NIST Network Common Memory User Manual

Don Libes

National Institute of Standards and Technology
Gaithersburg, MD 20899

NISTIR 90-4233
PB90-183260/AS
January, 1990

ABSTRACT

This manual describes how to use the NIST Network Common Memory System (CMS), version 7. The CMS provides a common memory that may be shared among processes distributed across a local area network. Unlike other shared memory systems, CMS accesses variables by name rather than by address. These and other features make the system more comparable to a primitive distributed database than a shared memory system.

CMS currently runs on Berkeley UNIX systems but can be supported on any POSIX-like system which provides a stream protocol at the transport layer. Interfaces exist for usage from C and Franz Lisp.

Keywords: common memory, shared memory, distributed common memory, distributed shared memory, POSIX, UNIX.

Introduction

This manual describes how to use the NIST Network Common Memory System (CMS), version 7. The system provides a primitive distributed data system, but is called "common memory" because of its efficiency and secondly due to historical reasons. (This system is loosely based on the Hierarchical Control System Emulator (HCSE) built at Bolt Beranek and Newman Inc. (BBN) [Johnson 82], which provided a shared memory as one of its facilities.)

The reader is referred to [Libes 85] for a higher-level paper on the CMS. The only thing lower-level than this manual is the source code itself. For more information, see "How to get the software - FTP" on page 17.

The CMS provides a variety of interesting features for such a simple and easy to use service:

- rudimentary access permissions
- control over synchronization problems
- users can be distributed across a local area network.
- server is robust in face of experimental clients
- value updates may be queued if desired for slow consumers
- alternative synchronization styles
- hierarchical control features
- value serial numbers

This current implementation runs on Sun workstations (SunOS 1.0 through 4.0) on 68000 and 80386-based processors. It has also been ported to the Silicon Graphics Iris. It requires very little in the way of system support beyond a stream transport-level protocol [Libes 89] and some similarity to POSIX.

Overview of how it works

Each client keeps a local copy of a portion of common memory. This common memory is partitioned into common memory variables (called “cm_variables”), uniquely identified in the CMS by name. Each cm_variable holds a value (called a “cm_value”) which the client may read or write. Such accesses are performed locally.

In order to update clients' common memories, they must be brought into sync with a master common memory (see figure 1) maintained by the common memory manager (CMM). This is done synchronously at the convenience of the client (see "Synchronization" on page 8).

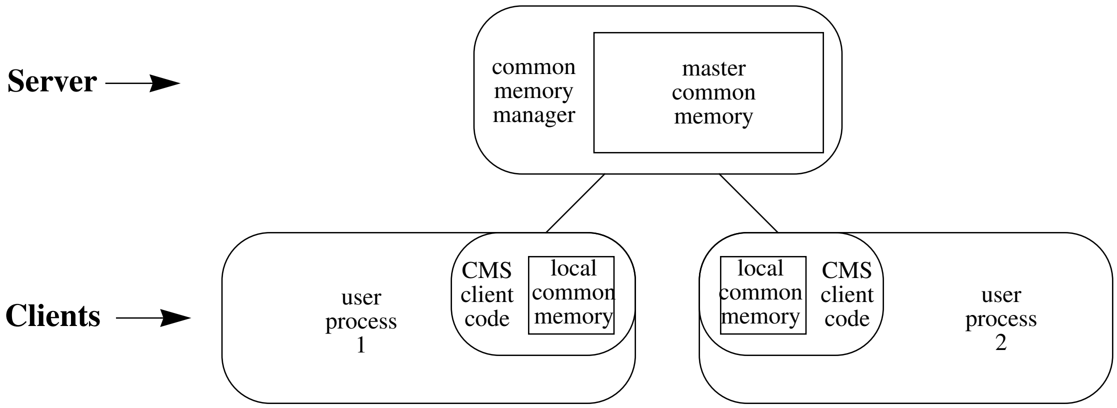

Figure 1: Common memory is implemented in the client-server paradigm.

Synchronization is implemented by sending messages (usually containing updated cm_values) from the client to the server and back. Updating common memories (by sending a message and waiting for a response) is expensive, but the CMS has several techniques to make synchronization extremely efficient. For example, the CMM can be told ahead of time which cm_variables are of interest to a client. Thus, when a client synchronizes common memories, it will find that the CMM has already sent it any new common memory values. Messages are still sent, but the client no longer has to wait for responses.

For more information on the theory of the CMS, read [Libes 85].

Let us get right to the usage details.

Client initialization and exit

Prior to executing any CMS calls, the CMM must be started. This is described in "Compiling/interpreting/running CMS programs" on page 11.

Before any other CMS calls, the client process should identify itself to the system.

```
int rc = cm_init("HWS","cmm_host",0);/* C */

(cm_init "HWS" "cmm_host" 0)          ; Lisp
```

In each of these examples, we will give the first form in C. Following that, we will repeat it in Lisp. In case you find this confusing, just remember that the Lisp forms all begin with a left parenthesis.

In this example we have declared ourselves as “HWS”. This name need not be distinct. However, different names are helpful when watching the commands sent to the CMM. Any C-style string may be used for a name. The only limitation is in length (see "Limits of the CMS" on page 14).

The second parameter to `cm_init` is the host on which the common memory manager is running which you wish to use. Note that there may be a CMM running on the local machine, while you use one on a remote machine. The local machine may be designated by a zero character pointer or empty string.

The third parameter to `cm_init` is an integer which indicates the debugging level. Debugging levels cause an associated level of internal information to output. For example, level 0 indicates no debugging. Larger values request more debugging info. Level 2 will give you information about messages sent and received. Level 5 will generate information about individual common memory values being manipulated. With level 9, you will get a veritable flood of garbage (that you almost certainly don’t want) including things like memory allocation, variable copying, etc. Level 10 forces every byte in every incoming and outgoing message to be displayed.

Note that debugging is only enabled when the common memory system itself has been compiled with `DEBUG` defined. See the `Makefile` for more information.

`cm_init` also performs some necessary initialization of CMS client data structures. `cm_init` returns 0 if successful. Anything else is an error. A common error is that the common memory manager is not running.

Before exiting the CMS, a process should call `cm_exit`. At `cm_exit` various data structures are cleaned up, both in the server and the client. It is particularly important that one call `cm_exit` before calling `cm_init` again. For example:

```
cm_exit();

(cm_exit)
```

Declaring and undeclaring cm_variables

In order to communicate information to the common memory, it must be stored in an object called a *cm_variable*. The name of a common memory variable uniquely identifies it throughout the CMS. The name space is not predefined in any way, except by a length restriction (see "Limits of the CMS" on page 14). Any C-style string may be used.

All variables must be declared before use. `cm_declare` is used to declare common memory variables.

```
cm_variable *date;
date = cm_declare("date",CM_ROLE_XWRITER);

(setq date (cm_declare "date" CM_ROLE_WRITER))
```

`cm_declare` returns an object that can be used when referring to this variable in the future. This object can be stored into a variable declared as type cm_variable. If `cm_declare` returns `NULL`, the declaration has failed and an error message will be printed out explaining why. Declarations can fail for a variety of reasons (e.g., bad or conflicting arguments, no space left to store values).

Once `cm_declare()` has returned an object, this object should be used whenever referring to the variable. In the case of `cm_declare`, the first argument is a string, while in all other functions the variable identifier is almost always a `cm_variable`.

The second argument of `cm_declare` specifies access rights. The available access rights are:

	may also be written as
`CM_ROLE_NONEXCLUSIVE_WRITER`	`CM_ROLE_NONXWRITER`
`CM_ROLE_EXCLUSIVE_WRITER`	`CM_ROLE_XWRITER`
`CM_ROLE_READER`	
`CM_ROLE_WAKEUP`	

CM_ROLE_NONEXCLUSIVE_WRITER tells the CMS that you would like to be able write a variable, but not block other clients from writing the same variable. CM_ROLE_EXCLUSIVE_WRITER requires that no other clients is a non-exclusive writer. CM_ROLE_READER is used if you wish to read a variable. CM_ROLE_WAKEUP causes the CMM to automatically send a client new cm_values as soon as they are received by other clients.

These access rights can be combined by logical ORing. For example, the wakeup right is always combined with at least one of the others. Conflicting or nonsense combinations should be avoided. For example, it makes no sense to declare a role as both exclusive and nonexclusive writer. Here is an example of declaring a cm_variable that you want to receive new cm_values for automatically.

```
cm_variable *date;
date = cm_declare("date",CM_ROLE_READER | CM_ROLE_WAKEUP);

(setq cmv (cm_declare "date"
            (boole 7 CM_ROLE_READER CM_ROLE_WAKEUP)
```

Processes may call `cm_declare` on the same variable but with a different role. The new role completely replaces the old role. When a variable is undeclared, the roles are implicitly forsaken. For example, suppose a process has declared itself as the exclusive writer of a variable. If the process undeclares the variable (or dies or calls `cm_exit`), another process may then declare itself as a writer of the variable.

Cm_variables may also be undeclared. From a client's point of view, undeclaring a variable causes the CMS to delete any information about that variable. For example, a client will no longer automatically receive new cm_values from the CMM, even if another CMS client continues to write them. The cm_variable is completely removed from the CMS only when every client that declared the variable, has also undeclared it (or exited). Here is an example of undeclaring a cm_variable.

```
cm_variable *cmv = cm_declare("foo",CM_ROLE_READER);
cm_undeclare(cmv);

(cm_undeclare (cm_declare "foo" CM_ROLE_READER))
```

Common memory values

Common memory variables may hold one *cm_value* (for "common memory value") at a time. Common memory values are structured in the following way.

```
typedef struct {                /* common memory value */
      char *data;
      unsigned short msize;  /* size of malloc'd space */
      unsigned short size;   /* size of used space */
      char mallocable;       /* 1 if space is malloc'd */
                             /* 0 if space is static */
} cm_value;
```

The first field, `data`, points to an arbitrary user-supplied area of memory. This memory area is not interpreted in any way by the CMS. While it is typed as a "pointer to char", you should read this is "pointer to anything". In practice, you should cast your object to an array of chars and then assign the pointer to data.

An earlier release (version 6) of this system supported user-defined types, but experience with other common memory systems have convinced us that this is "a bad thing". Indeed, there is no reason why the common memory should know the type of the data that it is storing. To provide user-defined structured values, use ASN.1 or some other suitable Presentation Layer protocol which provides for structured types that are machine independent.

`size` should be set to the length of the object in `data`. If `mallocable` is 1, CMS will allocate space using `malloc` whenever the CMS passes a value to the user. Further, if `msize` is ever smaller than the incoming value, the pointer will be `realloc`'d and `msize` increased appropriately.

If you are not prepared to handle objects larger than a given size, set `msize` yourself, and set `mallocable` to 0. Lisp is an example where this must be done, as otherwise the common memory may attempt to free a Lisp object, which would be a serious mistake since Lisp does not use `malloc` to handle its internal storage.

The address of such a structure may be passed to `cm_set_value` and similar functions.

A cm_value is declared in Lisp (via `c-declare`) along with corresponding access functions. For example, to declare and set the various elements of the structure `foo`:

```
(setq foo (make-cm_value))                    ; cm_value foo;
(setq data (new-vectori-byte 1000))   ; char data[1000];
(setf (cm_value->data foo) (ptr data)); foo.data = data;
(setf (cm_value->size foo) 1000)      ; foo.size = 1000;
```

```
(setf (cm_value->msize foo) 1000)     ; foo.msize = 1000;
(setf (cm_value->mallocable foo) 0)   ; foo.mallocable = FALSE;
```

Reading and writing cm_variables

Cm_variables may be read and written with the following calls. (Note that the effects are local to the process until the common memory is synchronized (see "Synchronization" on page 8).

```
cm_value value = {NULL,0,0,1};
cm_get_value(variable,&value);
cm_set_value(variable,&value);

(setq value (make-cm_value))
(setf (value->data) 0)
(setf (value->msize) 0)
(setf (value->size) 0)
(setf (value->mallocable) 1)
(cm_get_value variable value)
(cm_set_value variable value)
```

`cm_get_value` retrieves the value of `variable` and stores it in `value`. Similarly, `cm_set_value` updates the value of `variable` with `value`.

`cm_get_new_value` is similar to `cm_get_value`. Its syntax is:

```
boolean cm_get_new_value(variable,&value);

(cm_get_new_value variable value)
```

If `variable` has not been read since it was last set, `value` will be updated by the value of variable and `TRUE` is returned. If `variable` has been read since it was last set `value` is not updated and `FALSE` is returned.

`cm_new_value_pending` returns the same value that `cm_get_new_value` does but without any side effects. In otherwords, `cm_new_value_pending` returns `TRUE` if a new value has been written to the variable without it having been read, otherwise it returns `FALSE`. It is called as follows:

```
boolean cm_new_value_pending(variable);

(cm_new_value_pending variable)
```

Several additional functions exist for handling handshaking between superior and subordinate processes in a control relationship such as the AMRF hierarchy [Nanzetta 84]. Specifically, variables can be used for command or status. Status variables are identified by the system with the command value that they are associated with. See [Libes 84] for more information.

Variables which are command variables should be read and written with the following routines:

```
cm_set_new_command_value(variable,&value);
cm_get_new_command_value(variable,&value);

(cm_set_new_command_value variable value)
(cm_get_new_command_value variable value)
```

One utility routine is available for determining whether a new command has been received. `cm_new_command_pending` returns `TRUE` or `FALSE` depending on whether a new command has been received.

```
maybe = cm_new_command_pending(command_variable);

(setq maybe (cm_new_command_pending command_variable))
```

When a new command has been received, `cm_new_command_pending` will return `TRUE` until `cm_get_new_command_value` has been called, after which it will return `FALSE`. `cm_get_new_command_value` also returns `TRUE` or `FALSE`, depending upon whether it has detected a new command. (Lisp users can expect `t/nil` instead of `TRUE/FALSE`.)

Status variables must be written with the routine, `cm_set_status_value`.

```
cm_set_status_value(command,variable,&value);

(cm_set_status_value command variable value)
```

Status (and any other) variables may be read with the routine `cm_get_value`.

Two predicates are available that are of use to the superior process in determining which command a subordinate process' status is in response to.

```
maybe = cm_status_equal(cmd_var,stat_var,&s_value);
maybe = cm_status_synchronized(cmd_var,stat_var);

(setq maybe (cm_status_equal cmd_var stat_var s_value))
(setq maybe (cm_status_synchronized cmd_var stat_var))
```

`cm_status_equal` returns `TRUE` or `FALSE`, depending on whether or not the status variable, `stat_var`, has the value, `s_value`, and is in response to the command specified by `cmd_var`.

`cm_status_synchronized` returns `TRUE` or `FALSE`, depending on whether or not the status variable, `stat_var`, is in response to the command specified by `cmd_var`. This is very helpful to the superior process in finding out whether the subordinate process is responding the command or a different one entirely.

Synchronization

All of the `cm_get_*` and `cm_set_*` functions read or write cm_variables local to the process itself. (Refer back to discussion in "Overview of how it works" on page 2.) In order to "*synchro-*

nize" the local common memory against the master common memory of the CMM, the client must call `cm_sync`:

```
cm_sync(behavior);

(cm_sync behavior)
```

`cm_sync` takes one argument that allows several different styles of synchronization. There are three sets of orthogonal options to control synchronization style. In all cases, cm_variables that have been changed are sent to the CMM.

1. `CM_WAIT` or `CM_NO_WAIT`

   If `CM_WAIT` is specified, `cm_sync` will not return until a set of variable updates is received from the CMM. If at least one is already waiting to be received, `cm_sync` will return. `CM_WAIT` is the default. Note that at least one variable must be declared `CM_ROLE_WAKEUP` in order for the CMM to automatically send any variable updates.

   If `CM_NO_WAIT` is specified, `cm_sync` checks for any pending updates waiting to be received before returning.

2. `CM_WAIT_FOR_ALL` or `CM_WAIT_AT_MOST_ONCE`

   If `CM_WAIT_FOR_ALL` is specified, `cm_sync` will read all pending updates that have been sent from the CMM. `CM_WAIT_FOR_ALL` is the default. This behavior is particularly useful when a reader is much slower than a writer and is only interested in the latest values of a variable. By using `CM_WAIT_FOR_ALL` it does not have to process data only to find out that it is out of date.

   If `CM_WAIT_AT_MOST_ONCE` is specified, `cm_sync` will read at most one pending update that has been sent from the CMM, even if more have been sent. This behavior is useful when, for example, a server is receiving requests in a single command variable common to all of its clients.

3. `CM_WAIT_READ`

   If `CM_WAIT_READ` is specified, the CMM will immediately respond with the latest values of all variables that have been changed since the CMM last sent any updates to the client. This is a desirable behavior if you expect to poll intermittently for a variable that is being regularly set.

   If the CMS has already sent variable updates to you, `CM_WAIT_READ` will get whichever ones are appropriate depending upon the other options you have supplied in `cm_sync`. For example, specifying both `CM_WAIT_READ` and `CM_WAIT_FOR_ALL` gives an exact simulation of a true common memory. Note that using `CM_WAIT_READ` may be slower than other forms of `cm_sync` since you may have to wait for a reply message from the common memory.

It is important to realize that the CMM will only send automatic updates if one of the variables you have declared as `CM_ROLE_WAKEUP` has been changed. At that time, all variables that have been changed (whether or not they have been declared for wakeup) are sent also.

To combine options, bitwise-OR them together. For example, to poll for at most one new set of variable values:

```
cm_sync(CM_NO_WAIT|CM_WAIT_AT_MOST_ONCE);
```

```
(cm_sync (boole 7 CM_NO_WAIT CM_WAIT_AT_MOST_ONCE))
```

However, it is expected that most clients will simply want to use:

```
cm_sync(CM_WAIT);

(cm_sync CM_WAIT)
```

`cm_sync` returns either 0 (normal completion) or negative numbers denoting an error (see the file `cm.h` for a complete listing).

Waiting for common memory and other I/O at the same time

Using `cm_sync` alone, it is possible to wait or poll for common memory activity. However, `cm_sync` will not wait for other sources of I/O activity. If you need to do this, you can call select (or another function that does so) yourself, as long as you pass to it the dedicated common memory socket (called `cm_server_socket`).

For example, the following code fragment assumes that `bit_mask` already contains other I/O descriptors of interest, and that we wish to wait for common memory activity as well.

```
extern int cm_server_socket;
main() {
      FD_SET(cm_server_socket,&bit_mask);
      select(...,&bit_mask,...);
}
```

When `select` indicates there is something to be read from `cm_server_socket`, we call then call `cm_sync` which is guaranteed not to block. (This is actually used in example 3 - see"Longer examples" on page 10.)

Longer examples

The CMS source directory contains many sample programs. Each program is usually paired with one or more other samples. For example, `client2` should be run with `server2`. A complete listing of sample programs is as follows:

client1a, client1b, server1y, server1z - illustrate the behavior of different cm_sync options

client2, server2 - demonstrate passing C types in a machine-dependent way

client3, server3 - demonstrate CMS integrated in Sun windows (via cm_server_socket).

client4, server4 - demonstrate cm_declare, cm_undeclare, cm_init and cm_exit

client6, server6 - demonstrate AMRF-style mailboxes in C and Lisp client8, server8a, server8b - demonstrate handling of SIGPIPE

Compiling/interpreting/running CMS programs

Two libraries are necessary for using common memory in the Berkeley UNIX environment. `libcm.a` is the common memory client code. This uses a lower-level communications library, `libstream.a`, which provides connection and packet service on top of TCP [Libes 89]. Both libraries normally live in `/usr/local/lib` (or `/depot/sundry/arch/lib` if you are using our "depot" structure).

Thus, to link common memory programs:

```
cc foo.c -lcm -lstream
```

`#include` files live in `/usr/local/include/cm` (or `/depot/sundry/include/cm`). Normally, it is necessary only to include `/usr/local/include/cm/cm.h` as follows:

```
#include <cm/cm.h>
```

Some compilers might need to be told explicitly to look in this directory.

`.lisp` files are stored in the same directory. There is one file provided to initialize common memory from lisp, `cm.lisp`. Thus to use common memory, you should execute the following:

```
(load 'cm/cm_user.lisp)
```

The common memory manager itself, `cmm`, lives in `/usr/local/bin` (or `/depot/sundry/arch/bin`).

The CMM should be started before any client processes call `cm_init`. Any user can run the CMM. It does not require root permissions, nor does it need to be started from the same user-id as any of the client processes. Just type:

```
/usr/local/bin/cmm
```

Normally, nothing else is required, however the CMM can take some arguments to modify the default behavior. These follow the usual UNIX (using `getopt`) conventions.

```
-d [0-9]
```

This `-d` flag will cause the CMM to print out debugging information. For more information on debugging levels see "Client initialization and exit" on page 3.

```
-t seconds
```

The `-t` flag will cause the CMM to block waiting for up to this time period when the kernel queue is full while the CMM is trying to send a message to a client. After the timeout expires, the CMM goes on and will retry later. The default timeout period is 5 seconds.

Such a situation, usually indicates the client is hung. It can also be caused if the client is consuming updates from the CMM too slowly. This situation can be remedied by disabling wakeup service and using CM_WAIT_READ.

```
-p port_number
```

The `-p` flag specifies that the initial connection port for the CMM to use. The default is 1525. (Choose values in consultation with your system manager, so as not to interfere with other Internet servers.) This is useful if you want to have multiple separate common memories on a single computer.

The CMM does not require a controlling terminal to run. Note, that if the CMM is killed, all the processes using it will terminate if they are writing at the moment that the CMM is killed. This is due to a SIGPIPE being sent to each of the clients. If you do not want this behavior, you should surround calls to `cm_sync` with a `setjmp`/`longjmp` alarm, just the way one normally does with blocking `writes`. (The common memory manager does this internally to protect itself from the clients dying. You can look at it in `man.c`.) In typical use, however, people do not do this since the common memory never dies of its own accord.

Additional Lisp notes

In Lisp, all functions are identical to their C counterparts. Common sense dictates usage differences. A small Rosetta stone will be presented:

```
date = cm_declare(......);
foo = cm_declare(.....,CM_ROLE_READER);
cm_sync(WAIT);
cm_set_value(date,"12 Dec...");
cm_get_value(foo,foolist);

(setq date (cm_declare 'date CM_ROLE_XWRITER))
(setq foo (cm_declare 'foo CM_ROLE_READER))
(cm_sync WAIT)
(cm_set_value date "12 Dec...")
(setq foo (cm_get_value foolist))
```

Note that uppercase values denote constants that should be evaluated before use (i.e. unquoted). For example, to check if cm_declare() returns without failure the code would look like:

```
(cond ((eq CM_BAD_OBJECT (cm_declare ....)) nil)
      (t t))
```

Porting code over to the VAX (HCSE)

The following section is only appropriate to people using BBN's HCSE code on the VAX running VMS. BBN's HCSE (see "Introduction" on page 1) was a simulation package that contained a common memory system as part of it. The HCSE became very popular if only because people used the common memory subsystem. Unfortunately, the HCSE common memory is quite primitive in some respects to the CMS. Nonetheless, one of our desires was to be able to write code that is portable to both systems. This has not been completely acheived, but it is possible to run the same code on the Sun (Sun CM) and the VAX (BBN HCSE) if certain steps are taken.

An interface library is supported that effectively replaces the Sun CM user calls with calls into the HCSE library. This library is currently available in `amrf.cme.nist.gov:~libes/cm/src.7v`.

Code using the the UNIX CMS library with HCSE should add the following parameters to the link command (in a `.opt` file)

```
user1:[libes.cm.src$5n7v]suncmlib/lib,
cm_library:cmlib/lib,
psect=cm_shrmem,page
```

Versions of the code compiled for debugging are available by specifying:

```
user1:[libes.cm.src$5n7v]suncmlib_debug/lib,
cm_library:cmlib_debug/lib,
psect=cm_shrmem,page
```

There are three restrictions upon use of the HCSE with CMS code.

1. Types

   The type systems in both the CMS and the HCSE system are quite different. The HCSE types are based on Praxis. Primarily this means that they have user-definable types. Secondly, the HCSE has no arbitrarily-sized data objects.

   Two extra parameters on the `cm_declare` statement exist in the HCSE version of CMS to get around this.

   The first is a maximum size. The second is a pointer to a type structure. If the type structure pointer is zero, the size is used to automatically select a type structure. For more information about creating type structures, see [Johnson 82].

   A typical call that would be portable to both the CMS and HCSE looks like the following:

```
        if (!(date = cm_declare("data",CM_ROLE_XWRITER
#ifdef VAX11C
                ,1000,0
#endif
                        ))) {
                printf("declare of var failed\n");
                exit(-1);
        }
```

   When compiled by the DEC C compiler, the additional two parameters will be included.

2. The second difference in the CMS system and the HCSE is that the HCSE does not support queuing of variable updates (see "Synchronization" on page 8). This means that if one process goes to sleep while a second process writes a variable several times, if the first process then wakes up, it will see only the last values written by the second process, not all the intermediate ones.

   A different explanation of this phenomenon is that there is no difference between specifying `CM_WAIT_AT_MOST_ONCE` and `CM_WAIT_FOR_ALL` in `cm_sync`. There is no way to get around this.

3. The third difference is that the HCSE does not store command associations with variables in the CMM itself. (In fact, they are just dropped on the floor). Therefore, routines like `cm_status_equal` don't work.

   If you need command associations or their effect, you must do what other HCSE users do. Namely, stuff a number in front of every variable indicating who many times this variable has been written. Then create a new variable, that holds the old value of the number which you can use to compare it against.

   For more information on this, read [McLean 85-1] and [McLean 85-2].

A package that implements this along with current mailbox types in use in the AMRF lives in `goon.cme.nist.gov:~network/mailbox`. An example using these functions is in the cm source directory in `vws.c`. `vws.lisp` is a lisp version of the same thing.

Limits of the CMS

Certain limitations exist in the CMM. It is possible to change most of these and recompile. Changeable limitations (and their defaults as the system is distributed are):

```
CM_MSGSIZE   100000       /* Max size of any single set of
       variable updates between the cmm and a user */
CM_SLOTSIZE  20000        /* Max size of any one value */
CM_PROCESSNAMELENGTH   20/* Max length of process name */
CM_VARIABLENAMELENGTH  20/* Max length of any var name */
CM_MAXPROCESSES        256/* Max processes that can talk to
       cmm simultaneously. Absolute max of 32 (or number
       of user file descriptors) under older versions of
       4.2BSD. */
```

Errors

Most types of errors are reported to the user program. Some messages cannot be reported back to the user, and are reported at the CMM itself. Some errors are serious enough that they are reported at both the user and CMM.

User errors

Most user errors are due to programmer error and can be fixed when identified. For example, writing into a variable declared to be read-only would be a user error. In such case, the CMS usually prints out a message indicating the problem. It also returns an error code if possible. It is sometimes not possible to do this. For instance, the above example would not be detected until after `cm_set_value` returned.

The actual message would be printed from `cm_sync` when it is processing incoming messages from the CMM. Most types of errors are detected by `cm_sync`.

System errors

System errors are caused by limitations in the CMS itself, the environment it is running in, and the user demands upon the system. Often, these cannot be avoided. For example, if the user attempts to send too much data to the common memory at once, the maximum message size can be exceeded.

Listing and explanation of error messages

cm_init:

```
return(E_CM_INIT_FAILED)
initport(client): Connection refused
```

Problem: CMM is not running.

cm_sync:

```
failed to send msg to cmm.  cmm disappeared?
```

Problem: cmm died. Detected while writing to it.

```
cm library (version #) is older/newer than cmm (version #)
```

Problem: cmm is a different version than the libraries your code is compiled with. This can also be caused by a corrupted message. This is usually identifiable by wildly different version #s.

```
bad slot encountered...aborting msg
user_decode_slot: unknown slot type (#)...msg aborted
```

Problem: corrupted message or internal error in CMS.

```
CMM: error processing variable <name> - error message
```

Problem: CMM detected error “error message” in processing “name”. See below.

```
get_slot_read_response: <name> unknown (sent from cmm)
```

Problem: corrupted message or internal error in CMS

```
too much data for msg!!
output msg size = #  slotsize = #
```

Problem: User value is too large for CMS configuration. Either user error, or message size limit should be increased.

```
cm_sd_free() called on nonmallocable object?
```
Problem: internal error in CMS

*:

```
error: bcopy src/dest is null ptr
```
Problem: internal error or user error. If user error, check elements of cm_value structures to see that they are consistent.

cmm:

```
bind() failed
initport(server): Address already in use
failed to initialize connection socket
```
Problem: another cmm is running, or a process already has the CMM connection socket open.

```
get_variables(name) failed
```
Problem: too many variables in CMM.

```
process <name> is being antisocial on fd #
```
Problem: process has requested wakeup service but is not listening to CMM updates.

```
slot bad
```
Problem: corrupted message or internal error in CMS

```
slot error in <name>  type # - error message
```
Problem: corrupted message or internal error in CMS or user error. See error message. This message is sent back to the user. See below.

Error messages generated by the CMM and sent back to the user:

```
version
```
Problem: version mismatch. See above.

```
bad slot type
```
Problem: corrupted message or internal CMS error.

```
not enough common memory to declare variable
```
Problem: too many variables stored at cmm.

```
cannot get nonexclusive write access
```
Problem: a process has already received exclusive write access to this variable.

```
undeclare of undeclared variable
```
Problem: a nonexistent variable is being undeclared.

```
variable has not been declared
```
Problem: attempt to read/write variable not yet declared.

```
not declared as writer
```
Problem: attempt to write variable declared as read-only.

```
get_slot_write: cm_flat_to_sd() failed!  no space?
```

Problem: cmm ran out of memory trying to read user message. Indicates lack of system resources or user sent value that was too large.

`not declared as reader:`
Problem: attempt to read variable declared as write-only.

There are several places in the CMS where memory is dynamically allocated. These may fail with an error such as:

```
func: failed malloc(object,size)
```

or

```
resized failed! - out of space
```

where `func` is the CMS function calling `malloc`, `object` is the object being `malloc`'d and `size` is the size of the object. These errors typically indicate that either:

1. the user is storing or receiving incredibly lengthy values (probably by mistake, but see "Limits of the CMS" on page 14), or
2. the system is running out of internal space.

How to get the software - FTP

This software may be retrieved by ftp to the Internet site `durer.cme.nist.gov` using the account name: `anonymous`. The software is called `pub/cm` and is stored as a directory of compressed shar files.

Retrieve all the files. `uncompress` and `unshar` them. Edit the `Makefile` so that the destination directories are appropriate for your installation. Then type: `make install`. If you want to play around with the examples, you may do so before installation by typing: `make` <sample program names>. Several other alternatives should be evident by examining the `Makefile`.

Comments or problems with the software may sent electronically to `libes@cme.nist.gov` or `cme-durer!uunet!libes` or mailed to:

Don Libes
National Institute of Standards and Technology
Bldg 220, Rm A-127
Gaithersburg, MD 20899

References

[Johnson 82] "Hierarchical Control System Emulation User's Manual", Timothy Johnson, Stephen Milligan, Thomas Fortmann, NBS-GCR-82-413, Bolt Beranek and Newman, Inc., October 1982.

[Libes 84] "Implementing Command-Feedback Variables in a Hierarchical Environment", Don Libes, Internal NBS Memorandum, 1984.

[Libes 85] "User-Level Shared Variables", Usenix Conference Proceedings, Portland, Oregon, June 1985.

[Libes 89] "Packet-oriented Communication Using a Stream Protocol or Making TCP/IP on Berkeley UNIX a Little More Pleasant to Use", Don Libes, NIST IR 90-4232, January 1990.

[McLean 85-1] "AMRF Notice 85-1: Command-Status Message Structure (1/8/85)", Chuck McLean, Internal NBS Memorandum, January 1985.

[McLean 85-2] "AMRF Architectural Decision Document: Mailbox Format (2/20/85)", Chuck McLean, Internal NBS Memorandum, Feb. 1985.

[Nanzetta 84] "Update: NBS research facility addresses problems in set-ups for small batch manufacturing", Phil Nanzetta, *Industrial Engineering*, pps 68-73, June, 1984.