IEN-164

CMOS SYSTEM OVERVIEW

Jerry Stern, Greg Ruth, and Jack Haverty

Bolt Beranek and Newman Inc.

January 12, 1981

1. Introduction

CMOS is a multiprogrammed real-time operating system for BBN's C-machines. It is essentially a reimplementation of MOS, (1) a PDP-11 operating system developed by SRI. Whereas MOS is written in Macro-11 assembly language, CMOS is written in C. (2)

Programming support for CMOS is provided by the UNIX operating system. CMOS itself, as well as program modules written to run as CMOS processes, are edited and compiled on the UNIX operating system, and object modules are loaded into the target machine. Since both the development and target machines support C-based code, it is also feasible to do some initial debugging in the time-shared environment.

CMOS is a small, simple operating system that provides the following basic features:

- multiple processes
- interprocess communication/coordination
- asynchronous I/O
- memory allocation
- system clock management.

CMOS development was motivated by the desire to produce a C-machine operating system suitable for use in communications-oriented applications. In light of favorable experience with MOS, it was decided to adapt a version of MOS for the C-machine. The choice of C as a system programming language was dictated by the specific nature of the C-machine. The C-machine is a microprogrammed, 20-bit machine which has an architecture explicitly designed to support the C language. The C-machine comes in two models: the C/50, which has a 1-megabyte (1 "byte" = 10 bits) physical address space and no memory management; and the C/70, which has a 2-megabyte address space and memory management. Versions of CMOS have been developed for both of these machines, as well as for the LSI-11 and the Z8000.

---

(1) Kunzelman, R. C., J. E. Mathis, and D. L. Retz, "Progress Report on Packet Radio Experimental Network, Quarterly Technical Report No. 8."

(2) Kernighan, B. W. and D. M. Ritchie, "The C Programming Language", Prentice-Hall, Inc., 1978.

It should be noted that, although CMOS is targeted for use in several applications, it has not yet been used anywhere and should be considered as still under development.

One major motivation for the creation of CMOS was to provide an operating system for the C machines, for use in applications where the memory limitations make LSI-11 approaches unsuitable. Although CMOS will run on LSI-11s, this is not an intended use. CMOS systems may use processes which exceed the address space of the LSI-11 architecture, and can only be run on C-machines or other machines which support the needed memory.

The important aspects of the CMOS design are described in the following sections. The final sections provide a detailed description of CMOS primitives and general system generation information.

2 General Design Considerations

The design and programming of CMOS have been motivated by goals of style, clarity, and consistency rather than a desire to achieve ultimate efficiency. This is not to say that efficiency issues have been ignored. CMOS is quite compact and efficient by virtue of its simplicity. Design principles and programming practices have not been compromised, however, for the sake of saving every possible byte or cpu cycle.

CMOS is designed to be an "open" operating system. This means that no distinct division exists between the operating system and the application program. One can view the operating system as a collection of library routines. The operating system can be easily extended by adding new routines and can be reduced by excluding unneeded routines. The programmer is not confined to the outermost interface presented by the operating system. If appropriate, the programmer can directly access lower-level interfaces.

Although CMOS is intended primarily for C-machines, it is designed to be a portable operating system. The use of a high-level language is, of course, the principal factor in CMOS portability. Small size and simplicity are other important factors. The design attempts to minimize the amount of machine-dependent code and to segregate it into separate modules. The I/O system design allows for easy replacement of device-dependent modules. Versions of CMOS exist for the PDP-11 and the Z8000 computers.

CMOS does not support either virtual memory or virtual address spaces. The entire system shares a single, physical address space. This lack of sophistication is due, in large part, to the nature of real-time systems. Programs and data must be continuously available in main memory in order to meet response-time requirements. Thus, virtual memory techniques such as swapping or paging are not suitable for real-time applications.

The issue of virtual address spaces is more complicated. The most common reason for providing virtual address spaces in real-time systems is to overcome an architectural deficiency of the computer. Many computers have small address spaces, yet can support a much larger amount of physical memory. Therefore, multiple address spaces are required to take advantage of larger memory sizes.

The C-machines do not suffer from this architectural defect. The C/50 provides a one-megabyte address space and the C/70 twice that. This is sufficient for all currently envisioned CMOS applications. For this reason, the current CMOS does not need, and does not support, virtual address spaces. For other machines (e.g., the PDP-11), address space limitations are more severe. On these machines, CMOS may be limited to a class of smaller applications.

Other applications may motivate further extensions to CMOS, to introduce process isolation using memory mapping, dynamic process creation, preemption, or other additions to the basic functionality.

The use of a single address space gives CMOS several important advantages over multiple address space systems. First, the single address space is a major contributing factor to the overall simplicity of CMOS. Not only is the operating system relieved of address space management chores, but also, programming and debugging are generally facilitated. Second, data sharing among processes is direct, convenient, and efficient. In multiple address space systems, memory sharing is often a difficult problem. Third, I/O devices have direct access to all of memory. In multiple address space systems, I/O devices are typically restricted to a single address space. This often produces a need for extra data copying, especially in connection with DMA devices. Fourth, an entire CMOS system is linked together as one composite program. This means that non-identical processes can still share a single copy of common subroutines. Multiple address space systems usually cannot match this level of code-space efficiency. The large address space provided by CMOS obviates the need to artificially split systems into a number of

processes because of the address space limitations.

3 Process Management

CMOS processes are defined at compilation time. They cannot be dynamically created or destroyed during system operation. For each process, a set of basic attributes is specified including a name, an initial program entrypoint, and a stack size.

CMOS employs a rudimentary process scheduling method. Three process states are defined: (1) running; (2) ready to run; and (3) waiting for an event. A running process always runs to completion. This means that the processor is relinquished only by explicit action of the running process. It is never taken away by the operating system. There is no time-slicing or other form of preemption. The next process to run is selected by a simple round-robin algorithm. All processes have a uniform scheduling priority.

This non-preemptive scheduling discipline has important implications. First, processes must be designed not to monopolize the processor for long time periods. Otherwise, crucial tasks may fail to be serviced in a timely fashion. Second, critical program sections (i.e., program sections that can be safely entered by only one process at a time) need no explicit protection. The absence of preemption guarantees the integrity of critical program sections.

Interrupt handling creates a separate class of critical sections that are not protected by the scheduling discipline. These critical sections exist only within the operating system and are of no concern to application programs. CMOS protects these critical sections in the standard manner (viz., by temporarily disabling interrupts).

4 Interprocess Communication

CMOS processes communicate with one another by passing messages known as "events". For this purpose, the operating system provides primitives called "signal", "wait", and "recv". The signal primitive permits a process to send an event to another process. The wait primitive permits a process to wait for an event that may or may not have arrived. The recv primitive permits a process to receive an event that has already arrived.

An event message contains the sender process ID, an event ID, and one word of unspecified data. The event ID is used to indicate the type of event. Both the wait and receive primitives allow a process to select the event IDs of immediate interest. The meaning of the data word depends on the event type. It is quite common for the data word to contain a pointer to a larger data structure.

CMOS provides a facility that helps to automate event processing activities. A process can designate a procedure to be the event handler for a particular event type. Thereafter, the event handler becomes active whenever a special wait primitive, called "waith", is invoked. For each event received, waith checks to see if an event handler has been defined. If so, the event handler procedure is automatically dispatched. This frees the caller of waith from the responsibility of having to deal with events not of direct interest. Processing of these events can be viewed as a background activity.

5 Input/Output

CMOS provides an asynchronous I/O facility. To perform I/O, a process creates an I/O request block (IORB). The IORB identifies the target device, the type of operation (e.g., read, write, abort), and information relevant to the particular operation (e.g., buffer areas for data transfer). The IORB also specifies an event ID. To initiate processing, the IORB is passed to the operating system. When the request is completed, the operating system signals an event to the requesting process. The event message contains the event ID taken from the IORB and a data word that contains the address of the IORB. In this way, the requesting process can easily associate the completion event with the original request. Status information is returned in the IORB.

A process can direct I/O to a specific device or to a special "primary" device. Primary devices are defined on a per-process basis and can be either assigned (to a specific device) or unassigned. If a process attempts to perform I/O on an unassigned primary device, the process is suspended until a primary device is assigned. This permits a single device to be moved from one process to another and thereby provides a simple way to share a terminal among several processes.

The core of the CMOS I/O system is a device-independent module, "eior" (enter I/O request), that provides a centralized interface between the application program and the various device driver modules. As described above, this interface accepts IORBs from the application program. The IORBs are automatically queued

on a per-device basis. If desired, requests from different processes can be interspersed for the same device. When a device becomes ready to accept the next request, the first IORB in the device queue, if any, is passed to the appropriate device driver module.

All device driver modules provide a standardized interface expected by the core I/O system. This interface consists of four entrypoints: (1) a configuration entry; (2) an initialization entry; (3) a request entry; and (4) an interrupt handler entry. A system configuration table specifies the driver configuration entry for each device. During system initialization, the configuration entry is invoked to obtain the other three driver entrypoints, and the size of any per-device data base required by the driver. The initialization entry is invoked automatically before the first IORB is passed to the driver. The request and interrupt handler entries perform standard device control functions. At present, CMOS includes driver modules for asynchronous terminals and for 1822 network interfaces.

6 Memory Allocation

CMOS includes routines that allocate and deallocate blocks of memory from a free storage pool. Both the operating system and the application program share a common pool. Three allocation options are available to control operating system behavior in the case of an allocation failure: (1) return an error code; (2) wait for more memory to become available; and (3) cease system operation.

CMOS provides an allocation mechanism only, not an allocation policy. The policy, of course, is the responsibility of the application program. In practice, however, few application programs incorporate a memory allocation policy that eliminates the possibility of free space exhaustion. Instead, some applications include a recovery mechanism to deal with this problem. It is reasonable to expect that such a mechanism will depend upon the continued functioning of the operating system. Therefore, the operating system must not itself become immediately disabled as a result of free space exhaustion.

To prevent disablement, CMOS depends on "reserve storage pools". A separate reserve storage pool is created for each type of object needed by a crucial function. The operating system uses two such pools, one for event messages and one for timer queue entries. Reserve storage pools are managed by special allocation and deallocation routines. The special allocation routine first attempts to obtain space from the common pool. If this fails, space is taken instead from the reserve pool and the

caller is so informed. If the reserve pool is exhausted, the system dies.

System primitives that use reserve storage pools return an indication of when reserve storage has been tapped. An application program can therefore detect free space exhaustion by this means or by the direct failure of a simple allocation request. At this point, the operating system will continue to function for a period of time (or number of calls) determined by reserve storage pool sizes.

7 System Clock

CMOS provides a clock management facility that maintains a time-of-day clock and permits processes to set "alarms". An alarm is simply an event that is signalled by the clock manager after a specified time period has elapsed. Both the event ID and the data word of the event message are specified by the process that sets the alarm. An alarm can be either a one-time alarm or an interval alarm that is automatically repeated at regular intervals.

8 Software Development Tools

All programming support for CMOS software development is now provided by the UNIX time-sharing system, via the UNIX C compiler and linker. BBN has developed a Version 7 UNIX and a C compiler/linker to run on the C-machines.

The same hardware configuration of a C-machine can support both the UNIX and CMOS systems, although not simultaneously, of course. We plan to use the UNIX system development tools to create CMOS systems, which can then be run and tested by bootstrapping the CMOS code in place of UNIX on the same or different hardware.

9 Future Development

There is a variety of possible extensions to CMOS, which take advantage of the increased flexibility provided by the hardware base. We intend to pursue these as specific applications arise which require additional functionality.

The most interesting category of extensions involves the use of the memory mapping hardware available for C machines. In the standard C-machine configuration, the 20-bit address space provides access to a physical memory of 1 Mbyte.

Within this physical address space, processes can share any or all of the memory, since the process address space is also 1 Mbyte.

The memory mapper hardware extends the machine's capabilities in two ways. The first extension provides for support of 2 Mbytes of physical memory. Each process is, however, limited to 1 Mbyte of address space. The second extension lies in the ability of the memory map to support eight independent active process maps. This creates an environment in which processes can share portions of their address spaces with the system or other processes, with fast context switching between the eight active processes. This removes two of the basic limitations we have encountered in real-time designs based on PDP-11 architectures, namely, the granularity of memory sharing and the speed of context switching.

The CMOS environment has not yet been extended to utilize these additional facilities, although we anticipate that this effort will begin soon.

10 CMOS System Calls

This section describes CMOS system calls available to the application programmer. These calls are divided into two major groups, low-level functions and higher-level functions. The low-level functions correspond roughly to the MOS interface, and the higher-level functions provide certain additional capabilities. The usage of each system call is described in terms of the C language. Two typedefs are first defined and then referenced by a few of the system call descriptions.

```
typedef struct {                    /* event message buffer */
        char msevent;               /* event ID */
        char mssender;              /* sender process ID */
        int msdata;                 /* user data */
} MSG;
```

```
typedef struct iorb {           /* I/O request block */
        struct iorb *irnextp;   /* ptr to next IORB on chain */
        int irdevid;            /* device ID */
        char irevent;           /* completion signal event */
        char irpid;             /* requestor's process ID */
        char *irbufp;           /* buffer ptr */
        char irport;            /* port number of request */
        char iropcode;          /* operation code */
        int irbufsiz;           /* buffer size (in bytes) */
        int irstatus;           /* status of I/O operation */
        int irnxfer;            /* number of bytes transferred */
        int irpad[2];           /* mysterious padding */
} IORB;
```

LOW-LEVEL FUNCTIONS

Process Attributes

```
Name:     getpid
Function: convert process name to process ID
Usage:    pid = getpid (name)

          char name[];  /* process name to convert
                           null name => calling process */
          int pid;      /* process ID for given name */
```

```
Name:     getpn
Function: convert process ID to process name
Usage:    pn = getpn (pid, namep)

          int pid;      /* process ID to convert
                           0 => calling process */
          char *namep;  /* place to store name */
          char *pn;     /* same as namep */
```

```
Name:     getprio
Function: get primary I/O devices of specified process
Usage:    getprio (pid, priop)

          int pid;      /* process ID, 0 => calling process */
          struct {
           int idevid;  /* primary input device ID */
           int odevid;  /* primary output device ID */
          } *priop;
```

```
Name:     setprio
Function: set primary I/O devices of specified process
Usage:    setprio (pid, idevid, odevid)

          int pid;      /* process ID, 0 => calling process */
          int idevid;   /* input device ID, <0 => no change */
          int odevid;   /* output device ID, <0 => no change */
```

```
Name:     movprio
Function: move primary I/O devices of caller to another process
Usage:    movprio (pid)

          int pid;      /* target process ID */
```

Device Attributes

```
Name:     getdid
Function: convert device name to device ID
Usage:    devid = getdid (name)

          char name[];  /* device name to convert */
          int devid;    /* device ID */
```

```
Name:     getdn
Function: convert device ID to device name
Usage:    dn = getdn (devid, namep)

          int devid;    /* device ID to convert */
          char *namep;  /* place to store name */
          char *dn;     /* same as namep */
```

Input/Output

```
Name:     eior
Function: enter an I/O request
Usage:    ec = eior (iorbp)

          IORB *iorbp;  /* I/O request block ptr */
          int ec;       /* error code */
```

Interprocess Communication

```
Name:     signal
Function: signal an event to a process
Usage:    sw = signal (pid, event, data)

          char pid;     /* target process  ID */
          char event;   /* event number */
          int data;     /* data for target process */
          int sw;       /* 1 if reserve pool used to queue
                           signal, else 0 */
```

```
Name:     wait
Function: wait for any event
Usage:    wait (msgp)

          MSG *msgp;    /* ptr to message buffer */
```

```
Name:     waits
Function: wait for a single specified event
Usage:    waits (event, msgp)

          char event;   /* desired event */
          MSG *msgp;    /* message buffer ptr */
```

```
Name:     waitm
Function: wait for one of multiple specified events
Usage:    waitm (evlist, nev, msgp);

          char *evlist; /* event list (array) */
          int nev;      /* number of events in list */
          MSG *msgp;    /* message buffer ptr */
```

```
Name:     recv
Function: receive any pending event
Usage:    sw = recv (msgp)

          MSG *msgp;    /* ptr to message buffer */
          int sw;       /* 1 if event returned, else 0 */
```

```
Name:     recvs
Function: receive a single specified pending event
Usage:    sw = recvs (event, msgp)

          char event;   /* desired event */
          MSG *msgp;    /* message buffer ptr */
          int sw;       /* 1 if event returned, else 0 */
```

```
Name:     recvm
Function: receive one of multiple specified pending events
Usage:    sw = recvm (evlist, nev, msgp);

          char *evlist; /* event list (array) */
          int nev;      /* number of events in list */
          MSG *msgp;    /* message buffer ptr */
          int sw;       /* 1 if event returned, else 0 */
```

Memory Allocation

```
Name:     alloc
Function: allocate memory block, return if not available
Usage:    blkp = alloc (nbytes)

          int nbytes;   /* size of block desired */
          char *blkp;   /* ptr to allocated block, else null */
```

```
Name:     allocw
Function: allocate memory block, wait if not available
Usage:    blkp = allocw (nbytes)

          int nbytes;   /* size of block desired */
          char *blkp;   /* ptr to allocated block */
```

```
Name:     allocd
Function: allocate memory block, die if not available
Usage:    blkp = allocd (nbytes)

          int nbytes;   /* size of block desired */
          char *blkp;   /* ptr to allocated block */
```

```
Name:     free
Function: free a previously allocated block
Usage:    free (blkp)

          char *blkp;   /* ptr to block */
```

System Clock Management

```
Name:     alarm
Function: set alarm to awaken process
Usage:    sw = alarm (event, data, delay)

          char event;   /* signal event */
          int data;     /* signal data */
          int delay;    /* timeout period in seconds/60 */
          int sw;       /* 1 if reserve pool used to queue
```

```
Name:     ialarm
Function: set alarm to awaken process at regular intervals
Usage:    sw = ialarm (event, data, interval)

          char event;   /* signal event */
          int data;     /* signal data */
          int interval; /* timeout interval in seconds/60 */
          int sw;       /* 1 if reserve pool used to queue
```

```
Name:     kalarm
Function: kill any specified pending alarms
Usage:    kalarm (event, data)

          char event;   /* event of requests to kill */
          int data;     /* data of requests to kill */
```

```
Name:     setod
Function: set time of day
Usage:    setod (time)

          long time;    /* time of day */
```

```
Name:     getod
Function: get time of day
Usage:    time = getod ()

          long time;    /* time of day */
```

HIGHER-LEVEL FUNCTIONS

Event Management

```
Name:     newev
Function: generate a new event number, unique system-wide
Usage:    event = newev ()

          char event;   /* event number */
```

```
Name:     setevh
Function: associate an event handler routine with a specified
          event for this process
Usage:    oldent = setevh (event, entryp)

          char event;   /* event to be handled */
          int (*entryp) (); /* event handler entrypoint */
                        /* if null, cancel event handler */
          int (*oldent) (); /* previous entryp, else null */
```

```
Name:     waith
Function: wait for an event; dispatch event handler if one
          is defined, else return.
Usage:    waith (msgp)

          MSG *msgp;    /* ptr to message buffer */
```

```
Name:     waitsh
Function: wait for an event; dispatch event handler if one is
          defined; else return if event is the one specified;
          else ignore event;
Usage:    waitsh (event, msgp)

          char event;   /* desired event */
          MSG *msgp;    /* message buffer ptr */
```

Synchronous Input/Output

```
Name:     read
Function: read from a specified device; event handlers are
          active while awaiting read completion.
Usage:    nbytes = read (devid, bufp, bufsiz)

          int devid;    /* device ID */
          char *bufp;   /* buffer ptr */
          int bufsiz;   /* buffer size */
          int nbytes;   /* number of bytes read */
```

```
Name:     write
Function: write to a specified device; event handlers are
          NOT active while awaiting write completion.
Usage:    nbytes = write (devid, bufp, bufsiz)

          int devid;    /* device ID */
          char *bufp;   /* buffer ptr */
          int bufsiz;   /* buffer size */
          int nbytes;   /* number of bytes written */
```

11 System Generation

The following CMOS modules must be linked into any system configuration:

cm_init	Initialization routines.
cm_data	Process control and configuration tables.
cm_util	CMOS utilities.
cm_err	Error message routines.
cm_proc	Basic process management routines.
cm_queue	Queue manipulation routines.
cm_ipc	Interprocess communication routines.
cm_mem	Memory management primitives.

cm_io	Basic I/O routines.

In addition to the required modules, the following optional modules may be included for specific hardware device support:

cm_time	Timer management routines.
cm_tty	Terminal driver routines (to be rewritten).
cm_1822	1822 driver routines (to be written).
cm_smd	Disk driver routines (to be written).
cm_mlc	MLC driver routines (to be written).

In order to include DDT the following modules must be included:

ddt_main

ddt_cmd

ddt_code

ddt_brk

ddt_sym

References

1. Mathis, J. and Klemba, K., "The Micro Operating System," Chapter 6 of Terminal Interface Unit Notebook, Vol. 2, SRI International, March 1980. <MOS reference>

2. Kraley, M. et al., "Design of a User-microprogrammable Building Block," Thirteenth Annual Workshop on Microprogramming, Colorado Springs, Colorado, 1980.

3. Ritchie, D.M. and Thompson, K., "The UNIX Time-Sharing System," Bell System Technical Journal 57(6) pp. 1905-1929 (1978).

4. Kernighan, B.W. and Ritchie, D.M., The C Programming Language, Prentice-Hall, Inc., 1978.

APPENDIX

CMOS Error Messages (C-machine version)

cvdevnm: Device not found

The mate specified for a device (in the device control table initialization data) is not the name of any existing device.

getdcte: Bad device ID

The CMOS primitive (e.g. eior, getdn) was called with an invalid device id.

dlvrmsg: NULL msg ptr

Due to an internal error (blush).

mkroom: Memory full

Insufficient space in the free memory pool to accommodate device driver data and/or process stacks during system initialization.

alloc: Invalid request

The CMOS primitive "alloc" has been called with a negative block size.

free: Invalid addr

The CMOS primitive "free" has been called with a pointer outside the free memory pool.

allocd: Allocation failed

An allocation request via the CMOS primitive "allocd" has failed.

plalloc: Pool empty

The reserve memory pool has been exhausted.

dschd: Stack overflow

A process has overrun its stack. This may be due to an excessive depth of nested procedure calls. The only solution is to reassemble the system with more stack space.

getpcte: Bad pid

A CMOS primitive was called with a non-existent process id.

mktqe: Bad delay time

A CMOS clock management primitive was given a timeout period of 0 by the caller.

In addition, there are various fatal conditions trapped by CMOS:

TRAP: invalid memory addr

TRAP: illegal instruction

TRAP: illegal micro call

TRAP: privileged operation

TRAP: register overflow

TRAP: EDAC error

TRAP: register underflow

For every trap the following machine status values are printed out:

PC = program counter

PS = program status

SP = stack pointer