_get_fsm_stats failures: 1
    last by fsm: process_3
    for mbx:
    at: Sat Feb 27 19:51:35 1988

cm_get_fsm_stats successes: 1
    last by fsm: process_1
    for mbx:
    at: Sat Feb 27 19:51:35 1988

cm_get_mbx_stats failures: 1
    last by fsm: proc_3_mbx
    for mbx:
    at: Sat Feb 27 19:51:35 1988

cm_get_mbx_stats successes: 1
    last by fsm:
    for mbx: proc_1_mbx
    at: Sat Feb 27 19:51:35 1988
```

APPENDIX E

SOURCE CODE LISTINGS OF THE COMMON MEMORY PROGRAMS

The source code listings of the individual program files comprising the PC common memory have been placed at this appendix in the order shown below.

E.1. CM_CONST.H
E.2. CM_GLOBALS.H
E.3. CM_TYPES.H
E.4. CM_FUNCS.C
E.5. CM_UTILS.C
E.6. SFUNCS.C

Their pages are formatted and numbered differently from the rest of this document in order to provide a reference base for discussions of content.

```
/* cm_const.h  -  Contains all constant definitions        */

#define CM_VERSION             "1.0a"             /* chg here and in cm_types.h */
#define MAXMBXNAMELENGTH       32
#define MAXFSMNAMELENGTH       32
#define CM_MAXMBXNAMELENGTH    MAXMBXNAMELENGTH + 1 /* spc for trailing null */
#define CM_MAXFSMNAMELENGTH    MAXFSMNAMELENGTH + 1 /* spc for trailing null */

#ifndef TRUE
        #define TRUE           1
        #define FALSE          0
#endif


/* mbx access constants.  Combinations are achieved by ORing the values.
   For example, for both READ and WRITE access, the call to the appropriate
   CM routine would have CM_READ_ACCESS | CM_WRITE_ACCESS */
#define CM_READ_ACCESS         0x1        /* shared read access */
#define CM_XREAD_ACCESS        0x2        /* exclusive read access.  Use
                                             w/ caution, since it also
                                             precludes the network from
                                             accessing this mbx.  So,
                                             what's the value??  Future
                                             extension will allow declarer to
                                             generate a mbx access list  */
#define CM_WRITE_ACCESS        0x4        /* shared write access */
#define CM_XWRITE_ACCESS       0x8        /* exclusive write access */

/* status returns
        0x00 - 0x0F = normal returns.  This means that the requested action
                                        was performed.                   */
#define I_CM_OK                         0x0   /* normal success indicator */
#define I_CM_MBXACTV           0x2   /* mbx successfully undeclared, but
                                        other fsm's still connected.  In the
                                        case of cm_disc, this refers to the
                                        status of one or more mbx's.    */
#define I_CM_DUPMBX            0x4   /* mbx was previously declared by this
                                        fsm, and for similar access (READ/XREAD
                                        or WRITE/XWRITE).  If DECLARE was used to
                                        change acces from exclusive to non-exclusive
                                        (or vice-versa), then change was made.
                                        Additional entries are not made in the
                                        client list of the respective mailbox. */
#define I_CM_MOREDATA          0x5   /* cm_read was successfully performed.
                                        However the int variable pointed to by
                                        "nr_bytes" was non-zero when the call
                                        was made, and specified a number of bytes
                                        that was less than the number of bytes
                                        actually available in the mailbox.  Only
                                        the number of bytes specified in the int
                                        variable pointed to by "nr_bytes" was
```

```
                                        transferred to the user data are.  More
                                        data is actually available.    */

/* Errors greater than 0Fx are FATAL errors.  This means that the requested
   action was NOT performed. */


/*      0x10 - 0x1F = general errors (or errors common to multiple categories   */
#define E_CM_FATALERR           0x10    /* defines the start of FATAL ERROR range */
#define E_CM_INSUFFMEM          0x10    /* insufficient memory.  malloc failed */


/*      0x20 - 0x2F = mbx errors    */
#define E_CM_MBXERR             0x20    /* base value for mbx errors */
        /* the following 4  error codes must always be related according to   */
        /*       <err_code>  =  E_CM_MBXERR  +  CM_<type>_ACCESS              */
#define E_CM_MBXNOREAD          0x21    /* can't have READ - another has XREAD  */
#define E_CM_MBXNOXREAD         0x22    /* can't have XREAD - another fsm has
                                           either READ or XREAD */
#define E_CM_MBXNOWRITE         0x24    /* can't have WRITE - another has XWRITE */
#define E_CM_MBXNOXWRITE        0x28    /* can't have XWRITE - another fsm has
                                           either WRITE or XWRITE */
#define E_CM_MBXSIZE            0x29    /* invalid size, returned if:
                                           (1) negative size in cm_declare, or
                                           (2) byte count < 1 for cm_write, or
                                           (3) size doesn't match previously-
                                               declared size (for cm_declare), or
                                           (4) attempt to write nr_bytes
                                               greater than declared size */
#define E_CM_MBXACCESS          0x2a    /* invalid mbx access, returned if:
                                           (1) unrecognizable mbx access code
                                               supplied for a cm_declare or
                                               cm_undeclare, or
                                           (2) invalid list_type supplied to
                                               cm_get_fsm_list  */
#define E_CM_MBXACCBOTH         0x2b    /* can't declare both READ and XREAD or
                                           WRITE and XWRITE in the same mbx declare.
                                           However, you can later declare a mbx
                                           to be READ after having previously
                                           declared it XREAD ... this changes your
                                           access and lets other fsm's have
                                           access.                        */
#define E_CM_MBXNAME            0x2c    /* invalid mbx name - too long or none given  */
#define E_CM_MBXNOTDECL         0x2d    /* returned if :
                                           (1) fsm attempts to undeclare a mbx
                                               for which it is not a client for
                                               the respective access, or
                                           (2) fsm attempts to perform read or
                                               write actions w/ a mbx for which
                                                it is not a client for the
                                               respective access, or
```

```
                                          (3) fsm requests any cm action
                                              without being a client of cm, or
                                          (4) attempt to get info for a mbx
                                              that is not in common memory */

/*     0x30 - 0x3F = fsm errors    */
#define E_CM_FSMERR            0x30    /* base value for fsm errors */
#define E_CM_FSMNAME           0x31    /* invalid name - too long or none given */
#define E_CM_FSMNOTINCM        0x32    /* fsm not a common memory client */
```

```
/*  cm_globals.h  -  contains all global variable declarations */

#include <stdio.h>
#include "cm_const.h"                  /* include cm_const.h before cm_types.h */
#include "cm_types.h"

int cm_mbxhandle = 0;                  /* used by CMM in assigning mbxhandles */

fsm_rec  *cm_fsm_list = NULL;          /* list of active fsm's maintained by CMM */
mbx_rec  *cm_mbx_list = NULL;          /* list of declared mbx's maintained by CMM */


/* The following are initialized in function cm_ini found in cm_utils.h.
   The first 2 are EXTERN so they can be adjusted by the user application pgm.
   For testing and development purposes, they are currently initialized in
   routine cm_ini.   */

int CM_DEBUG_LEVEL = -1;               /* causes debugging statments to be displayed */
boolean CM_GET_STATS = TRUE;           /* tells cmm to gather some performance stats */
cm_activity_stats *cm_activity = NULL; /* tracks useage of cm routines */
cm_client_stats *cm_clients = NULL;    /* keeps count of mbx's and fsm's */
```

```
/* cm_types.h  -  definitions of types used in the common memory and
                  interface routines.

                  The source program that includes this file must also
                  include file "cm_const.h".

          ||||||||||||||||||||||||||||
          |  library version   1.0a  |   <--  chg here and in cm_const.h
          ||||||||||||||||||||||||||||

*/
#ifndef boolean
        typedef int boolean;
#endif



/*  some standardized definitions  */
typedef unsigned char    byte;
typedef long int         timestamp;           /* nr of seconds since 01 Jan '70 */



/* this is how mailbox variables are stored */
typedef struct mbx_stats {
        int  nr_fsms;                         /* nr of subscribers to this mbx */
        unsigned int  nr_accesses;            /* nr of times mbx accessed */
        timestamp when;                       /* time of last access */
        struct fsm_rec_type *who;             /* pts to last accessing fsm */
        };
typedef struct client_chain {
        struct fsm_rec_type *who;             /* ptr to subscribing fsm record */
        boolean exclusive;                    /* TRUE if fsm has XREAD or XWRITE */
        struct client_chain *next;            /* ptr to next fsm or NULL */
        };
typedef struct mbx_rec_type {
        char mbxname [CM_MAXMBXNAMELENGTH];
        int  handle;                          /* short (numeric) name for mbx */
        int  declaredlength;                  /* nr bytes declared */
        int  msglength;                       /* nr bytes currently stored */
        byte *data;                           /* ptr to actual data area */
        struct mbx_stats readers;             /* reader statistics */
        struct mbx_stats writers;             /* writer statistics */
        struct client_chain *readerlist;      /* linked list of fsm_rec ptrs */
        struct client_chain *writerlist;      /* linked list of fsm_rec ptrs */
        struct mbx_rec_type *next;            /* ptr to next mbx list entry */
        }mbx_rec;

/* this is how fsm variables are stored */
typedef struct mbx_decl_chain {
        struct mbx_rec_type *mbx;             /* ptr to subscribed mbx record */
```

```
        struct mbx_decl_chain *next;            /* ptr to next mbx or NULL */
        };
typedef struct fsm_stats {
        int  nr_mbxes;                          /* nr of mbxes declared for "this" access */
        unsigned int  nr_accesses;              /* ttl nr of accesses */
        timestamp when;                         /* time of last access */
        int  mbxhandle;                         /* last mbx accessed */
        struct mbx_decl_chain *mbx_list;        /* chain of declared mailboxes  for "this" access */
        };
typedef struct update_list {
        int mbxhandle;                          /* id's mbx that has been changed */
        struct update_list *next;               /* pts to next list entry */
        };
typedef struct fsm_rec_type {
        char fsmname[CM_MAXFSMNAMELENGTH];      /* name, null terminated */
        struct fsm_stats read;                  /* READ stats */
        struct fsm_stats write;                 /* WRITE stats */
        struct update_list *update_top;         /* FIFO list of subscribed READ mbxes
                that have been updated.  CMM will FREE them after fsm reads
                the mbxes.  Or, fsm can be handed this list w/ CM_SYNC call,
                and fsm must FREE the list when done.  If a subsequent update
                is made to a mbx that is already on the list, no additional
                list entry will be submitted.  */
        struct update_list *update_bot;         /* position of the update list where
                                                   new entries are made to maintain
                                                   the FIFO order. */
        int nr_updates;                         /* nr of update entries on the list */
        struct fsm_rec_type *next;              /* ptr to next fsm record or NULL */
        } fsm_rec;



/* this is how the cmm stores statistics */
typedef struct {
        char fsm[CM_MAXFSMNAMELENGTH];          /* name of fsm */
        char mbx[CM_MAXMBXNAMELENGTH];          /* name of mbx, if appropriate */
        timestamp when;
        unsigned int nr_times;                  /* nr times this service called */
        } base_stats;

typedef struct {
        base_stats success;                     /* for when the call succeeded */
        base_stats failure;                     /* for when it failed */
        } group_stats;

typedef struct {
        group_stats cm_declare,
                    cm_undeclare,
                    cm_write,
                    cm_read,
                    cm_ckmail,
```

```
                        cm_disc,
                        cm_get_mbx_list,
                        cm_get_fsm_list,
                        cm_get_cm_stats,
                        cm_get_fsm_stats,
                        cm_get_mbx_stats;
        } cm_activity_stats;                /* stats for cm calls */

typedef struct {
        unsigned int mbx_ttl,               /* ttl nr mbx's declared */
                     mbx_active,            /* nr mbx's currently active */
                     fsm_ttl,               /* ttl nr fsm's declared */
                     fsm_active;            /* nr fsm's currently active */
        } cm_client_stats;



/* the following structure is used to return a copy of the mbx list to the cm client */
typedef struct mbx_list_type {
        mbxname[CM_MAXMBXNAMELENGTH];       /* contains name of mbx */
        struct mbx_list_type *next;         /* pts to next, or is NULL at end */
        };
/* the following structure is used to return a copy of the fsm list to the cm client */
typedef struct fsm_list_type {
        fsmname[CM_MAXFSMNAMELENGTH];       /* contains name of fsm */
        struct fsm_list_type *next;         /* pts to next, or is NULL at end */
        };


/* the following is returned whenever fsm information is requested via
                              cm_get_fsm_stats
*/
typedef struct {
        int  nr_read_mbx;                   /* nr of read mbx declared */
        unsigned int  nr_reads;             /* nr of mbx reads performed */
        timestamp read_time;                /* time of last read */
        char mbx_read[CM_MAXMBXNAMELENGTH]; /* name of last mbx read, null-terminated*/
        int  nr_write_mbx;                  /* nr of write mbx declared */
        unsigned int  nr_writes;            /* nr of writes performed */
        timestamp write_time;               /* time of last write */
        char mbx_write[CM_MAXMBXNAMELENGTH]; /* name of last mbx written, null-terminated */
        int nr_updates;                     /* nr updates waiting in the update_list */
        }cm_fsm_stats_rec;

/* the following is returned whenever mbx information is requested via
                              cm_get_mbx_stats
*/
typedef struct {
        int  handle;                        /* short (numeric) name for mbx */
        int  declaredlength;                /* nr bytes declared */
        int  msglength;                     /* nr bytes currently stored */
```

```
        int  read_fsms;                          /* nr of READ subscribers */
        unsigned int  read_accesses;             /* nr of READ accesses */
        timestamp read_time;                     /* time of last read */
        char reader[CM_MAXFSMNAMELENGTH];        /* name of last reader fsm */
        int  write_fsms;                         /* nr of WRITE subscribers */
        unsigned int  write_accesses;            /* nr of WRITE accesses */
        timestamp write_time;                    /* time of last write */
        char writer[CM_MAXFSMNAMELENGTH];        /* name of last writer fsm */
        char xreader[CM_MAXFSMNAMELENGTH];       /* name of xreader fsm */
        char xwriter[CM_MAXFSMNAMELENGTH];       /* name of xwriter fsm */
        }cm_mbx_stats_rec;
```

```
/*   cm_funcs.c    -   contains the common memory interface routines
                       used by client processes.


     The following section describes the purpose of each interface function.
     The argument list for each function is described in detail.  Each
     function returns an integer status value that correlates with what the
     function is to perform (hence the 'int' before each function name).

     The list of all potential status values that this family of functions
     can return, and their significance, is provided in file cm_const.h


int cm_declare(fsm              - INPUT
                                  TYPE char *fsm
                                  fsm name string.  String must
                                  be null-terminated, and must be <= 32
                                  characters in length (excluding NULL).
               mbxname          - INPUT
                                  TYPE char *mbxname
                                  mbx name string.  String must
                                  be null-terminated, and must be <= 32
                                  characters in length (excluding NULL).
               mbxsize          - INPUT
                                  TYPE int mbxsize
                                  max size of mbx to be created.
               mbxaccess        - INPUT
                                  TYPE int mbxaccess
                                  can be READ | WRITE | XREAD | XWRITE,
                                  but not both of the same kind in the same
                                  declaration.  The associated constants are
                                  listed in cm_const.h.
               mbxhandle        - OUTPUT
                                  TYPE int *mbxhandle
                                  Value returned in the int variable is to be
                                  used as a shorthand reference for mbx for
                                  calls to all other cm routines.
               )
      PURPOSE: Routine creates the necessary fsm, mbx, and mbx client
               structures within common memory to support future mailbox
               manipulations by the declaring fsm.

               NOTE: When you cm_declare to an existing mbx for READ or XREAD,
               the cmm will NOT place an entry into your "update list" for
               that mbx.  The purpose of the update list is to indicate that
               the mbx contents have been written SINCE the time of your
               cm_declare or cm_read.  It is assumed that you will perform an
               initial cm_read as a matter of course (if desired).

      RETURNS: status, as id'd in cm_const.h
```

```
int cm_undeclare(fsm            - INPUT
                                  TYPE char *fsm
                                  fsm name string.  String must
                                  be null-terminated, and must be <= 32
                                  characters in length (excluding NULL).
                 mbxaccess      - INPUT
                                  TYPE int mbxaccess
                                  is READ | WRITE | XREAD | XWRITE,
                                  but not both of the same kind in the same
                                  declaration.  The access constants are
                                  listed in cm_const.h.
                 mbxhandle      - INPUT
                                  TYPE int *mbxhandle
                                  Variable value was initially set by
                                  cm_declare and is used as a fast way to
                                  reference a specific mbx.  Although this does
                                  not need to be a pointer, it is specified as
                                  such for compatability with cm_declare
                                  (which requires it) and other common memory
                                  routines.
                 )
      PURPOSE: Routine is used to remove an fsm from a particular mbx's
               client list for the specified access.  More than
               one access type may be specified at each call, subject to
               the access rules identified in the cm_declare section.
               If the action results in a mbx without any clients, the
               mailbox is deleted and the space returned to the operating
               system.  Likewise, if the action results in an fsm that has
               no other mbx's declared, that fsm is removed as a cm client.
               If the mbx was identified on the update list, then the
               update list entry will be purged.

      RETURNS: status, as id'd in cm_const.h


int cm_write (fsm               - INPUT
                                  TYPE char *fsm
                                  fsm name string.  String must
                                  be null-terminated, and must be <= 32
                                  characters in length (excluding NULL).
               mbxhandle        - INPUT
                                  TYPE int *mbxhandle
                                  Variable value was initially set by
                                  cm_declare and is used as a fast way to
                                  reference a specific mbx.  Although this does
                                  not need to be a pointer, it is specified as
                                  such for compatability with cm_declare
                                  (which requires it) and other common memory
                                  routines.
               usr_data         - INPUT
                                  TYPE byte *usr_data
```

```
                                 points to user data area from which
                                 bytes are to be transferred.
              nr_bytes         - INPUT
                                 TYPE int *nrbytes
                                 int variable contains nr of bytes to be
                                 transferred from user data area to common
                                 memory.  Although this does not need to be
                                 a pointer, it is specified as such for
                                 compatability with cm_read, which requires it.
              )
      PURPOSE: Routine is used to transfer the specified number of bytes
               from the user data area to the common memory mailbox.  All
               fsm's that have declared READ (or XREAD) access to this mbx
               will have an entry made on their update list.

      RETURNS: status, as id'd in cm_const.h


int cm_read  (fsm              - INPUT
                                 TYPE char *fsm
                                 fsm name string.  String must
                                 be null-terminated, and must be <= 32
                                 characters in length (excluding NULL).
              mbxhandle        - INPUT
                                 TYPE int *mbxhandle
                                 Variable value was initially set by
                                 cm_declare and is used as a fast way to
                                 reference a specific mbx.  Although this does
                                 not need to be a pointer, it is specified as
                                 such for compatability with cm_declare
                                 (which requires it) and other common memory
                                 routines.
              usr_data         - INPUT
                                 TYPE byte *usr_data
                                 points to user data area to which
                                 bytes are to be transferred.
              nr_bytes         - INPUT/OUTPUT
                                 TYPE int *nrbytes
                                 When cm_read is called, if :
                                    (1) the int variable = 0, then all data
                                        bytes are transferred from the mailbox
                                        to the user's data area.
                                    (2) the int variable is not equal to 0,
                                        the nr of bytes transferred will be
                                        min(nr_bytes, nr_bytes_in_mbx).
                                 Upon return, the variable pointed to by
                                 nr_bytes will contain the actual number of
                                 bytes transferred.  If fewer bytes are
                                 transferred to the user data area than are
                                 available in the mailbox, an "information-
                                 only" status of I_CM_MOREDATA is returned
```

```
                                    to alert the user who may have inadvertently
                                    called cm_read without clearing the variable
                                    pointed to by nr_bytes.

                                    It is the user's responsibility to make
                                    sure that the data area is large enough
                                    to contain the mailgram.
                  )
        PURPOSE: Routine is used to transfer the specified number of bytes
                 to the user data area from the common memory mailbox.
                 It is recommended that cm_ckmail be used together with
                 cm_read to minimize common memory accesses.

                 If an entry for this mbx exists on the update list of this fsm,
                 it is removed at completion of the cm_read operation.

        RETURNS: status, as id'd in cm_const.h



int cm_ckmail(fsm                 - INPUT
                                    TYPE  char *fsm
                                    fsm name string.  String must
                                    be null-terminated, and must be <= 32
                                    characters in length (excluding NULL).
              list_ptr            - INPUT
                                    TYPE  struct update_list **list_ptr;
                                    If an update list exists for this fsm, cm_ckmail
                                    will return a ptr to the top of the
                                    update list in this location.  If none
                                    exists, cm_ckmail will return NULL.
              nr_entries          - INPUT
                                    TYPE  int *nr_entries;
                                    If an update list exists, the int variable
                                    will contain the number of entries in the
                                    update list;  else, it will contain ZERO.
              )

        PURPOSE: For each fsm that is a READer client of common memory, the
                 common memory manager creates and maintains a list of those
                 mailboxes that have changed since the last time the fsm read
                 them (ie, an update list).  Whenever an fsm writes to a
                 common memory mbx, the common memory manager posts an entry
                 on this "update" list.  If an entry already exists for a
                 changed mbx, no additional entry is made.  The list is
                 maintained in first-in-first-out (FIFO) order.

                 Entries are removed from this list whenever an fsm calls
                 cm_read for the respective mailbox.  Alternately, an fsm may
                 call cm_ckmail.  If an update list exists, cm_ckmail returns a
                 pointer to the top of the list and releases the list to the
```

```
               fsm.  If no update list exists, cm_ckmail returns NULL.  (The
               common memory manager will start a new list when the next
               mbx update arrives.)

               Once the list is released to the fsm, it is the responsibility
               of the fsm to FREE the memory allocated for the list.  The
               fsm may do so itself, or it may call cm_free_update_list,
               passing it a pointer to the top of the list.

               Using the update list, the fsm can now perform sequential
               cm_read operations and only access those mailboxes that have
               changed since the last read operation.

      RETURNS: status, as id'd in cm_const.h


int cm_disc (fsm                - INPUT
                                  TYPE char *fsm
                                  fsm name string.  String must
                                  be null-terminated, and must be <= 32
                                  characters in length (excluding NULL).
          )

      PURPOSE: Provides a short-cut method for a process to undeclare all of
               its mailboxes at one time.  All data structures within common
               memory that are associated with that fsm are freed.  The fsm
               must issue a cm_declare before it can again access common
               memory variables.

      RETURNS: status, as id'd in cm_const.h



int cm_get_fsm_list(
               mbxname          - INPUT
                                  TYPE  char *mbxname
                                  If NULL, this routine will return, through
                                     fsm_list_ptr, the list of all fsm names
                                     known to the common memory manager.
                                     Arg "list_type" has no effect.
                                  If not NULL, it must point to a mbx name.
                                     This routine will return a list of all fsm's
                                     that are a client of that specific mbx. The
                                     arg "list_type" is used to qualify whether
                                     the caller wants the list of READer clients
                                     or the list of WRITer clients.
               list_type        - INPUT
                                  TYPE  char
                                  May only have the values 'R' (READ) and
                                  'W' (WRITE) when *mbxname is non-NULL.  If
                                  *mbxname is NULL, list_type is ignored.
```

```
               fsm_list_ptr    - OUTPUT
                                 TYPE  struct fsm_list_type **fsm_list_ptr
                                 This routine will create a linked list of fsm
                                 names and return a ptr to the top of the
                                 list if any fsm's exists, or NULL if none
                                 exist.  It is the user's responsibility
                                 to free this list when it is no longer needed.
               int_ptr         - OUTPUT
                                 TYPE int *int_ptr
                                 Upon return, the int variable will equal the
                                 number of entries in the list.
               )

       PURPOSE: This routine allows any fsm to determine what fsm's are
                currently active in common memory.  Using the mbxname and
                list_type appropriately, the caller can retrieve the list of
                all fsm's, or only the list of clients (read or write) for a
                specific mbx.  The information that is returned on the list
                can be used to solicit other fsm (and, indirectly, mbx)
                statistics.

       RETURNS: status, as id'd in cm_const.h



int cm_get_mbx_list(
               fsmname         - INPUT
                                 TYPE  char *fsmname
                                 fsm name string.  String must
                                 be null-terminated, and must be <= 32
                                 characters in length (excluding NULL).
                                 If NULL, this routine will return, through
                                    mbx_list_ptr, the list of all mbx names
                                    known to the common memory manager.
                                    Arg "list_type" has no effect.
                                 If not NULL, it must point to a fsm name.
                                    This routine will return a list of all mbx's
                                    that are a declared by that fsm. The
                                    arg "list_type" is used to qualify whether
                                    the caller wants the list of READer mbx's
                                    or the list of WRITEr mbx's.
               list_type       - INPUT
                                 TYPE  char
                                 May only have the values 'R' (READ) and
                                 'W' (WRITE) when *fsmname is non-NULL.  If
                                 *fsmname is NULL, list_type is ignored.
               mbx_list_ptr    - OUTPUT
                                 TYPE  struct mbx_list_type **mbx_list_ptr
                                 This routine will create a linked list of mbx
                                 names and return a ptr to the top of the
                                 list if any mbx's exists, or NULL if none
```

```
                           exist.  It is the user's responsibility
                           to free this list when it is no longer needed.
          int_ptr        - INPUT
                           TPYE  int *int_ptr
                           Upon return, the int variable will equal the
                           number of entries in the list.
          )

     PURPOSE: This routine allows any fsm to determine what mbx's are
              currently active in common memory.  The information that is
              returned on the list can be used to solicit other cm fsm
              (and, indirectly, mbx) statistics.  Using the fsmname and
              list_type appropriately, the caller can retrieve the list of
              all mbx's in common memory, or only the list of mbx's (read
              or write) for a specific fsm.

     RETURNS: status, as id'd in cm_const.h


int cm_get_cm_stats(
          activity_ptr   - INPUT
                           TYPE  cm_activity_stats *activity_ptr
                           Points to user-allocated data area of
                           appropriate size.  This routine will copy
                           the activity statistics into that data area.
                           It has been implemented in this fashion to
                           minimize malloc and free operations, since it
                           is assumed the user will want this information
                           more than once.

          client_ptr     - INPUT
                           TYPE  cm_client_stats *client_ptr;
                           Points to user-allocated data area of
                           appropriate size.  This routine will copy
                           the client statistics into that data area.
                           It has been implemented in this fashion to
                           minimize malloc and free operations, since it
                           is assumed the user will want this information
                           more than once.
          )

     PURPOSE: This routine provides cm operating statistics.
              Since the size of the statistics areas is static,
              the user must provide pointers to space in the
              data area into which the statistics will be
              copied.  This avoids malloc overhead in case the
              user wishes to call this routine multiple times.

     RETURNS: status, as id'd in cm_const.h
```

```
int cm_get_fsm_stats(
                fsm             - INPUT
                                  TYPE  char *fsm
                                  fsm name string.  String must
                                  be null-terminated, and must be <= 32
                                  characters in length (excluding NULL).
                ptr             - INPUT
                                  TYPE  cm_fsm_stats_rec *ptr
                )

        PURPOSE: This routine returns the common memory statistics
                 for the specified fsm, as identified in cm_fsm_stats_rec.
                 Since the size of the statistics area is static,
                 the user must provide a pointer to space in the
                 user data area into which the statistics will be
                 copied.  This avoids malloc overhead in case the
                 user wishes to call this routine multiple times.

        RETURNS: status, as id'd in cm_const.h


int cm_get_mbx_stats(
                mbx             - INPUT
                                  TYPE  char *mbx
                                  mbx name string.  String must
                                  be null-terminated, and must be <= 32
                                  characters in length (excluding NULL).
                ptr             - INPUT
                                  TYPE  cm_mbx_stats_rec *ptr
                )

        PURPOSE: This routine returns the common memory statistics
                 for the specified mbx, as identified in cm_mbx_stats_rec.
                 Since the size of the statistics area is static,
                 the user must provide a pointer to space in the
                 user data area into which the statistics will be
                 copied.  This avoids malloc overhead in case the
                 user wishes to call this routine multiple times.

        RETURNS: status, as id'd in cm_const.h


*/

#include "cm_utils.c"


/*  ||||||||||||||||||||||    Declare a mailbox in common memory.  This
```

```
 * |    cm_declare    |   can result in a connection to an already
 * ||||||||||||||||||||   existing mbx, or creation of a new one.
 */
int cm_declare (fsm, mbxname, mbxsize, mbxaccess, mbxhandle)
char *fsm, *mbxname;
int mbxsize, mbxaccess, *mbxhandle;
{ int return_status;
  byte *data_ptr;
  fsm_rec *tmp_fsm_rec;
  mbx_rec *tmp_mbx_rec;
  struct client_chain *tmp_client_rec;
  boolean new_fsm, new_mbx;

  if (!cm_activity) cm_ini();                /* init cmm structures */
  new_fsm = FALSE;
  new_mbx = FALSE;

  if (return_status = cm_validate_fsm(fsm)) {
     log_status(&cm_activity->cm_declare.failure,NULL,NULL);
     return(return_status);                  /* fsm name error */
  }
  if (return_status = cm_validate_mbx(mbxname,mbxsize)) {
     log_status(&cm_activity->cm_declare.failure,fsm,NULL);
     return(return_status);                  /* mbx error */
  }
  if (return_status = cm_validate_access(mbxaccess)) {
     log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
     return(return_status);                  /* invalid access */
  }

  if (tmp_fsm_rec = cm_fsm_find(fsm)) {
     eprintf(9,"cm_declare:  fsm already known to common memory\n");
  }else {                                    /* init new rec for this fsm */
     eprintf(6,"cm_declare: add '%s' as cm client\n",fsm);
     tmp_fsm_rec = (fsm_rec *) malloc (sizeof(fsm_rec));
     if (tmp_fsm_rec == NULL)  {
        log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
        return(E_CM_INSUFFMEM);
     }
     bclr(tmp_fsm_rec,sizeof(fsm_rec));
     strcpy(tmp_fsm_rec->fsmname,fsm);       /* save fsm name */
     new_fsm = TRUE;
  }

  if (tmp_mbx_rec = cm_mbx_find(mbxname)){   /* ck if mbx already exists */
     eprintf(9,"cm_declare: This mbx is already known to common memory\n");
     *mbxhandle = tmp_mbx_rec->handle;
  }else {                                    /* init new rec for this mbx */
     eprintf(6,"cm_declare: add '%s' as new cm mbx\n", mbxname);
     tmp_mbx_rec = (mbx_rec *) malloc (sizeof(mbx_rec));
     if (tmp_mbx_rec == NULL)  {
```

```
          log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
         return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,E_CM_INSUFFMEM));
      }
      bclr(tmp_mbx_rec,sizeof(mbx_rec));
      strcpy(tmp_mbx_rec->mbxname,mbxname);     /* save fsm name */
      *mbxhandle = ++cm_mbxhandle;
      tmp_mbx_rec->handle = cm_mbxhandle;
      tmp_mbx_rec->declaredlength = mbxsize;
      new_mbx = TRUE;
      data_ptr = (byte *) malloc (mbxsize);     /* spc available for data mbx ? */
      if (data_ptr == NULL)  {
         log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
         return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,E_CM_INSUFFMEM));
      }
      bclr(data_ptr,mbxsize);
      tmp_mbx_rec->data = data_ptr;
   }

   if (mbxaccess & CM_XREAD_ACCESS) {            /* ck if another fsm has READ or XREAD */
      if ((tmp_mbx_rec->readers.nr_fsms > 1)   /* DIE if more than 1 reader */
          || ((tmp_mbx_rec->readerlist)          /* DIE if I'm not the only reader */
              && (find_mbx_client(tmp_mbx_rec->readerlist, tmp_fsm_rec) == NULL)))  {
        log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
        return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,E_CM_MBXNOXREAD));
      }else
        if (tmp_client_rec = find_mbx_client(tmp_mbx_rec->readerlist, tmp_fsm_rec)){
           tmp_client_rec->exclusive = TRUE;      /* reset from READ to XREAD */
           return_status = I_CM_DUPMBX;           /* let user know he was already a client for
                                                   * this mbx w/ either READ/XREAD access */
        }else {
           ++(tmp_mbx_rec->readers.nr_fsms);      /* NEW, so increment nr of readers */
           ++(tmp_fsm_rec->read.nr_mbxes);        /* incr nr of mbxes this fsm reads */
           return_status = add_mbx_client (tmp_fsm_rec, &tmp_mbx_rec->readerlist, TRUE);
           if (return_status >= E_CM_FATALERR)
              return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,return_status));
           return_status = add_fsm_mbx (&tmp_fsm_rec->read.mbx_list, tmp_mbx_rec); /* add mbx to fsm's list */
           if (return_status >= E_CM_FATALERR)  {
              log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
              return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,return_status));
           }
        }
   }else
      if (mbxaccess & CM_READ_ACCESS)            /* ck if another fsm has XREAD */
         if ((tmp_mbx_rec->readers.nr_fsms == 1)      /* DIE if someone has XREAD */
             && ((tmp_mbx_rec->readerlist->exclusive)  /* and it isn't ME */
                 && (tmp_mbx_rec->readerlist->who != tmp_fsm_rec)))  {
           log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
           return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,E_CM_MBXNOREAD));
         }else
           if (tmp_client_rec = find_mbx_client(tmp_mbx_rec->readerlist, tmp_fsm_rec)){
              tmp_client_rec->exclusive = FALSE;    /* reset from XREAD to READ */
```

```
            return_status = I_CM_DUPMBX;     /* let user know he was already a client for
                                             /* this mbx w/ either READ/XREAD access */
         }else {
            ++(tmp_mbx_rec->readers.nr_fsms);/* NEW, so increment nr of writers */
            ++(tmp_fsm_rec->read.nr_mbxes);  /* incr nr of mbxes this fsm reads */
            return_status = add_mbx_client (tmp_fsm_rec, &tmp_mbx_rec->readerlist, FALSE);
            if (return_status >= E_CM_FATALERR)  {
               log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
               return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,return_status));
            }
            return_status = add_fsm_mbx (&tmp_fsm_rec->read.mbx_list, tmp_mbx_rec);  /* add mbx to fsm's list */
            if (return_status >= E_CM_FATALERR)  {
               log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
               return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,return_status));
            }
         }


   if (mbxaccess & CM_XWRITE_ACCESS)           /* ck if another fsm has WRITE or XWRITE */
      if ((tmp_mbx_rec->writers.nr_fsms > 1)   /* DIE if more than 1 writer */
          || ((tmp_mbx_rec->writerlist)        /* DIE if I'm not the only writer */
             && (find_mbx_client(tmp_mbx_rec->writerlist, tmp_fsm_rec) == NULL)))  {
         log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
         return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,E_CM_MBXNOXWRITE));
      }else                                     /*  allowed to have XWRITE */
         if (tmp_client_rec = find_mbx_client(tmp_mbx_rec->writerlist, tmp_fsm_rec)) {
            tmp_client_rec->exclusive = TRUE;  /* reset from WRITE to XWRITE */
            return_status = I_CM_DUPMBX;       /* let user know he was already a client for
                                                * this mbx w/ either WRITE/XWRITE access */
         }else {
            ++(tmp_mbx_rec->writers.nr_fsms);  /* NEW, so increment nr of writers */
            ++(tmp_fsm_rec->write.nr_mbxes);   /* incr nr of mbxes this fsm writes to */
            return_status = add_mbx_client (tmp_fsm_rec, &tmp_mbx_rec->writerlist, TRUE);
            if (return_status >= E_CM_FATALERR) {
               log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
               return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,return_status));
            }
            return_status = add_fsm_mbx (&tmp_fsm_rec->write.mbx_list, tmp_mbx_rec);  /* add mbx to fsm's list */
            if (return_status >= E_CM_FATALERR)  {
               log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
               return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,return_status));
            }
         }
   else
      if (mbxaccess & CM_WRITE_ACCESS)            /* ck if other fsm has XWRITE */
         if ((tmp_mbx_rec->writers.nr_fsms == 1)  /* DIE if someone has XWRITE */
             && ((tmp_mbx_rec->writerlist->exclusive) /* and it isn't ME */
                && (tmp_mbx_rec->writerlist->who != tmp_fsm_rec)))  {
            log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
            return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,E_CM_MBXNOWRITE));
         }else                                     /* only add client if not already on list */
```

```
            if (tmp_client_rec = find_mbx_client(tmp_mbx_rec->writerlist, tmp_fsm_rec)){
               tmp_client_rec->exclusive = FALSE; /* reset from XWRITE to WRITE */
               return_status = I_CM_DUPMBX;       /* let user know he was already a client for
                                                   * this mbx w/ either WRITE/XWRITE access */
            }else {
               ++(tmp_mbx_rec->writers.nr_fsms);  /* NEW, so increment nr of writers */
               ++(tmp_fsm_rec->write.nr_mbxes);   /* incr nr of mbxes this fsm writes to */
               return_status = add_mbx_client (tmp_fsm_rec, &tmp_mbx_rec->writerlist, FALSE);
               if (return_status >= E_CM_FATALERR) {
                  log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
                  return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,return_status));
               }
               return_status = add_fsm_mbx (&tmp_fsm_rec->write.mbx_list, tmp_mbx_rec);  /* add mbx to fsm's list
               if (return_status >= E_CM_FATALERR)  {
                  log_status(&cm_activity->cm_declare.failure,fsm,mbxname);
                  return(clear_declare(new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec,return_status));
               }
            }


   if (new_fsm)                                   /* insert new fsm onto list */
      if (cm_fsm_list == NULL)
         cm_fsm_list = tmp_fsm_rec;
      else {
         tmp_fsm_rec->next = cm_fsm_list;         /* insert at front of list */
         cm_fsm_list = tmp_fsm_rec;
      }

   if (new_mbx)                                   /* insert new mbx onto list */
      if (cm_mbx_list == NULL)
         cm_mbx_list = tmp_mbx_rec;
      else {
         tmp_mbx_rec->next = cm_mbx_list;         /* insert at front of list */
         cm_mbx_list = tmp_mbx_rec;
      }
   log_status(&cm_activity->cm_declare.success,fsm,mbxname);
   if (new_mbx)  {
      ++(cm_clients->mbx_ttl);
      ++(cm_clients->mbx_active);
   }
   if (new_fsm)  {
      ++(cm_clients->fsm_ttl);
      ++(cm_clients->fsm_active);
   }
   return (return_status);
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   Remove a client from a mbx client list, but
 *  ¦    cm_undeclare    ¦   only for the specified access.
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
```

```
 */
int cm_undeclare (fsm, mbxaccess, mbxhandle)
char *fsm;
int mbxaccess, *mbxhandle;
{ int i, return_status;
  fsm_rec *tmp_fsm_rec;
  mbx_rec *tmp_mbx_rec;
  struct client_chain *tmp_client_rec;

  eprintf(9,"cm_undeclare: fsm %s, access %d, mbxhandle %d\n",
          fsm, mbxaccess, *mbxhandle);
  if (!cm_activity) cm_ini();                          /* init cmm structures */
  if ((!(tmp_fsm_rec = cm_fsm_find(fsm)))              /* fsm active in cm ? */
     || (!(tmp_mbx_rec = cm_mbxhandle_find(*mbxhandle)))) {  /* mbxhandle in cm ? */
    log_status(&cm_activity->cm_undeclare.failure,NULL,NULL);
    return(E_CM_MBXNOTDECL);
  }
  if (return_status = cm_validate_access(mbxaccess)) {
    log_status(&cm_activity->cm_undeclare.failure,fsm,NULL);
    return(return_status);                             /* invalid access */
  }

  eprintf(9,"cm_undeclare: passed validation tests\n");
  if (mbxaccess & (CM_READ_ACCESS | CM_XREAD_ACCESS))      /* is fsm a READ client ? */
    if (tmp_client_rec = find_mbx_client(tmp_mbx_rec->readerlist,tmp_fsm_rec)) {
      eprintf(9,"cm_undeclare: found client in readerlist\n");
      if ((mbxaccess & CM_READ_ACCESS) && (tmp_client_rec->exclusive)) {
        log_status(&cm_activity->cm_undeclare.failure,fsm,tmp_mbx_rec->mbxname);
        return(E_CM_MBXNOTDECL);            /* DIE if undeclaring READ but have XREAD */
      }
      if ((mbxaccess & CM_XREAD_ACCESS) && (!tmp_client_rec->exclusive)) {
        log_status(&cm_activity->cm_undeclare.failure,fsm,tmp_mbx_rec->mbxname);
        return(E_CM_MBXNOTDECL);            /* DIE if undeclaring XREAD but have READ */
      }
      del_update_rec(tmp_fsm_rec,tmp_mbx_rec->handle); /* remove update notification, if on list */
      del_mbx_client(tmp_client_rec,&(tmp_mbx_rec->readerlist),&(tmp_mbx_rec->readers.nr_fsms));
      del_fsm_mbx_entry(&tmp_fsm_rec->read.mbx_list,tmp_mbx_rec);
      --(tmp_fsm_rec->read.nr_mbxes);      /* decrease nr of mbxes this fsm reads */
      eprintf(7,"cm_undeclare: deleted READ/XREAD entry\n");
    }else {
      log_status(&cm_activity->cm_undeclare.failure,fsm,tmp_mbx_rec->mbxname);
      return(E_CM_MBXNOTDECL);
    }

  eprintf(9,"cm_undeclare: finished READ/XREAD checks\n");
  if (mbxaccess & (CM_WRITE_ACCESS | CM_XWRITE_ACCESS))   /* is fsm a WRITE client ? */
    if (tmp_client_rec = find_mbx_client(tmp_mbx_rec->writerlist,tmp_fsm_rec)) {
      eprintf(9,"cm_undeclare: found client in writerlist\n");
      if ((mbxaccess & CM_WRITE_ACCESS) && (tmp_client_rec->exclusive)) {
        log_status(&cm_activity->cm_undeclare.failure,fsm,tmp_mbx_rec->mbxname);
        return(E_CM_MBXNOTDECL);            /* DIE if undeclaring WRITE but have XWRITE */
```

```
         }
         if ((mbxaccess & CM_XWRITE_ACCESS) && (!tmp_client_rec->exclusive))  {
            log_status(&cm_activity->cm_undeclare.failure,fsm,tmp_mbx_rec->mbxname);
            return(E_CM_MBXNOTDECL);          /* DIE if undeclaring XWRITE but have WRITE */
         }
         del_mbx_client(tmp_client_rec,&(tmp_mbx_rec->writerlist),&(tmp_mbx_rec->writers.nr_fsms));
         del_fsm_mbx_entry(&tmp_fsm_rec->write.mbx_list,tmp_mbx_rec);
         --(tmp_fsm_rec->write.nr_mbxes);      /* decrease nr of mbxes this fsm writes */
         eprintf(7,"cm_undeclare: deleted WRITE/XWRITE entry\n");
      }else {
         log_status(&cm_activity->cm_undeclare.failure,fsm,tmp_mbx_rec->mbxname);
         return(E_CM_MBXNOTDECL);
      }

   /* if mbx has no other clients, remove it from cm_mbx_list */
   if ((!tmp_mbx_rec->readerlist)  &&  (!tmp_mbx_rec->writerlist)) {
      --(cm_clients->mbx_active);
      cm_mbx_del(tmp_mbx_rec);
   }

   /* if the fsm has no other mailboxes declared, remove it from cm_fsm_list */
   if ((!tmp_fsm_rec->read.mbx_list)  &&  (!tmp_fsm_rec->write.mbx_list)) {
      --(cm_clients->fsm_active);
      cm_fsm_del(tmp_fsm_rec);
   }
   log_status(&cm_activity->cm_undeclare.success,fsm,tmp_mbx_rec->mbxname);
   return(I_CM_OK);
 }


 /*  ||||||||||||||||||||||    Transfer the specified number of bytes from the
  *  |      cm_write      |    the user data area to the common memory mbx.
  *  ||||||||||||||||||||||
  */
 int cm_write (fsm, mbxhandle, usr_data, nrbytes)
 char *fsm;
 byte *usr_data;
 int *mbxhandle, *nrbytes;
 { fsm_rec *tmp_fsm_rec;
   mbx_rec *tmp_mbx_rec;
   timestamp when;

   eprintf(5,"cm_write: at entry, fsm = %s, mbxhandle = %d, nrbytes = %d\n",
            fsm, *mbxhandle, *nrbytes);

   if (!cm_activity) cm_ini();                   /* init cmm structures */
   if ((!(tmp_fsm_rec = cm_fsm_find(fsm))      /* fsm active in cm ? */
       || (!(tmp_mbx_rec = cm_mbxhandle_find(*mbxhandle)))){/* mbxhandle in cm ? */
      log_status(&cm_activity->cm_write.failure,NULL,NULL);
      return(E_CM_MBXNOTDECL);
   }
```

```
  if (!find_mbx_client(tmp_mbx_rec->writerlist,tmp_fsm_rec)) {
     log_status(&cm_activity->cm_write.failure,fsm,tmp_mbx_rec->mbxname);
     return(E_CM_MBXNOTDECL);                 /* not a WRITE/XWRITE client of this mbx */
  }
  if (*nrbytes > tmp_mbx_rec->declaredlength) {
     log_status(&cm_activity->cm_write.failure,fsm,tmp_mbx_rec->mbxname);
     return(E_CM_MBXSIZE);                    /* attempt to write more than mbx space allows */
  }
  eprintf(5,"cm_write: passed validation tests\n");
  if (*nrbytes)                              /* transfer data to mbx */
     bcpy(usr_data,tmp_mbx_rec->data,*nrbytes);
  tmp_mbx_rec->msglength = *nrbytes;          /* nr bytes in msg */
  ++(tmp_mbx_rec->writers.nr_accesses);
  tmp_mbx_rec->writers.who = tmp_fsm_rec;
  time(&when);                                /* timestamp */
  tmp_mbx_rec->writers.when = when;
  tmp_fsm_rec->write.when = when;
  ++(tmp_fsm_rec->write.nr_accesses);
  tmp_fsm_rec->write.mbxhandle = *mbxhandle;

  /* notify all READ clients of this mbx that a change has occured in the mbx */
  notify_clients(tmp_mbx_rec->readerlist,*mbxhandle);
  log_status(&cm_activity->cm_write.success,fsm,tmp_mbx_rec->mbxname);
  return(I_CM_OK);
}



/*  !!!!!!!!!!!!!!!!!!!!!!   Transfer the specified number of bytes from the
 *  |      cm_read       |   the common memory mbx to the user data area.
 *  !!!!!!!!!!!!!!!!!!!!!!
 */
int cm_read (fsm, mbxhandle, usr_data, nrbytes)
char *fsm;
byte *usr_data;
int *mbxhandle, *nrbytes;
{ fsm_rec *tmp_fsm_rec;
  mbx_rec *tmp_mbx_rec;
  timestamp when;
  int return_status;

  eprintf(5,"cm_read: at entry, fsm = %s, mbxhandle = %d, nrbytes = %d\n",
          fsm, *mbxhandle, *nrbytes);
  if (!cm_activity) cm_ini();                 /* initialize cmm structures */
  return_status = I_CM_OK;                    /* assume clean exit */
  if ((!(tmp_fsm_rec = cm_fsm_find(fsm)))     /* fsm active in cm ? */
      || (!(tmp_mbx_rec = cm_mbxhandle_find(*mbxhandle)))) { /* mbxhandle in cm ? */
     log_status(&cm_activity->cm_read.failure,NULL,NULL);
     return(E_CM_MBXNOTDECL);
  }
  if (!find_mbx_client(tmp_mbx_rec->readerlist,tmp_fsm_rec)) {
```

```
      log_status(&cm_activity->cm_read.failure,fsm,tmp_mbx_rec->mbxname);
      return(E_CM_MBXNOTDECL);                /* not a READ/XREAD client of this mbx */
   }
   if ((*nrbytes < tmp_mbx_rec->msglength) && (*nrbytes))  {
      log_status(&cm_activity->cm_read.failure,fsm,tmp_mbx_rec->mbxname);
      return_status = I_CM_MOREDATA;          /* user wants to read less than available */
   }
   else *nrbytes = tmp_mbx_rec->msglength;
   eprintf(5,"cm_read: passed validation tests\n");
   if (*nrbytes)                              /* transfer data to mbx */
      bcpy(tmp_mbx_rec->data,usr_data,*nrbytes);
   ++(tmp_mbx_rec->readers.nr_accesses);
   tmp_mbx_rec->readers.who = tmp_fsm_rec;
   time(&when);                               /* timestamp */
   tmp_mbx_rec->readers.when = when;
   tmp_fsm_rec->read.when = when;
   ++(tmp_fsm_rec->read.nr_accesses);
   tmp_fsm_rec->read.mbxhandle = *mbxhandle;

   /*  Check if fsm has an update record for this mailbox in his update list.  */
   /*  if found, remove it.    */
   del_update_rec(tmp_fsm_rec,*mbxhandle);
   log_status(&cm_activity->cm_read.success,fsm,tmp_mbx_rec->mbxname);
   return(return_status);
}


/*  |||||||||||||||||||||    Pass the fsm the update list if one exists, or
 *  |     cm_ckmail      |    pass it NULL if none exists.
 *  |||||||||||||||||||||
 */
int cm_ckmail(fsm, list_ptr, nr_entries)
char *fsm;
struct update_list **list_ptr;
int *nr_entries;
{ fsm_rec *tmp_fsm_rec;

   eprintf(9,"cm_ckmail: at entry, fsm = %s\n", fsm);
   if (!cm_activity) cm_ini();                /* initialize cmm structures */
   if (!(tmp_fsm_rec = cm_fsm_find(fsm))) {   /* fsm active in cm ? */
      log_status(&cm_activity->cm_ckmail.success,NULL,NULL);
      return(E_CM_MBXNOTDECL);
   }
   if (tmp_fsm_rec->update_top)  {            /* if update list exists, send it */
      *list_ptr = tmp_fsm_rec->update_top;
      *nr_entries = tmp_fsm_rec->nr_updates;
      tmp_fsm_rec->update_top = NULL;         /* reset */
      tmp_fsm_rec->update_bot = NULL;
      tmp_fsm_rec->nr_updates = 0;
   }else {
      *list_ptr = NULL;
```

```
      *nr_entries = 0;
   }
   log_status(&cm_activity->cm_ckmail.success,fsm,NULL);
   return(I_CM_OK);
}


/*  ||||||||||||||||||||||    Pass the fsm the update list if one exists, or
 *  |      cm_disc       |    pass it NULL if none exists.
 *  ||||||||||||||||||||||
 */
int cm_disc(fsm)
char *fsm;
{ int return_status;
   byte *data_ptr;
   fsm_rec *tmp_fsm_rec;
   mbx_rec *tmp_mbx_rec;
   struct client_chain *tmp_client_rec;
   struct mbx_decl_chain *tptr;

   eprintf(9,"cm_disc: at entry, fsm = %s\n", fsm);
   if (!cm_activity) cm_ini();                 /* initialize cmm structures */
   if (!(tmp_fsm_rec = cm_fsm_find(fsm))) {    /* fsm active in cm ? */
      log_status(&cm_activity->cm_disc.success,NULL,NULL);
      return(E_CM_MBXNOTDECL);
   }
   cm_free_update_list(tmp_fsm_rec->update_top); /* free the update list */

   while (tmp_fsm_rec->read.mbx_list)  {       /* process the READ/XREAD list */
      tmp_mbx_rec = tmp_fsm_rec->read.mbx_list->mbx;
      eprintf(7,"cm_disc: processing read mbx %s\n",tmp_mbx_rec->mbxname);
      if (tmp_client_rec = find_mbx_client(tmp_mbx_rec->readerlist,tmp_fsm_rec)) {
         del_mbx_client(tmp_client_rec, &(tmp_mbx_rec->readerlist), &(tmp_mbx_rec->readers.nr_fsms));
         if ((!tmp_mbx_rec->readerlist)  &&  (!tmp_mbx_rec->writerlist)) {
            eprintf(7,"cm_disc: %s has no more clients... freed\n",tmp_mbx_rec->mbxname);
            --(cm_clients->mbx_active);
            cm_mbx_del(tmp_mbx_rec);            /* delete mbx if it has no more clients */
         }
      }else {
         printf("cm_disc: FATAL ERROR: attempt to undeclare mbx for which fsm isn't client.\n"
                "Contact the network team.\n");
         exit(0x40);
      }
      tptr = tmp_fsm_rec->read.mbx_list;
      tmp_fsm_rec->read.mbx_list = tmp_fsm_rec->read.mbx_list->next;
      free(tptr);
   }

   while (tmp_fsm_rec->write.mbx_list)  {      /* process the WRITE/XWRITE list */
      tmp_mbx_rec = tmp_fsm_rec->write.mbx_list->mbx;
      eprintf(7,"cm_disc: processing write mbx %s\n",tmp_mbx_rec->mbxname);
```

```
    if (tmp_client_rec = find_mbx_client(tmp_mbx_rec->writerlist,tmp_fsm_rec)) {
      del_mbx_client(tmp_client_rec, &(tmp_mbx_rec->writerlist), &(tmp_mbx_rec->readers.nr_fsms));
      if ((!tmp_mbx_rec->readerlist) && (!tmp_mbx_rec->writerlist)) {
        --(cm_clients->mbx_active);
        eprintf(7,"cm_disc: %s has no more clients... freed\n",tmp_mbx_rec->mbxname);
        cm_mbx_del(tmp_mbx_rec);            /* delete mbx if it has no more clients */
      }
    }else {
      printf("cm_disc: FATAL ERROR: attempt to undeclare mbx for which fsm isn't client.\n"
             "Contact the network team.\n");
      exit(0x40);
    }
    tptr = tmp_fsm_rec->write.mbx_list;
    tmp_fsm_rec->write.mbx_list = tmp_fsm_rec->write.mbx_list->next;
    free(tptr);
  }

  /* fsm should have no other mailboxes declared, so remove it from cm_fsm_list */
  if ((!tmp_fsm_rec->read.mbx_list) && (!tmp_fsm_rec->write.mbx_list)) {
    --(cm_clients->fsm_active);
    cm_fsm_del(tmp_fsm_rec);
  }else{
    printf("cm_disc: Didn't work... still have read/write clients for this fsm.\n"
           "Contact the network team.\n");
    exit(0x40);
  }
  log_status(&cm_activity->cm_disc.success,fsm,NULL);
  return(I_CM_OK);
}


/*  !!!!!!!!!!!!!!!!!!!!!    Construct a linked list of mbx names and return
 *  ! cm_get_mbx_list  !    a pointer to it to the caller.  If fsmname == NULL
 *  !!!!!!!!!!!!!!!!!!!!!    return a list of all mbx's in cm; else return the
 *                           list of mbx's of the specified fsm and for the
 *                           specified list_type access.
 */
int cm_get_mbx_list(fsmname, list_type, mbx_list_ptr, int_ptr)
char *fsmname, list_type;
struct mbx_list_type **mbx_list_ptr;
int *int_ptr;
{ struct mbx_list_type *tptr;
  fsm_rec *tmp_fsm_rec;
  mbx_rec *tmp_mbx_rec;
  struct mbx_decl_chain *tmp_client_rec;
  int return_status;

  if (!cm_activity) cm_ini();             /* initialize cmm structures */
  *mbx_list_ptr = NULL;                   /* initialize ptr */
  *int_ptr = 0;                           /* initialize nr entries on list */
  if (!(tmp_mbx_rec = cm_mbx_list))
```

```
      return(I_CM_OK);                        /* no mbx's in cm */
   tmp_client_rec = NULL;
   if (fsmname) {                             /* is a mbx name provided ? */
      if (return_status = cm_validate_fsm(fsmname))  {
         eprintf(9,"cm_get_mbx_list: invalid fsm name '%s'\n",fsmname);
         log_status(&cm_activity->cm_get_mbx_list.failure,NULL,NULL);
         return(return_status);               /* fsm error */
      }
      if (!(tmp_fsm_rec = cm_fsm_find(fsmname))){ /* ck if fsm exists */
         eprintf(9,"cm_get_mbx_list: fsm '%s' doesn't exist in cm\n",fsmname);
         log_status(&cm_activity->cm_get_mbx_list.failure,fsmname,NULL);
         return(E_CM_FSMNOTINCM);
      }
      switch (list_type) {
         case 'R':
         case 'r': tmp_client_rec = tmp_fsm_rec->read.mbx_list;
                   eprintf(9,"cm_get_mbx_list: for fsm '%s', read list\n",fsmname);
                   break;
         case 'W':
         case 'w': tmp_client_rec = tmp_fsm_rec->write.mbx_list;
                   eprintf(9,"cm_get_mbx_list: for fsm '%s', write list\n",fsmname);
                   break;
         default : eprintf(9,"cm_get_mbx_list: invalid list_type '%c' for fsm '%s'\n",list_type,fsmname);
                   log_status(&cm_activity->cm_get_mbx_list.failure,fsmname,NULL);
                   return(E_CM_MBXACCESS);
      }
      tmp_mbx_rec = NULL;                     /* so only tmp_client_rec is used */
   }else                                      /* no fsm name... get all mbx's */
      eprintf(9,"cm_get_mbx_list: no fsm name specified\n");

   /* build the list of fsm names */
   while ((tmp_mbx_rec) || (tmp_client_rec)) {
      tptr = (struct mbx_list_type *) malloc (sizeof(struct mbx_list_type));
      if (tptr) {
         bclr(tptr,sizeof(struct mbx_list_type));
         if (fsmname)
            strcpy(tptr->mbxname,tmp_client_rec->mbx->mbxname);
         else  strcpy(tptr->mbxname,tmp_mbx_rec->mbxname);
         tptr->next = *mbx_list_ptr;
         *mbx_list_ptr = tptr;
         if (fsmname)
            tmp_client_rec = tmp_client_rec->next;
         else tmp_mbx_rec = tmp_mbx_rec->next;
         ++(*int_ptr);
      }else{                                  /* else, out of memory */
         eprintf(9,"cm_get_mbx_list: ran out of memory\n");
         while (tptr = *mbx_list_ptr) {       /* free the list */
            *mbx_list_ptr = (*mbx_list_ptr)->next;
            free(tptr);
         }
         *int_ptr = 0;
```

```
        log_status(&cm_activity->cm_get_mbx_list.failure,fsmname,NULL);
        return(E_CM_INSUFFMEM);
      }
   }
   log_status(&cm_activity->cm_get_mbx_list.success,fsmname,NULL);
   return(I_CM_OK);                              /* nothing to report */
}


/*  ||||||||||||||||||||||   Construct a linked list of fsm names and return
 *  | cm_get_fsm_list    |   a pointer to the caller. If mbxname == NULL then
 *  ||||||||||||||||||||||   return all cm clients; else return list_type
 *                           clients of the specified mbx.
 */
int cm_get_fsm_list(mbxname, list_type, fsm_list_ptr, int_ptr)
char *mbxname, list_type;
struct fsm_list_type **fsm_list_ptr;
int *int_ptr;
{ struct fsm_list_type *tptr;
  fsm_rec *tmp_fsm_rec;
  mbx_rec *tmp_mbx_rec;
  struct client_chain *tmp_client_rec;
  int return_status;

  if (!cm_activity) cm_ini();                   /* initialize cmm structures */
  *fsm_list_ptr = NULL;                         /* initialize ptr */
  *int_ptr = 0;                                 /* initialize nr entries on list */
  if (!(tmp_fsm_rec = cm_fsm_list))
     return(I_CM_OK);                           /* no fsm's in cm */
  tmp_client_rec = NULL;
  if (mbxname) {                                /* is a mbx name provided ? */
     if (return_status = cm_validate_mbx(mbxname,1))  { /* fake the mbxsize */
        eprintf(9,"cm_get_fsm_list: invalid mbxname\n");
        log_status(&cm_activity->cm_get_fsm_list.failure,NULL,NULL);
        return(return_status);                  /* mbx error */
     }
     if (!(tmp_mbx_rec = cm_mbx_find(mbxname))){ /* ck if mbx exists */
        eprintf(9,"cm_get_fsm_list: mbxname doesn't exist in cm\n");
        log_status(&cm_activity->cm_get_fsm_list.failure,NULL,mbxname);
        return(E_CM_MBXNOTDECL);
     }
     switch (list_type) {
        case 'R':
        case 'r': tmp_client_rec = tmp_mbx_rec->readerlist;
                  eprintf(9,"cm_get_fsm_list: for mbx '%s', readerlist\n",mbxname);
                  break;
        case 'W':
        case 'w': tmp_client_rec = tmp_mbx_rec->writerlist;
                  eprintf(9,"cm_get_fsm_list: for mbx '%s', writerlist\n",mbxname);
                  break;
        default : eprintf(9,"cm_get_fsm_list: invalid list_type '%c' for mbx '%s'\n",list_type,mbxname);
```

```
                    log_status(&cm_activity->cm_get_fsm_list.failure,NULL,mbxname);
                    return(E_CM_MBXACCESS);
      }
      tmp_fsm_rec = NULL;                       /* so only tmp_client_rec is used */
   }else                                        /* no mbx name... get all fsm's */
      eprintf(9,"cm_get_fsm_list: no mbxname specified\n");

   /* build the list of fsm names */
   while ((tmp_fsm_rec) || (tmp_client_rec)) {
      tptr = (struct fsm_list_type *) malloc (sizeof(struct fsm_list_type));
      if (tptr) {
         bclr(tptr,sizeof(struct fsm_list_type));
         if (mbxname)
            strcpy(tptr->fsmname,tmp_client_rec->who->fsmname);
         else  strcpy(tptr->fsmname,tmp_fsm_rec->fsmname);
         tptr->next = *fsm_list_ptr;
         *fsm_list_ptr = tptr;
         if (mbxname)
            tmp_client_rec = tmp_client_rec->next;
         else tmp_fsm_rec = tmp_fsm_rec->next;
         ++(*int_ptr);
      }else{                                    /* else, out of memory */
         eprintf(9,"cm_get_fsm_list: ran out of memory\n");
         while (tptr = *fsm_list_ptr) {         /* free the list */
            *fsm_list_ptr = (*fsm_list_ptr)->next;
            free(tptr);
         }
         *int_ptr = 0;
         log_status(&cm_activity->cm_get_fsm_list.failure,NULL,mbxname);
         return(E_CM_INSUFFMEM);
      }
   }
   log_status(&cm_activity->cm_get_fsm_list.success,NULL,mbxname);
   return(I_CM_OK);                             /* nothing to report */
}


/*  |||||||||||||||||||||   Since the size of the statistics areas is static,
 *  |  cm_get_cm_stats  |   the user must provide pointers to space in the
 *  |||||||||||||||||||||   user data area into which the statistics will be
 *                          copied.  This avoids malloc overhead in case the
 *                          user wishes to call this routine multiple times.
 */
int cm_get_cm_stats(activity_ptr, client_ptr)
cm_activity_stats *activity_ptr;
cm_client_stats *client_ptr;
{
   if (!cm_activity) cm_ini();                  /* initialize cmm structures */
   if (CM_GET_STATS)  {                         /* collecting stats ? */
      bcpy(cm_activity, activity_ptr, sizeof(cm_activity_stats));
      bcpy(cm_clients, client_ptr, sizeof(cm_client_stats));
```

```
     eprintf(9,"cm_get_cm_stats: stats transferred to user area\n");
   }else{
     eprintf(9,"cm_get_cm_stats: cm is not logging stats\n");
     bclr(activity_ptr, sizeof(cm_activity_stats));
     bclr(client_ptr, sizeof(cm_client_stats));
   }
   log_status(&cm_activity->cm_get_cm_stats.success,NULL,NULL);
   return(I_CM_OK);                              /* nothing to report */
}


/*  ||||||||||||||||||||||   Since the size of the statistics area is static,
 *  |  cm_get_fsm_stats   |   the user must provide a pointer to space in the
 *  ||||||||||||||||||||||   user data area into which the statistics will be
 *                            copied.  This avoids malloc overhead in case the
 *                            user wishes to call this routine multiple times.
 */
int cm_get_fsm_stats(fsm, ptr)
char *fsm;
cm_fsm_stats_rec *ptr;
{ fsm_rec *tmp_fsm_rec;
  mbx_rec *tmp_mbx_rec;
  int return_status;

  if (!cm_activity) cm_ini();                  /* initialize cmm structures */
  if (return_status = cm_validate_fsm(fsm))  {
     eprintf(9,"cm_get_fsm_stats: invalid fsm name '%s'\n",fsm);
     log_status(&cm_activity->cm_get_fsm_stats.failure,NULL,NULL);
     return(return_status);                    /* fsm error */
  }
  if (!(tmp_fsm_rec = cm_fsm_find(fsm))){      /* ck if fsm exists */
     eprintf(9,"cm_get_fsm_stats: fsm '%s' doesn't exist in cm\n",fsm);
     log_status(&cm_activity->cm_get_fsm_stats.failure,fsm,NULL);
     return(E_CM_FSMNOTINCM);
  }
  ptr->nr_read_mbx  = tmp_fsm_rec->read.nr_mbxes;
  ptr->nr_reads     = tmp_fsm_rec->read.nr_accesses;
  ptr->read_time    = tmp_fsm_rec->read.when;
  if (tmp_fsm_rec->read.mbxhandle != 0)  {
     tmp_mbx_rec = cm_mbxhandle_find(tmp_fsm_rec->read.mbxhandle);
     (tmp_mbx_rec) ? strcpy(ptr->mbx_read,tmp_mbx_rec->mbxname)
                   : sprintf(ptr->mbx_read,"mbxhandle %d",tmp_fsm_rec->read.mbxhandle);
  }else
     sprintf(ptr->mbx_read,"<none>");
  ptr->nr_write_mbx = tmp_fsm_rec->write.nr_mbxes;
  ptr->nr_writes    = tmp_fsm_rec->write.nr_accesses;
  ptr->write_time   = tmp_fsm_rec->write.when;
  if (tmp_fsm_rec->write.mbxhandle != 0)  {
     tmp_mbx_rec = cm_mbxhandle_find(tmp_fsm_rec->write.mbxhandle);
     (tmp_mbx_rec) ? strcpy(ptr->mbx_write,tmp_mbx_rec->mbxname)
                   : sprintf(ptr->mbx_write,"mbxhandle %d",tmp_fsm_rec->write.mbxhandle);
```

```
  }else
     sprintf(ptr->mbx_write,"<none>");
  ptr->nr_updates  = tmp_fsm_rec->nr_updates;
  log_status(&cm_activity->cm_get_fsm_stats.success,fsm,NULL);
  return(I_CM_OK);
}


/*  |||||||||||||||||||||||   Since the size of the statistics area is static,
 *  |  cm_get_mbx_stats   |   the user must provide a pointer to space in the
 *  |||||||||||||||||||||||   user data area into which the statistics will be
 *                            copied.  This avoids malloc overhead in case the
 *                            user wishes to call this routine multiple times.
 */
int cm_get_mbx_stats(mbx, ptr)
char *mbx;
cm_mbx_stats_rec *ptr;
{ fsm_rec *tmp_fsm_rec;
  mbx_rec *tmp_mbx_rec;
  int return_status;

  if (!cm_activity) cm_ini();                 /* initialize cmm structures */
  if (return_status = cm_validate_mbx(mbx,1))  {  /* fake a mbxsize */
     eprintf(9,"cm_get_mbx_stats: invalid mbx name '%s'\n",mbx);
     log_status(&cm_activity->cm_get_mbx_stats.failure,NULL,NULL);
     return(return_status);                   /* fsm error */
  }
  if (!(tmp_mbx_rec = cm_mbx_find(mbx))){     /* ck if fsm exists */
     eprintf(9,"cm_get_mbx_stats: mbx '%s' doesn't exist in cm\n",mbx);
     log_status(&cm_activity->cm_get_mbx_stats.failure,mbx,NULL);
     return(E_CM_MBXNOTDECL);
  }

  ptr->handle         = tmp_mbx_rec->handle;
  ptr->declaredlength = tmp_mbx_rec->declaredlength;
  ptr->msglength      = tmp_mbx_rec->msglength;
  ptr->read_fsms      = tmp_mbx_rec->readers.nr_fsms;
  ptr->read_accesses  = tmp_mbx_rec->readers.nr_accesses;
  ptr->read_time      = tmp_mbx_rec->readers.when;
  (tmp_mbx_rec->readers.who) ? strcpy(ptr->reader,tmp_mbx_rec->readers.who->fsmname)
                             : sprintf(ptr->reader,"<none>");
  ptr->write_fsms     = tmp_mbx_rec->writers.nr_fsms;
  ptr->write_accesses = tmp_mbx_rec->writers.nr_accesses;
  ptr->write_time     = tmp_mbx_rec->writers.when;
  (tmp_mbx_rec->writers.who) ? strcpy(ptr->writer,tmp_mbx_rec->writers.who->fsmname)
                             : sprintf(ptr->writer,"<none>");
  (tmp_mbx_rec->readerlist->exclusive)
                             ? strcpy(ptr->xreader,tmp_mbx_rec->readerlist->who->fsmname)
                             : sprintf(ptr->xreader,"<none>");
  (tmp_mbx_rec->writerlist->exclusive)
                             ? strcpy(ptr->xwriter,tmp_mbx_rec->writerlist->who->fsmname)
```

```
                              : sprintf(ptr->xwriter,"<none>");
   log_status(&cm_activity->cm_get_mbx_stats.success,NULL,mbx);
   return(I_CM_OK);
}
```

```
/*  cm_utils.c -  contains the following utility routines (in the order
                  shown).  These are primarily used by the common memory
                  interface routines (in file cm_funcs.c), but are also
                  accessible to the user.



eprintf                   - displays variable-length arg list using fmt
                            if level <= DEBUG_LEVEL.  As written, is
                            specific to Turbo C.

cm_free_mbx_client_list - FREE the mbx's entire client list in preparation for
                            deleting the mbx.

cm_free_fsm_mbx_list    - FREE the fsm's entire mbx list in preparation for
                            deleting the fsm entry.

cm_free_update_list     - Free all memory allocated to an update list.

cm_fsm_find             - Check if the fsm is on the cm's client list.
                            If found, return ptr; else return NULL.

cm_mbx_find             - Check if the mbx is already in the cm.
                            If found, return ptr; else return NULL.

cm_mbx_del              - Remove a mailbox entry from the cm_mbx_list.

cm_fsm_del              - Remove an fsm entry from the cm_fsm_list.

cm_mbxhandle_find       - Find the mbx record associated w/ this mbxhandle.
                            Return NULL if not found.

find_mbx_client         - Check if the fsm is on this mailbox's client list.
                            If found, return ptr to client_chain record.
                            Else, return NULL.

add_mbx_client          - Add an fsm as a mbx client.

del_mbx_client          - Remove an fsm from the respective mbx client list.

find_fsm_mbx            - Find a mbx (by ptr) on the list kept by the fsm.

add_fsm_mbx             - Add a mbx to the list kept by the fsm.

del_fsm_mbx_entry       - Find a mbx (by ptr) on the list kept by the fsm
                            and remove it from the list.

cm_validate_fsm         - validate the fsm name.

cm_validate_mbx         - validate the mbx name and size.
```

```
cm_validate_access      - validate the mbx access code.

find_update_rec         - Look for an update record in the update list
                          of this fsm that references mbxhandle.  If found,
                          return pointer, else return NULL.

add_update_rec          - Add an update record to the list for this fsm.

del_update_rec          - If this mailbox is on the update list of this
                          fsm (waiting to be read) then delete it.

notify_clients          - Notify all clients on the access list of this
                          mbx that the mbx has been updated.  If the client
                          already has an update notice for this mbx, do not
                          make another entry.

clear_declare           - If new structures were allocated for this
                          mbx declaration, FREE them and return.

cm_ini                  - Initialize the variables and data structures
                          required by the common memory manager.  This
                          routine is called the 1st time a user makes a cm
                          call.  This call is triggered because cm_activity
                          is NULL.  The application can call cm_ini directly in
                          order to force the initialization at a known point.
                          This would be desirable, for example, if the
                          application wanted to change CM_DEBUG_LEVEL or
                          CM_GET_STATS.  Function can be called multiple times
                          without detremental results (ostensibly, this would
                          be for the purpose of displaying the library
                          version number).

log_status              - Routine to log cm useage statistics.

cm_get_statusname       - Return a pointer to string that gives the status name
                          associated with the status code.

*/

#include <stdio.h>
#include "cm_globals.h"


#include <stdarg.h>   /* as written, this routine is specific to Turbo C */
/* ||||||||||||||||||||||    displays variable-length arg list using fmt
 * |      eprintf       |    IF level <= DEBUG_LEVEL
 * ||||||||||||||||||||||
 */
void eprintf(int level, char *fmt, ...)
{ va_list argptr;
```

```
  if (level <= CM_DEBUG_LEVEL) {
     va_start(argptr, fmt);
     vprintf(fmt,argptr);
     va_end(argptr);
  }
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   FREE the mbx's entire client list
 *  ¦ cm_free_mbx_client_list ¦    in preparation for deleting the mbx
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
static void cm_free_mbx_client_list (list)
   struct client_chain *list;
{  struct client_chain *tptr;

  while (tptr = list)   {
     list = list->next;
     free(tptr);
  }
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   FREE the fsm's entire mbx list in
 *  ¦  cm_free_fsm_mbx_list   ¦   preparation for deleting the fsm entry
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
static void cm_free_fsm_mbx_list (list)
struct mbx_decl_chain *list;
{ struct mbx_decl_chain *tptr;

  while (tptr = list)   {
     list = list->next;
     free(tptr);
  }
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   free all memory allocated to an update list
 *  ¦ cm_free_update_list ¦
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
void cm_free_update_list(list)
struct update_list *list;
{
  struct update_list *tptr;

  while (tptr = list)  {
     list = tptr->next;
     free(tptr);
  }
```

```
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦    Check if the fsm is on the cm's client list.
 *  ¦   cm_fsm_find   ¦    If found, return ptr; else return NULL.
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
static fsm_rec  *cm_fsm_find (fsm)
   register char *fsm;
{
   register fsm_rec *tptr;

   if (tptr = cm_fsm_list)
      while (tptr)
         if (strcmp(fsm,tptr->fsmname) == 0)
            return (tptr);
         else tptr = tptr->next;
   return (NULL);
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦    Check if the mbx is already in the cm.
 *  ¦    cm_mbx_find     ¦    If found, return ptr; else return NULL.
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
static mbx_rec  *cm_mbx_find (mbxname)
   char  *mbxname;
{
   register mbx_rec *tptr;

   if (tptr = cm_mbx_list)
      while (tptr)
         if (strcmp(mbxname,tptr->mbxname) == 0)
            return (tptr);
         else tptr = tptr->next;
   return (NULL);
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   Remove a mailbox entry from the cm_mbx_list
 *  ¦    cm_mbx_del    ¦
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
static int cm_mbx_del(entry)
mbx_rec  *entry;
{
  mbx_rec *tptr;

  free(entry->data);                     /* free the data mailbox */
  if (cm_mbx_list == entry)  {           /* if it's the 1st entry, adj ptrs */
     cm_mbx_list = entry->next;
```

```
      free(entry);
      eprintf(6,"cm_mbx_del: mbx entry deleted from top of list\n");
      return;
   }
   tptr = cm_mbx_list;                     /* look for this entry's predecessor */
   while (tptr)
      if (tptr->next == entry) {           /* found it.  adjust the pointer */
         tptr->next = entry->next;
         free(entry);
         eprintf(6,"cm_mbx_del: mbx entry deleted\n");
         return;
      } else tptr = tptr->next;
   printf("%ccm_mbx_del: FATAL ERROR - entry not found!  Call network team.");
   exit(0x40);
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   Remove an fsm entry from the cm_fsm_list
 *  ¦      cm_fsm_del    ¦
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
static int cm_fsm_del(entry)
fsm_rec  *entry;
{
   fsm_rec *tptr;

   cm_free_update_list(entry->update_top);
   if (cm_fsm_list == entry)  {            /* if it's the 1st entry, adj ptrs */
      cm_fsm_list = entry->next;
      free(entry);
      eprintf(6,"cm_fsm_del: fsm entry deleted from top of list\n");
      return;
   }
   tptr = cm_fsm_list;                     /* look for this entry's predecessor */
   while (tptr)
      if (tptr->next == entry) {           /* found it.  adjust the pointer */
         tptr->next = entry->next;
         free(entry);
         eprintf(6,"cm_fsm_del: fsm entry deleted\n");
         return;
      } else tptr = tptr->next;
   printf("%ccm_fsm_del: FATAL ERROR - entry not found!  Call network team.");
   exit(0x40);
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   Find the mbx record associated w/ this mbxhandle.
 *  ¦ cm_mbxhandle_find  ¦   Return NULL if not found.
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
static mbx_rec  *cm_mbxhandle_find (mbxhandle)
```

```
int mbxhandle;
{
   register mbx_rec *tptr;

   if (tptr = cm_mbx_list)
      while (tptr)
         if (tptr->handle == mbxhandle)
            return (tptr);
         else tptr = tptr->next;
   return (NULL);
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦    Check if the fsm is on this mailbox's client list.
 *  ¦   find_mbx_client   ¦    If found, return ptr to client_chain record.
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦    Else, return NULL.
 */
static struct client_chain *find_mbx_client (list,fsm)
   struct client_chain *list;
   register char *fsm;
{
   struct client_chain *tptr;

   if (tptr = list)
      while (tptr)
         if (tptr->who == fsm)
            return (tptr);
         else tptr = tptr->next;
   return(NULL);
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦    Add an fsm as a mbx client.
 *  ¦   add_mbx_client    ¦
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
static int add_mbx_client (fsm, clientlist, exclusive)
   struct client_chain **clientlist;
   boolean exclusive;
   char *fsm;
{
   struct client_chain *tmp_client_rec;

   eprintf(7,"add_mbx_client: attempt to add %s\n",fsm);
   tmp_client_rec = (struct client_chain *) malloc (sizeof(struct client_chain));
   if (!tmp_client_rec)  return(E_CM_INSUFFMEM);  /* verify that malloc worked */
   tmp_client_rec->who = fsm;
   tmp_client_rec->next = *clientlist;
   tmp_client_rec->exclusive = exclusive;
   *clientlist = tmp_client_rec;
   return(I_CM_OK);
```

```
}


/*  ||||||||||||||||||||||    Remove an fsm from the respective mbx client list.
 *  |   del_mbx_client   |
 *  ||||||||||||||||||||||
 */
static void del_mbx_client (client, clientlist, nr_clients)
struct client_chain *client, **clientlist;
int *nr_clients;
{ struct client_chain *tptr;

   eprintf(7,"del_mbx_client: attempt to add %s\n",client->who->fsmname);
  --(*nr_clients);                        /* decrement client count */
  if (client == *clientlist) {            /* if it's the 1st entry, adj ptrs */
     *clientlist = client->next;
     free(client);
     eprintf(9,"del_mbx_client: fsm client entry deleted from top of list\n");
     return;
  }
  tptr = *clientlist;                     /* look for this entry's predecessor */
  while (tptr)
     if (tptr->next == client) {          /* found it.  adjust the pointer */
        tptr->next = client->next;
        free(client);
        eprintf(9,"del_mbx_client: fsm client entry deleted\n");
        return;
     } else tptr = tptr->next;
  printf("%cdel_mbx_client: FATAL ERROR - client entry not found!  Call network team.");
  exit(0x40);
}


/*  ||||||||||||||||||||||    Find a mbx (by ptr) on the list kept by the fsm
 *  |    find_fsm_mbx    |
 *  ||||||||||||||||||||||
 */
static struct mbx_deci_chain *find_fsm_mbx (list, mbx)
   struct mbx_deci_chain *list;
   mbx_rec *mbx;
{
   while (list)
      if (list->mbx == mbx)
         return(list);
      else list = list->next;
   return(NULL);
}


/*  ||||||||||||||||||||||    Add a mbx to the list kept by the fsm
 *  |     add_fsm_mbx    |
```

```
 * ||||||||||||||||||||||
 */
static int add_fsm_mbx (list, mbx)
   struct mbx_decl_chain **list;
   mbx_rec *mbx;
{  struct mbx_decl_chain *tptr;

   if (find_fsm_mbx(*list,mbx))
      return(I_CM_OK);                      /* already exists, don't need to add it */
   eprintf(7,"add_fsm_mbx: attempt to add %s\n",mbx->mbxname);
   tptr = (struct mbx_decl_chain *) malloc (sizeof(struct mbx_decl_chain));
   if (!tptr) return(E_CM_INSUFFMEM);       /* verify that malloc worked */
   tptr->mbx = mbx;
   tptr->next = *list;
   *list = tptr;
   return(I_CM_OK);
}


/* |||||||||||||||||||||    Find a mbx (by ptr) on the list kept by the fsm
 *  | del_fsm_mbx_entry |    and remove it from the list
 *  |||||||||||||||||||||
 */
static void  del_fsm_mbx_entry (list, mbx)
struct mbx_decl_chain **list;
mbx_rec *mbx;
{  struct mbx_decl_chain *tptr, *previous;

   eprintf(7,"del_fsm_mbx_entry: looking for mbx %s\n",mbx->mbxname);
   tptr = *list;
   previous = NULL;
   while (tptr) {
      eprintf(9,"del_fsm_mbx_entry: comparing %s\n",tptr->mbx->mbxname);
      if (tptr->mbx == mbx)
         if (previous == NULL)  {
            *list = tptr->next;
            free (tptr);
            return;
         }else {
            previous->next = tptr->next;
            free (tptr);
            return;
         }
      else {
         previous = tptr;
         tptr = tptr->next;
      }
   }
   eprintf(7,"del_fsm_mbx_entry: couldn't find mbx on list\n");
}
```

```
/* ||||||||||||||||||||||  validate the fsm name
 * |   cm_validate_fsm  |
 * ||||||||||||||||||||||
 */
int cm_validate_fsm(fsm)
char * fsm;
{ int i;

  i = strlen(fsm);
  eprintf(9,"cm_validate_fsm: fsm = \"%s\", length = %d characters\n",fsm,i);
  if ((i > MAXFSMNAMELENGTH) || (i < 1))
     return (E_CM_FSMNAME);
  return(I_CM_OK);
}


/* ||||||||||||||||||||||  validate the mbx name and size
 * |   cm_validate_mbx  |
 * ||||||||||||||||||||||
 */
int cm_validate_mbx(mbxname,mbxsize)
char * mbxname;
int mbxsize;
{
  int i;

  i = strlen(mbxname);                          /* validate mbx name */
  eprintf(9,"cm_validate_mbx: mbxname = \"%s\", length = %d characters\n",mbxname,i);
  if ((i > MAXMBXNAMELENGTH) || (i < 1))
     return (E_CM_MBXNAME);

  /* validate mbx size  -  must be greater than 0 bytes long */
  eprintf(9,"cm_validate_mbx: mbxsize = %d\n",mbxsize);
  if (mbxsize < 1)
     return (E_CM_MBXSIZE);
  return(I_CM_OK);
}

/* ||||||||||||||||||||||  validate the mbx access code
 * | cm_validate_access |
 * ||||||||||||||||||||||
 */
int cm_validate_access(mbxaccess)
int mbxaccess;
{
  int i;

  eprintf(9,"cm_validate_access: mbxaccess = (hex) %x\n",mbxaccess);
  i = mbxaccess & (CM_READ_ACCESS | CM_WRITE_ACCESS | CM_XREAD_ACCESS | CM_XWRITE_ACCESS);
  if ((mbxaccess == 0) || (i != mbxaccess))
```

```
      return (E_CM_MBXACCESS);                  /* unrecognized code */
    if ((mbxaccess & (CM_READ_ACCESS ¦ CM_XREAD_ACCESS)) == (CM_READ_ACCESS ¦ CM_XREAD_ACCESS))
      return (E_CM_MBXACCBOTH);                 /* can't request both concurrently */
    if ((mbxaccess & (CM_WRITE_ACCESS ¦ CM_XWRITE_ACCESS)) == (CM_WRITE_ACCESS ¦ CM_XWRITE_ACCESS))
      return (E_CM_MBXACCBOTH);                 /* can't request both concurrently */
    return(I_CM_OK);
  }


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   Look for an update record in the update list
 *  ¦  find_update_rec   ¦   of this fsm that references mbxhandle.  If found,
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   return pointer, else return NULL.
 */
static struct update_list *find_update_rec(fsm,mbxhandle)
int mbxhandle;
fsm_rec  *fsm;
{
  struct update_list *tptr;

  if (tptr = fsm->update_top)                 /* a list exists */
    while (tptr)
      if (tptr->mbxhandle == mbxhandle) {
        eprintf(9,"find_update_rec: FOUND for %s, mbxhandle %d\n",
                fsm->fsmname, mbxhandle);
        return(tptr);
      }else tptr = tptr->next;
  eprintf(9,"find_update_rec: NOT FOUND for %s, mbxhandle %d\n",
          fsm->fsmname, mbxhandle);
  return(NULL);
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   Add an update record to the list for this fsm
 *  ¦   add_update_rec   ¦
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
static void add_update_rec(fsm,mbxhandle)
int  mbxhandle;
fsm_rec  *fsm;
{
   struct update_list *tptr;

   tptr = (struct update_list *) malloc (sizeof(struct update_list));
   if (!tptr)  {                               /* verify that malloc worked */
      printf("%cadd_update_rec:  FATAL ERROR malloc failed.  Out of Memory ?",7);
      exit(0x40);
   }
   tptr->mbxhandle = mbxhandle;
   tptr->next = NULL;
   if (fsm->update_bot)
      fsm->update_bot->next = tptr;            /* adjust NEXT ptr for old BOTTOM */
```

```
   fsm->update_bot = tptr;                   /* point to new bottom */
   if (!fsm->update_top)                     /* if this is 1st structure in chain ... */
      fsm->update_top = tptr;                /* then set TOP ptr */
   ++(fsm->nr_updates);
   eprintf(8,"add_update_rec: update record added to %s, mbxhandle %d\n",
           fsm->fsmname, tptr->mbxhandle);
}


/* ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   if this mailbox is on the update list of this
 * ¦   del_update_rec   ¦   fsm (waiting to be read) then delete it.
 * ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
static void del_update_rec(fsm,mbxhandle)
int  mbxhandle;
fsm_rec  *fsm;
{ struct update_list *tptr, *t2ptr;

  if (tptr = fsm->update_top)  {             /* make sure we have a list */
     if (tptr->mbxhandle == mbxhandle)  {    /* is it at the top of the list */
        eprintf(8,"del_update_rec: deleted from top of update rec list\n");
        if (fsm->update_top == fsm->update_bot)  {
           fsm->update_top = NULL;           /* only 1 entry on the list */
           fsm->update_bot = NULL;           /* so adjust bottom ptr too */
        }else
           fsm->update_top = fsm->update_top->next;
        free(tptr);
        --(fsm->nr_updates);                 /* decrement update count */
        return;
     }

     while(tptr->next)
        if (tptr->next->mbxhandle == mbxhandle) {
           --(fsm->nr_updates);              /* decrement update count */
           t2ptr = tptr->next;
           tptr->next = tptr->next->next;    /* adjust the ptr */
           free(t2ptr);                      /* free the update record */
           if (tptr->next == fsm->update_bot)
              fsm->update_bot = tptr;
           eprintf(8,"del_update_rec: deleted\n");
           return;
        }else tptr = tptr->next;
  }
}


/* ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   Notify all clients on the access list of this
 * ¦   notify_clients   ¦   mbx that the mbx has been updated.  If the client
 * ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   already has an update notice for this mbx, do not
 *                          make another entry.
 */
```

```
static void notify_clients(clientlist, mbxhandle)
struct client_chain  *clientlist;
int  mbxhandle;
{
  eprintf(9,"notify_clients: for mbxhandle %d\n",mbxhandle);
  if (clientlist)                                        /* client list exists ? */
     while (clientlist) {
        if (!find_update_rec(clientlist->who, mbxhandle))  /* update exist ? */
           add_update_rec(clientlist->who,mbxhandle);      /* NO - add it */
        clientlist = clientlist->next;                     /* check next client */
     }
}


/*  ||||||||||||||||||||||   If new structures were allocated for this
 *  |    clear_declare    |   mbx declaration, FREE them and return.
 *  ||||||||||||||||||||||
 */
static int clear_declare (new_fsm, new_mbx, tmp_fsm_rec, tmp_mbx_rec, err_code)
boolean  new_fsm, new_mbx;
fsm_rec  *tmp_fsm_rec;
mbx_rec  *tmp_mbx_rec;
int      err_code;
{
   if (new_fsm)  {                               /* FREE the fsm rec */
      eprintf(6,"clear_declare: free the fsm record \n");
      if (tmp_fsm_rec->read.mbx_list)  {       /* FREE mbx lists first */
         eprintf(6,"clear_declare: free fsm reader list\n");
         cm_free_fsm_mbx_list(tmp_fsm_rec->read.mbx_list);
      }
      if (tmp_fsm_rec->write.mbx_list) {
         eprintf(6,"clear_declare: free fsm writer list\n");
         cm_free_fsm_mbx_list(tmp_fsm_rec->write.mbx_list);
      }
      free(tmp_fsm_rec);
   }
   if (new_mbx) {
      eprintf(6,"clear_declare: free the mbx record \n");
      if (tmp_mbx_rec->data)  {                 /* FREE the data mbx, too */
         eprintf(6,"clear_declare: free data mbx\n");
         free(tmp_mbx_rec->data);
      }
      if (tmp_mbx_rec->readerlist) {           /* FREE the client lists first */
         eprintf(6,"clear_declare: free mbx readerlist\n");
         cm_free_mbx_client_list(tmp_mbx_rec->readerlist);
      }
      if (tmp_mbx_rec->writerlist) {
         eprintf(6,"clear_declare: free mbx writerlist\n");
         cm_free_mbx_client_list(tmp_mbx_rec->writerlist);
      }
      free(tmp_mbx_rec);
```

```c
        --cm_mbxhandle;
    }
    return (err_code);
}


/*  ||||||||||||||   Initialize the variables and data structures required
 *  |  cm_ini    |   by the common memory manager.  This routine is called
 *  ||||||||||||||   the 1st time a user makes a cm call.  This is triggered
 *                   because cm_activity == NULL;
 *
 *  The following values, when assigned to CM_DEBUG_LEVEL, result in the
 *  display of debugging data corresponding to that level via routine eprintf.
 *  The debugging levels "build" upon each other.  That is, selecting a
 *  debug level also displays those levels below it (ie, those that have
 *  a lower debug level number).
 *
 *          0  -  no debugging data is displayed
 *          1  -  display the common memory library version (compiled into lib)
 *          2  -
 *          3  -
 *          4  -
 *          5  -  when a mbx is written or read
 *          6  -  when fsm/mbx are added/deleted by cmm,
 *                or when cm_declare failed.
 *          7  -  when an fsm/mbx client is added/deleted
 *          8  -  when an fsm has update records added/deleted
 *          9  -  everything (includes 0-8 and more)
 *
 */
void cm_ini()
/* The initial global value of CM_DEBUG_LEVEL is set to -1 in file cm_globa.h.
   If the application program changes the value to be >= 0, then cm_ini will not
   change it again.  If it is still <0, cm_ini will turn off debugging satements */
{  if (CM_DEBUG_LEVEL < 0) CM_DEBUG_LEVEL = 0;
   eprintf(1,"cm_ini: using common memory library version %s\n",CM_VERSION);
   if (cm_activity)                            /* ck if we've been here before */
      eprintf(9,"cm_ini: function called again AFTER init already performed\n");
   else {
      eprintf(9,"cm_ini: init in progress\n");
      cm_activity = (cm_activity_stats *) malloc (sizeof(cm_activity_stats));
      cm_clients = (cm_client_stats *) malloc (sizeof(cm_client_stats));
      if ((!cm_activity) || (!cm_clients)) { /* verify that malloc worked */
         printf("%ccm_ini:  FATAL ERROR malloc failed.  Out of Memory ?",7);
         exit(0x40);
      }
      bclr(cm_activity,sizeof(cm_activity_stats));
      bclr(cm_clients,sizeof(cm_client_stats));
   }
}
```

```
/*  ||||||||||||||   Routine to log cm useage statistics.
 *  | log_status |
 *  ||||||||||||||
 */
static void log_status(func,fsm, mbx)
base_stats *func;
char *fsm, *mbx;
{
   if (CM_GET_STATS) {
      eprintf(9,"log_status: in progress\n");
      strcpy(func->fsm,fsm);                  /* save the fsm name */
      strcpy(func->mbx,mbx);                  /* save the mbx name */
      time(&func->when);                      /* note the time and date */
      ++func->nr_times;                       /* increment nr times this function called */
   }
}


/*  ||||||||||||||||||||||   return ptr to string that gives the status name
 *  | cm_get_statusname |   associated with the status code.
 *  ||||||||||||||||||||||
 */
char *cm_get_statusname (code)
int code;
{ char * status;

  switch (code) {
    case I_CM_OK           : status = "I_CM_OK";            break;
    case I_CM_MBXACTV      : status = "I_CM_MBXACTV";       break;
    case I_CM_DUPMBX       : status = "I_CM_DUPMBX";        break;
    case I_CM_MOREDATA     : status = "I_CM_MOREDATA";      break;
 /*  case E_CM_FATALERR    :    */
    case E_CM_INSUFFMEM    : status = "E_CM_FATALERR or E_CM_INSUFFMEM";
                             break;
    case E_CM_MBXERR       : status = "E_CM_MBXERR";        break;
    case E_CM_MBXNOREAD    : status = "E_CM_MBXNOREAD";     break;
    case E_CM_MBXNOXREAD   : status = "E_CM_MBXNOXREAD";    break;
    case E_CM_MBXNOWRITE   : status = "E_CM_MBXNOWRITE";    break;
    case E_CM_MBXNOXWRITE  : status = "E_CM_MBXNOXWRITE";   break;
    case E_CM_MBXSIZE      : status = "E_CM_MBXSIZE";       break;
    case E_CM_MBXACCESS    : status = "E_CM_MBXACCESS";     break;
    case E_CM_MBXACCBOTH   : status = "E_CM_MBXACCBOTH";    break;
    case E_CM_MBXNAME      : status = "E_CM_MBXNAME";       break;
    case E_CM_MBXNOTDECL   : status = "E_CM_MBXNOTDECL";    break;
    case E_CM_FSMERR       : status = "E_CM_FSMERR";        break;
    case E_CM_FSMNAME      : status = "E_CM_FSMNAME";       break;
    case E_CM_FSMNOTINCM   : status = "E_CM_FSMNOTINCM";    break;
    default                : status = "unknown code";       break;
  }
  return (status);
```

```
}
```

```
/*  sfuncs.c  -  miscellaneous functions
 *
 *    asciidump(b,n)  dump n bytes starting at b, in ascii.
 *
 *    bclr(b,n)       clears (sets = 0) n bytes starting at address b.
 *
 *    bcpy(s,t,n)     copies n bytes from s to t.
 *
 *    binstr(n,str)   convert byte to binary representation in string str.
 *
 *    cvtup(c)        converts a lowercase ascii string to upper.
 *
 *    hexdump(b,n)    dump n bytes starting at b, in hex.
 *
 *    is_ascii(c)     returns TRUE if c is a printable ascii character.
 *
 *    pause(fmt,...)  routine works like "printf", and accepts a variable-
 *                    length arg list after fmt.  After it displays the
 *                    user-specified data, it displays "Press any character
 *                    to continue...", waits for a kbd entry, then
 *                    outputs <CRLF> and returns.
 */


/*  ||||||||||||||||||||||     display n bytes in ascii, starting w/ address b.
 *  |     asciidump      |     'i' columns per line.
 *  ||||||||||||||||||||||
 */
void asciidump(b,n)
   register char *b;
   register int n;
{  register int i = 20;

   while (n--) {
     if ((*b > 0x1f)  &&  (*b < 0x7f))
        printf ("%3c",*b++);
     else printf (" %02x", 0xff & *b++);
     if (--i == 0) {
        printf ("\n");
        i = 20;
     }
   }
   if (i < 20) printf("\n");
}


/*  ||||||||||||||||||||||     set n bytes = 0, starting w/ address b.
 *  |        bclr        |
 *  ||||||||||||||||||||||
 */
void bclr(b, n)
```

```
   register char *b;
   register int n;
{
   while (n--)  *b++ = 0;
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦    copy n bytes from s to t
 *  ¦        bcpy        ¦
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
void bcpy(s, t, n)
   register char *s, *t;
   register int n;
{
   while (n--)  *t++ = *s++;
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦    convert byte n to its binary representation
 *  ¦       binstr       ¦    in string str.  Return ptr to str in case
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦    caller wants to display it as part of a printf.
 *                            The user-allocated space for str must be at least
 *                            9 characters long, 8 for the binary representation
 *                            and 1 for the trailing NULL;
 */
char * binstr(n,str)
unsigned char n;
char *str;                                /* must be at least 9 bytes long */
{  int i;

   for (i = 0; i < 8; i++)  {
      str[i] = (n & 0x80) ? '1' : '0';
      n = n << 1;
   }
   str[8] = 0;                            /* null - terminate the string */
   return (str);
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦    convert a lowercase ASCII string to uppercase.
 *  ¦       cvtup        ¦
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
void cvtup (s)
   register char *s;
{
   while (*s) {
      if (*s >= 'a' && *s <= 'z') *s -= 'a' - 'A';
      s++;
   }
```

```
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦    display n bytes in hex, starting w/ address b.
 *  ¦      hexdump       ¦    'i' columns per line.
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
void hexdump(b,n)
   char *b;
   int n;
{  int group, i, max;
   int max_per_line = 15;
   int group_size = 5;

   while (n) {
     max = max_per_line;
     if (max > n) max = n;
     group = group_size;
     for (i=1; i <= max; i++)  {
         printf (" %02x",*b++ & 0xff);
         if (!--group) {
            printf ("  ");
            group = group_size;
         }
     }
     if (i <= max_per_line)
        for (i = i; i <= max_per_line; i++)  {
            printf("   ");                       /* space over to ascii field */
            if (!--group) {
               printf ("  ");
               group = group_size;
            }
        }
     printf("    ");                             /* gap between hex and ascii areas */
     b -= max;                                   /* point to start of section */
     for (i=1; i <= max; i++) {
         printf("%c", is_ascii(*b) ? *b : 5);
         b++;
         if (!--group) {
            printf ("%c",176);
            group = group_size;
         }
     }
     printf("\n");
     n -= max;
   }
}


/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦    returns TRUE if c is a printable ascii character.
 *  ¦     is_ascii       ¦
```

```
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
 */
int is_ascii(c)
  char c;
{
  if ((c > 0x1f)  &&  (c < 0x7f))
     return (1);
  return (0);
}

#include <stdio.h>           /* as written, this routine is specific to Turbo C */
#include <stdarg.h>
/*  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   displays variable-length arg list using fmt. Then,
 *  ¦     pause          ¦   "Press any character to continue..." and waits for
 *  ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦   kbd input.  Then echoes <CRLF> and returns.  User
 *                           msg string can have format ctrl chars in it.   */
void pause(char *fmt, ...)
{ va_list argptr;

  va_start(argptr, fmt);
  vprintf(fmt,argptr);
  va_end(argptr);
  printf("Press any character to continue... ");
  getch();
  printf("\n");
}
```

GLOSSARY

AMRF - acronym, refers to the Automated Manufacturing Research Facility of the National Bureau of Standards.

cm - abbreviation for "common memory". See also common memory.

cmm - abbreviation for "common memory manager". See also common memory manager.

common memory - a term used to generically identify both local and global common memory. See also local common memory and global common memory.

common memory manager - collection of functions that establish the mailboxes on the local host and manage associated data structures to assure and provide proper access to them. The common memory manager also gathers and provides utilization statistics.

DOS - abbreviation for "Disk Operating System", a single-user operating system developed by Microsoft Corporation for use with the IBM PC class of machine. It is marketed by various companies under the names PC DOS and MS DOS.

finite state machine - a term assigned to user application programs that have a finite number of clearly defined processing states. Examples of such states are: data acquisition, data reduction, and data reporting. In the context of this documentation, "fsm" is intended to mean "user application program".

fsm - abbreviation for "finite state machine". See also finite state machine.

global common memory - two or more local common memories combine to form a global common memory. This is accomplished with the introduction of a network interface process (NIP) at each computer system that has a local common memory. The NIP becomes another client of its local common memory with all implied READ/WRITE privileges. NIPs exchange common memory mailgrams with each other using network services, propagating these mailgrams globally and creating the global common memory.

local common memory - a contiguous area of physical memory accessible to two or more distinct processes within a single computer system. This physical memory is divided into a collection of READ and WRITE mailbox areas. The data in these areas, called mailgrams, is available to all other applications on the computer system.

mailbox - a contiguous area of high-speed memory assigned and managed by the common memory manager. Messages (called mailgrams) can be placed into a mailbox by one or more writer applications and copied from the mailbox by one or more reader applications.

mailgram - term used to identify the contents of a mailbox. That is, a collection of contiguous bytes stored in a mailbox. The data representation (binary, ASCII, etc.) for mailgrams is determined by the application process.

mbx - abbreviation for "mailbox". See also mailbox.

NBS - acronym, refers to the National Bureau of Standards, located in Gaithersburg, Maryland.

PC - abbreviation for "personal computer". This term is used to identify all classes of personal computers that are compatible with the IBM PC and use the DOS operating system.

process - term used to identify a user application program. It is used interchangably with "fsm".

LIST OF REFERENCES

[1] Kernighan, Brian W., Ritchie, Dennis M., THE C PROGRAMMING LANGUAGE, Prentice-Hall, Inc., 1978.

[2] TURBO C USER'S GUIDE, Borland International, 1987.

[3] TURBO C REFERENCE GUIDE, Borland International, 1987.

[4] Holt, R. C., Graham, G. S., Lazowska, E. D., and Scott, M. A., Structured Concurrent Programming with Operating Systems Applications, Addison-Wesley Publishing Company, Reading, MA, 1978, p25.

[5] Barbera, A. J., Fitzgerald, M. L., Albus, J. S., "Concepts for a Real-Time Sensory Interactive Control System Architecture", Procedings of the Fourteenth Southeastern Symposium on System Theory, April 1982, pp 121-126.

[6] Furlani, C., Kent, E., Bloom, H., McLean, C., "The Automated Manufacturing Research Facility of the National Bureau of Standards", Proceedings of the Summer Simulation Conference, Vancouver, B.C., Canada, July 1983.

[7] Rybczynski, S., et.al., "AMRF Network Communications", NBS-IR-88-3816, June 1988, 206 pp.

[8] Barbera, A. J., Fitzgerald, M. L., Albus, J. S., and Haynes, L. J., "RCS: The NBS Real-Time Robot Control System", Proceedings of the Robots VIII Conference, Detroit, Michigan, June 1984.

[9] Libes, D., "User-Level Shared Variables", Tenth USENIX Conference Proceedings, Summer 1985.

[10] Libes, D., "Experiences with a Communications Paradigm: Common Memory", in preparation.

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. NISTIR 88-3838	2. Performing Organ. Report No.	3. Publication Date AUGUST 1988

4. TITLE AND SUBTITLE
Common Memory for the Personal Computer

5. AUTHOR(S)
Rybczynski, S.

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)	7. Contract/Grant No.
NATIONAL BUREAU OF STANDARDS **U.S. DEPARTMENT OF COMMERCE** **GAITHERSBURG, MD 20899**	8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

10. SUPPLEMENTARY NOTES

[X] Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

The Automated Manufacturing Research Facility (AMRF) at the National Bureau of Standards is using an architecture called "common memory" (also known as shared memory) for interprocess communication. This document describes the shared memory concept and defines the shared memory architecture as implemented on the IBM Personal Computer (or compatible) using the DOS operating system. A complete shared memory software library has been written using the C programming language to maximize portability to other systems. A sample program demonstrating the use of the common memory environment is included in the software distribution.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

AMRF; common memory; communications; shared memory

13. AVAILABILITY	14. NO. OF PRINTED PAGES
[X] Unlimited	161
[ ] For Official Distribution. Do Not Release to NTIS	
[ ] Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.	15. Price
[X] Order From National Technical Information Service (NTIS), Springfield, VA. 22161	$18.95

USCOMM-DC 6043-P80