PDOS Developer's Reference

PDOS Developer's Reference

Printed in the United States of America.
Product number 2520-2T (for PDOS revision 3.3)
January, 1988

Table of Contents

File Management 60

PDOS BIOS 67

Introduction

This manual is designed to help the PDOS developer understand the inner workings of the PDOS operating system. It describes the kernel, file manager and BIOS modules of PDOS in detail. This is not a beginner's manual. To learn how to use PDOS, consult the *PDOS User's Manual* Volume 1 and other PDOS reference manuals.

Before you consult this manual, you should be familiar with PDOS-related computer hardware and software, specifically the MC68000 microprocessor. If you need more information on the MC68000, consult one of the following books:

Motorola. 1984. *MC68000 - 16/32-BIT MICROPROCESSOR PROGRAMMER'S REFERENCE MANUAL*. Fourth Edition. Englewood Cliffs, N.J.: Prentice-Hall Inc..

Zarella, John. 1981. *MICROPROCESSOR OPERATING SYSTEMS*. Suisun City, California: Microcomputer applications.

Manual Organization

The first section of this manual describes the structure of PDOS. It is a good idea to read through this overview before you begin your development.

The next section deals with the kernel. The PDOS kernel handles tasking, so there you will find information on multi-tasking, the task control block, task communication, SYRAM, etc.

The file manager section follows and describes how PDOS handles file storage, file names, disk numbers, and directory levels.

The next section describes the BIOS or Basic I/O Subsystem of PDOS. If you are porting PDOS to new hardware, you will consult this section often. It goes into much detail about the user and common BIOS modules, disk read/write and layout, and the PDOS Winchester disk standard. You will also learn how to write PDOS I/O drivers from examples.

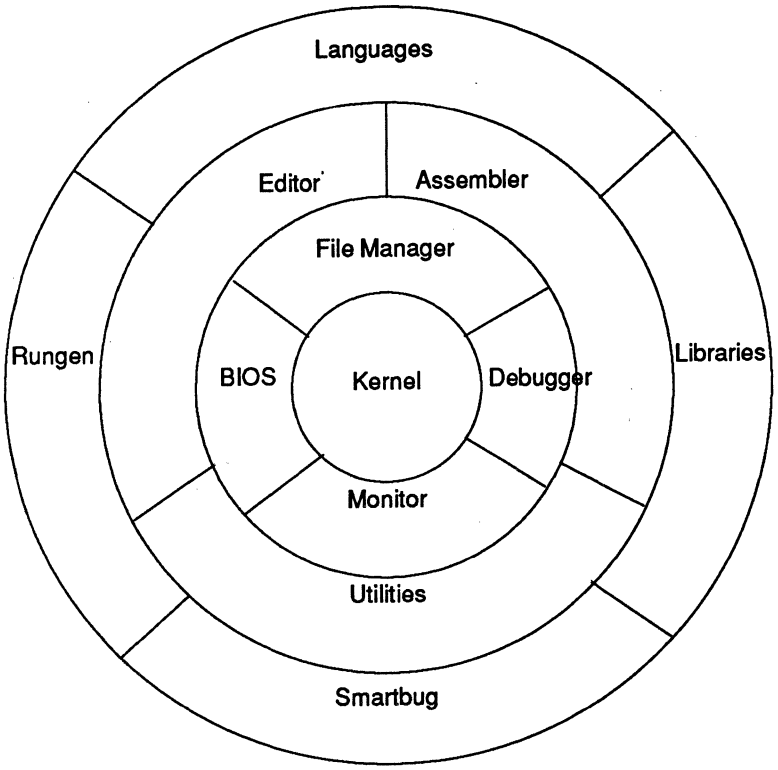
The final section lists PDOS errors with a description of each one.

Conventions

The following notations are used throughout this manual:

$	Hexadecimal number. (e.g. $1FFF = decimal 8191).
%	Binary number. (e.g. %1001101 = decimal 77).
< >	Parameter used with a PDOS command or primitive. (e.g. DL <filename> indicates that the DL command requires a filename as a parameter).
{ }	Optional. (e.g. SA <filename>{,<attributes>} indicates that the parameter <attributes> is optional).
(Ax)	Indirect assembly addressing. (e.g. (A2) = Buffer refers to register A2 pointing to a buffer).
Keys	Key names are denoted by bold. (e.g. **Cr** or ↵ means to press the carriage return key; **Esc** is the escape key; **Ctrl** is the control key (usually followed by a letter which also appears bold); and ↓ indicates a line feed).

PDOS Structure



The PDOS Kernel

PDOS is written in Motorola 68000 assembly language for fast, efficient execution. The small kernel handles multi-tasking, realtime clock, event processing, and memory management functions. Ready tasks are scheduled using a prioritized, round-robin method. The highest priority task in the ready state is always scheduled. Tasks with the same priority are scheduled in a round-robin fashion. A suspended task allows lower priority tasks to execute. The A-line ($A000) instruction interfaces over 100 system primitives to a user task.

Tasks are the components comprising a realtime application. Each task is an independent program that shares the processor with other tasks in the system. Tasks provide a mechanism that allows a complicated application to be subdivided into several independent, understandable, and manageable modules. Realtime, concurrent tasks are allocated in 2K byte increments. There are no 64K byte boundary restrictions since the full 32-bit address space is available.

Semaphores and events provide a low overhead facility for one task to signal another. Events indicate availability of a shared resource, timing pulses, or the occurrence of a hardware or software interrupt. Messages and mailboxes are used in conjunction with system lock, unlock, suspend, and event primitives. PDOS provides timing events that can be used in conjunction with desired events to prevent system lockouts. Other special system events signal character inputs and outputs.

PDOS handles all exception processing including interrupts, address errors, bus errors, illegal and unimplemented instructions, and privilege violations. Each task also has the option to process any or all 16 trap vectors, divide by zero, overflow check (TRAPV), and register out of bounds (CHK). System interrupts set the corresponding event and then can initiate a context switch. A high priority task waiting on that event is then immediately scheduled and begins executing.

The PDOS kernel handles user console, system clock, and other designated hardware interrupts. User consoles are interrupt-driven with character type-ahead. A task can be suspended pending a hardware or software event. Otherwise, a prioritized, round-robin scheduling of ready tasks occurs. Time slices are BIOS-dependent and adjustable on a task-by-task basis.

The File Manager

The PDOS file management module provides sequential, random, read only, and shared access to named files on a secondary storage device. These low overhead file primitives use a linked, random access file structure and a logical sector bit map for allocation of secondary storage. No file compaction is ever required. Files are time stamped with date of creation and last update. Up to 127 files can be open simultaneously. Complete device independence is achieved through read and write logical sector primitives.

BIOS

PDOS gives software portability systems through hardware independence of the system Basic Input/Output System (BIOS) module. All hardware functions such as read/write logical sector, clocks, mappers, and UARTs are conveniently isolated in this module for minimal customization to new 68000-based systems.

Supported Devices

PDOS is easily configured for any combination of large or small floppy disks, bubble memory devices, or Winchester mass storage devices. A wide variety of target system configurations are supported for fast development of memory-efficient, cost-effective end products.

Memory Requirements

PDOS is very memory efficient. The PDOS kernel, file manager, debugger, BIOS, and user monitor utilities require less than 26K bytes of memory plus an additional 8k bytes for system buffers and stacks. Most applications can be both developed and implemented on the target system. Further memory reduction is achieved by linking the user application to a 8K byte PDOS kernel for a small, ROMable, stand-alone, multi-tasking module. For large system configurations, PDOS effectively addresses up to the 32-bit address space of the 68000 processor.

PDOS Kernel

The PDOS kernel performs the following functions:

- Multi-tasking, multi-user scheduling
- System clock
- Memory allocation
- Task synchronization
- Task suspension
- Event processing
- Character I/O including buffering
- Support primitives

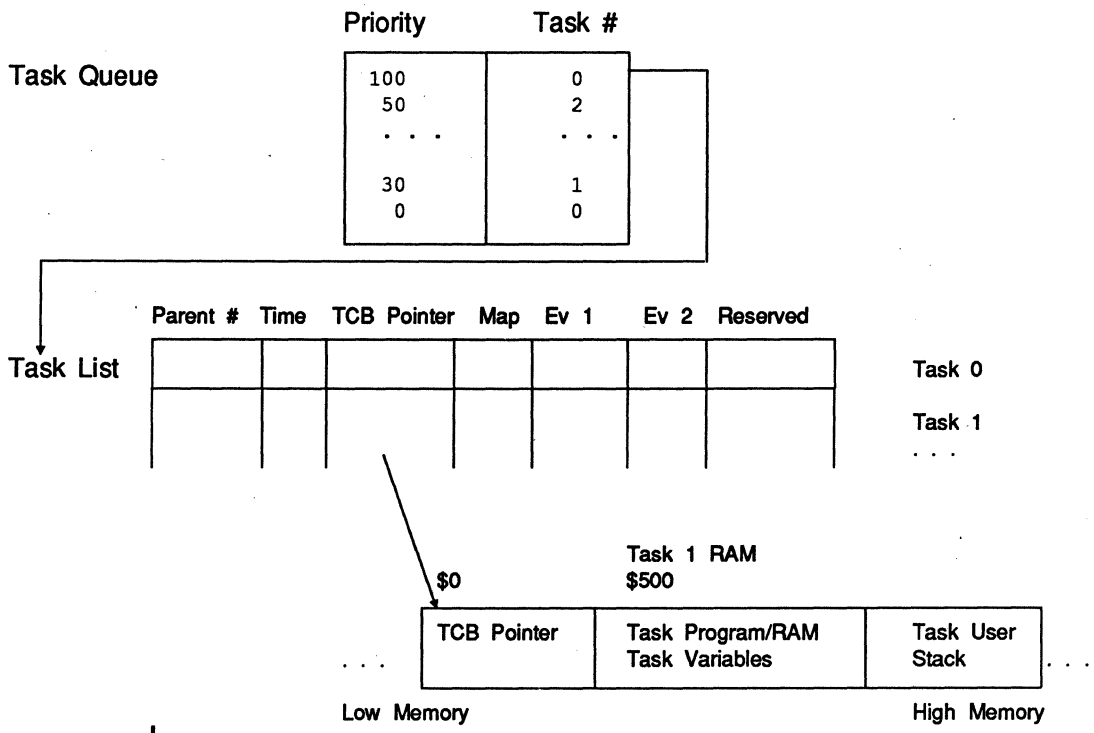
The PDOS kernel is the multi-tasking, realtime nucleus of the PDOS operating system. Tasks are the components comprising a realtime application. It is the main responsibility of the kernel to see that each task is provided with the support it requires in order to perform its designated function.

The PDOS kernel handles the allocation of memory and the scheduling of tasks. Each task must share the system processor with other tasks. The operating system saves the task's context when it is not executing and restores it again when it is scheduled. Other responsibilities of the PDOS kernel are maintenance of a 24-hour system clock, task suspension and rescheduling, event processing, character buffering, and other support functions.

The PDOS Task

A PDOS task is the most basic unit of software within an application. A user task consists of an entry in the PDOS task queue, task list, and a task control block with user program space.

The task queue and list are used by the PDOS kernel to schedule tasks. A task queue entry consists of a task priority and a task number. The list is ordered with the highest priority entry first. A task list entry consists of a parent task number, task time slice, task control block pointer, task map constant, two suspended event descriptors, along with other reserved information. The task number is assigned according to its entry position.



The first $500 (hex) bytes of a task are the task control block. This block of memory consists of buffers and parameters peculiar to the task. The 68000 address register A6 points to the status block when the user program is first entered. The task parameters may be referenced by a user program but you must be careful not to crash PDOS!

Task overhead = $500 (hex) bytes + user stack

The user program space begins immediately following the task control block. Position independent 68000 object programs or BASIC tokens are loaded into this area for execution. Task memory is allocated in 2k byte increments. The total task overhead is $500 or 1280 bytes. This leaves $300 or 768 bytes available for a user program and user stack in a minimal 2k byte task.

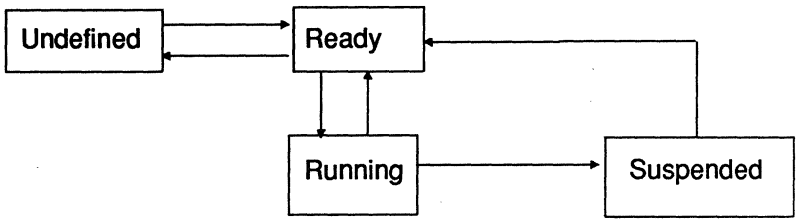
From the time a task is coded by a programmer until the task is destroyed, it is in one of four task states. Tasks move between these states as they are created, begin execution, are interrupted, wait for events, and finally complete their functions. These states are defined as follows:

Undefined. A task is in this state before it is loaded into the task list. It can be a block of executable code in a disk file or stored in memory.

Ready. When a task is loaded in memory and entered in the task queue and list but not executing or suspended, it is ready.

Running. A task is running when scheduled by the PDOS kernel from the task list.

Suspended. When a task is stopped pending an event external to the task, it is said to be suspended. A suspended task moves to the ready state when the event occurs.



A task remains undefined until it is made known to the operating system by making an entry in the task queue. Once entered, a task immediately moves to the ready state which indicates that it is ready for execution. When the task is selected for execution by the scheduler, it moves to the run state. It remains in the run state until the scheduler selects another task or the task requires external information and suspends itself until the information is available. The suspended state greatly enhances overall system performance.

Multi-Tasking

PDOS defaults to allow 32 independent tasks to reside in memory and share CPU cycles. Each task contains its own task control block and thus executes independently of any other task. A task control block consists of buffers, pointers, and a PDOS scratch area. By changing the "NT" parameter in MSYRAM and other parameters, PDOS can be configured to handle up to 127 tasks.

Four parameters are required for any new task generation. These are:

- A task priority. The range is from 255 (highest priority) to 1 (lowest priority).
- Tasking memory. Memory is allocated to a task in 2k byte increments. The first $500 bytes are assigned to the task TCB.
- An I/O port. Input ports are unique while many tasks may share the same output port for task console communication.
- A task command. This may be in the form of several monitor commands or a memory address to begin executing.

Each of the previous requirements defaults to a system parameter. Task priority defaults to the parent task's priority. Default memory allocation is 32k bytes and default console port is the phantom port.

If a task command is not specified, the new task reverts to the PDOS monitor. However, if no input is possible (i.e. port 0 or input already assigned), then the new task immediately kills itself. This is very useful since tasks automatically kill themselves as they complete their assignments (remove themselves from the task list and return memory to the available memory pool).

A task entry in the task list consists of a task number designation, parent task number, time interval, task priority, memory map constant, task control block pointer, and two event descriptors. Swapping from one task to the next is done when the task interval timer decrements to zero, during an I/O call to PDOS, or when an external event causes a context switch. The task interval timer decrements by one every ten milliseconds (or as defined in the system BIOS module).

Any task may spawn another task. Memory for the new task is allocated in 2k byte blocks from a pool of available memory. If no memory is free, the spawning task's own memory is used and the parent task's memory is reduced in size by the amount of memory allocated to the new task. It is important to note that some assembly coded programs and all high level language programs use both the low and high addresses of the task memory. To prevent memory loss from a task and program failure, it is necessary to allocate enough memory to the free memory pool before creating a new task under program control. Otherwise, the task may give up its variable space or stack to the spawned task.

PDOS maintains a memory bit map to indicate which segments of memory are currently in use. Allocation and deallocation are in 2k byte increments. When a task is terminated, the task's memory is automatically deallocated in the memory bit map and made available for use by other tasks.

The Task Control Block (TCB)

Although the locations of the task control block are made available to the user, you must be cautious when using these locations. Many PDOS primitives use these locations to perform their functions and any location may change at any time as a result of these PDOS calls. The TCB may be modified significantly at any time.

The Task Control Block contains most of the system variables that are specific to a task, including various buffers, I/O parameters, and vectors. When a program begins execution under PDOS the system automatically initializes register A6 to point to the TCB. If the register is overwritten by the user, a program can later recover it by executing the XGML (get memory limits) primitive.

Task Status Control Definitions

```
$000(A6) = 256 byte user buffer
$100(A6) = CLB$    - 82 byte monitor command line buf
$150(A6) = MWB$    - 32 byte monitor work buffer
$170(A6) = MPB$    - monitor parameter buffer
$3B0(A6) = TSP$.L - task stack pointer
$3B4(A6) = KIL$.L - kill self address
$3B8(A6) =        - reserved
$3BC(A6) = SVF$.B - save 68881 registers flag
$3BE(A6) = TRP$    - user TRAP vectors
$3FE(A6) = ZDV$.L - zero divide trap
$402(A6) = CHK$.L - CHCK instruction trap
$406(A6) = TRV$.L - TRAPV instruction trap
$40A(A6) = TRC$.L - trace vector
$40E(A6) = FPA$.8 - floating point accumulator
$416(A6) = FPE$.L - fp error processor address
$41A(A6) = CLP$.L - command line pointer
$41E(A6) = BUM$.L - beginning user memory
$422(A6) = EUM$.L - end user memory
$426(A6) = EAD$.L - entry address
$42A(A6) = IMP$.L - assigned input message pointer
$42E(A6) = ACI$.W - assigned input file ID
$432(A6) = LEN$.W - last error number
$434(A6) = SFI$.W - spooling unit file ID
$436(A6) = FLG$.W - task flags
$437(A6) = SLV$.B - directory level
$438(A6) = FEC$.B - reserved
$43A(A6) = CSC$.W - clear screen character(s)
$43C(A6) = PSC$.W - position cursor characters
$43E(A6) = SDS$.B - alternate system disk(s)
$441(A6) = SDK$.B - system disk
$442(A6) = EXT$.L - XEXT address
$446(A6) = ERR$.L - XERR address
$44A(A6) = CMD$.B - command line delimiter
$44B(A6) = TID$.B - task ID
$44C(A6) = ECF$.B - echo flag
$44D(A6) = CNT$.B - output column counter
$44E(A6) = MMF$.B - memory modified flag
$44F(A6) = PRT$.B - input port #
$450(A6) = SPU$.B - spooling unit mask
$451(A6) = UNT$.B - output unit mask
$452(A6) = U1P$.B - unit 1 port #
$453(A6) = U2P$.B - unit 2 port #
$454(A6) = U4P$.B - unit 4 port #
$455(A6) = U8P$.B - unit 8 port #
$456(A6) =        - reserved
$458(A6) = TW0$.W - monitor word temps
$45A(A6) = TW1$.W - TW0-TW2 used by level
$45C(A6) = TW2$.W -    2 primitives
$45E(A6) =        - reserved
$470(A6) =        - debugger parameters
<<<<< $500(A6) <<<<<<< USER PROGRAM
```

Assembly language programs normally access the individual fields of the TCB as offsets from A6. The definitions are shown in the previous table. The naming convention is a three-letter name followed by a dollar sign (e.g. the "units" field is defined as "UNT$(A6)"). The PDOS assembler, MASM, recognizes and defines these offsets if you declare the option "PDOS" at the beginning of your program.

The startup module for C programs "CSTART:ASM" automatically saves the TCB pointer in a variable called "_tcbptr" (note the leading underscore). To gain access to this variable, C programs need the include file "TCB:H."

The startup module for Pascal programs "PMAIN:SR" saves the TCB pointer in a variable defined as PTCB. Pascal programs need to include both "SYVAR:INC" and "TCB:INC" to use this variable.

FORTRAN programs accessing the TCB must use the XGML function to pick up the pointer. Here, as in BASIC, it is necessary to correctly specify the offset of the desired field. FORTRAN programs use the functions BYTE, WORD, and LONG to read or write the TCB variables.

BASIC programs have access to the TCB pointer through the variable SYS[9]. To access any of the fields in the TCB, however, a BASIC program must add the specific offset of that field and be careful to use the correct width for the variable (byte, word, long, etc.) BASIC programs use the primitives MEM, MEMW, and MEML to read or write the TCB variables.

The following examples show how a program can modify its own unit parameter to direct console output to the port defined as unit 2. The specific offset of the UNT field is $451 or 1105 (decimal). When you run any of these programs, all console output will be directed to the unit 2 port. Type UN 1↵ to restore console output to normal (nothing will echo as you type).

Assembly

```
        OPT    PDOS           ;LET ASSEMBLER RESOLVE TCB FIELD NAMES
START   MOVE.B #2,UNT$(A6)    ;FORCE A 2 INTO THE UNIT FIELD
        XEXT                  ;BACK TO THE MONITOR
        END    START
```

C

```
#include 'TCB:H'                /* define tcb offsets and tcbptr */
main()
{
      _tcbptr->_unt = 2;        /* set the unit to 2 */
}
```

Pascal

```
TYPE                              {define these field types }
    PTR = ^INTEGER;
    BYTE = -127..127;
    WORD = -32767..32767;

{$F=SYVAR:INC}                    {define the Pascal system variables}
{$F=TCB:INC}                      {define the TCB fields}
VAR
    TCB : TCBPTR;                 {copy of the pointer to the TCB}
PROCEDURE GETSYV(VAR SYVAR:SYVARPTR);EXTERNAL;
BEGIN
       GETSYV(SYVAR);             {point to the Pascal system variables}
       TCB := TYPEOF(SYVAR^.PTCB,TCBPTR);    {init my pointer to TCB}
        TCB^.UNT := 2;            {set the unit to 2}
END.
```

FORTRAN

```
      PROGRAM TEST
      INTEGER TCB,DUMMY
      PARAMETER (UNT=1105)                          !DECLARE THE FIELD
      CALL XGML(DUMMY,DUMMY,DUMMY,DUMMY,TCB)        !ONLY WANT THE TCB
      BYTE(TCB+UNT) = 2                             !SET THE UNT TO 2
      END
```

BASIC

```
10  UNT=0451H                 !DECLARE THE FIELD OFFSET
20  UNIT 2                    !THIS DOES IT AS A BASIC INTRINSIC
30  REM                       OR
40  MEM[SYS[9]+UNT]=2         !THIS DOES IT DIRECTLY
```

There are a number of fields in the TCB that may be of use to the advanced programmer. Be aware, however, that the definition of the TCB has changed in the past as PDOS has evolved, and changes are likely to take place in the future. If you can accomplish your purpose through more straight-forward techniques, it would be a good idea to do so. But there are times when modification of TCB variables is the only way to do what you need to do.

The TCB appears to the program as a record in memory. For those languages that support records (C, Pascal), you merely define a pointer to a record and then modify the field by name. The alternative is to look at the TCB as a collection of bytes/words/long-words and modify the TCB by reading and writing those bytes. The technique in assembly is to move data to and from various offsets of A6. In FORTRAN, the BYTE/WORD/LONG intrinsics do the job, and in BASIC, the MEM/MEMW/MEML intrinsics will work.

The following TCB definitions use C for the examples. You must use an approach consistent with your programming language.

```
/*000*/       char _ubuf[256];       /* 256 byte user buffer */
```

The first 256 bytes of the TCB are a general purpose buffer called the "user buffer." This buffer is used by many of the system primitives that do file access; generally the disk directory is read into this buffer and scanned for your file name whenever you open a file. You may use this buffer for anything that you want to do, but be aware that PDOS primitives use it too. It can be destroyed very easily.

```
/*100*/  char _clb[80];  /* 80 byte monitor command line buffer*/
```

At offset $100 is the monitor's command line buffer (CLB). The CLB is used by the monitor every time it gets a new command or retrieves a task message. The PDOS primitive XEXZ copies a text string into the CLB and exits. The XGLM primitive uses the CLB as its buffer automatically. The RUN primitive in BASIC uses the CLB to hold the message. Thus, the TCB fields CMD and CLP usually reference the CLB. You might want to parse data in the CLB if the XGNP primitive does not do the job you want. Another use for the CLB might be to inspect various tasks and see what task each one is running; the information is usually in the task's CLB.

```
/*150*/ char _mwb[32];       /* 32 byte monitor work buffer    */
```

The monitor work buffer is at offset $150. This is the buffer used by the monitor to build temporary strings or file name records. The primitive to parse a file name into its constituent parts, XFFN, leaves the name in the MWB. XFFN is usually followed in system calls by XRDN -- read directory name -- and XRDN expects input in the MWB. Most application programs do not use either XFFN or XRDN directly, but they call them indirectly by using the open file primitives (XNOP,XSOP,XROP, XROO), or the close file with attribute (XCFA) primitive, read file attribute (XRFA), write file attribute (XWFA), delete file (XDLF), rename file (XRNF), define file (XDFL), zero file (XZFL), file altered check (XFAC), or write file parameters (XWFP). The XCFA primitive is used by copy file (XCPY) and the XRFA primitive is used by load file (XLDF). The append file and copy file both use the open primitives. The MWB is also used by the PDOS conversion routines that need to return a string. Thus, the date/time conversions and the binary to ASCII conversions usually modify the MWB. These include the following:

XRDT - Read Date
XUAD - Unpack ASCII Date
XRTM - Read Time
XUTM - Unpack Time
XCBH - Convert Binary to Hex
XCBD - Convert Binary to Decimal
XCBM - Convert Binary to Decimal with Message

Access to the MWB is important to you if you use any of these functions, because you must be careful that they do not interact -- if you need both XRTM and XRDT, use the information provided by the first call before you perform the second or it will be overwritten. There are also a few occasions when you might want to access the MWB directly -- for instance, you might want to construct an output file name by parsing the input file name while substituting a different extension. Use the XFFN call to break the name up; then change the extension in the MWB and put the name back together.

```
/*170*/       char _mpb[60]; /* monitor parameter buffer        */
```

The MPB is used to store procedure file parameters or for command line recall -- but not both. If you are inside a procedure file, the MPB contains the strings used to expand the symbols &1-&9. These symbols are stored as nine null-terminated strings. Outside a procedure file, this buffer contains as many of the previous command lines as is possible to save in 60 bytes. The two uses of the MPB are the reason that you cannot recall a command line if a procedure file is involved. Also, since only one buffer is used, the procedure file parameters are not local to nested procedure files but global. If one procedure file calls another procedure file (with arguments), the expansion of the &1-&9 symbols will change for both.

The line input calls reference the contents of the MPB if they are executed from within a procedure file. The monitor alters the MPB with every command executed (when not in a procedure file). A **Ctrl A** from the monitor rolls the MPB to get the last command entered. A program may push commands into the MPB (so that the **Ctrl A** can retrieve them) with the XPCB call. Typing **LT 2** at the monitor displays the contents of the MPB buffer for all tasks. The line on the top of the list is generally the last command line typed. You might want to access the MPB if you want to write procedure file primitives that modifies the parameters. For instance, you could write a program called "TSTFIL" that would take the name of a file on the command line. This program could then look up the file on the disk, return a status in &0 if the file did not exist, parse the name into name, extension, level, disk, type, size, etc. and place those values into specific procedure file parameters for successive procedure file commands to use. Or, you could write a GETSTR program that pauses the procedure file while it reads a string from the keyboard. It then would put the string into a procedure file parameter to determine the next step of the procedure.

As an example, try the following program, named "TEST:"

```
START   LEA.L   A(PC),A1
                XPCB
                 LEA.L  B(PC),A1
                XPCB
                LEA.L    C(PC),A1
                XPCB
                LEA.L    D(PC),A1
                XPCB
                XEXT
A                DC.B   'This',0
B               DC.B    'is',0
C               DC.B     'a',0
D               DC.B    'test',0
                EVEN
                 END    START
```

This program is executed from the following procedure file "X:"

```
*&1 &2 &3 &4 &5 &6 &7 &8 &9
TEST
*&1 &2 &3 &4 &5 &6 &7 &8 &9
```

The following example shows an invocation of "X" with parameters:

```
3>X 1,2,3,4,5,6,7,8,9
3>*1 2 3 4 5 6 7 8 9
3>TEST
3>*test a is This 1 2 3 4 5
3>
```

```
/*1AC*/       char _cob[8];  /* character out buffer          */
```

This is a special buffer used by different PDOS primitives for temporary character strings. For instance, the XPSC primitive builds the output string for the position cursor sequence in this buffer.

```
/*1B4*/       char _swb[508];/* system work buffer/task stack */
```

Every task in PDOS has its own set of stacks -- the user stack is typically located at the end of user memory, and grows backwards towards the beginning of memory. The supervisor stack is located here in the TCB -- at the end of the SWB and growing back to the beginning. Whenever an interrupt or trap occurs (PDOS calls included) that puts the CPU into privileged mode, this stack is used. There are a couple of internal PDOS calls that also use the other end of the system stack for a temporary workspace.

```
/*3B0*/       char *_tsp;    /* task stack pointer            */
```

This is the top of the supervisor stack, SWB (see previous note).

```
/*3B4*/       void *_kil;    /* kill self pointer             */
```

The kill self pointer is a hook that allows a task to specify a special exit when it runs out of input. If a task is created in background mode (i.e., no input port) it will kill itself when it runs out of things to do. If this is not the desired action, you can have the task jump to another set of instructions by putting the proper address in the KIL vector. PDOS checks for procedure file input, unexecuted "FE" commands, data from the IMP pointer, and a valid PRT. If there is no input from any of those, it executes an XCTB -- unless the KIL field is non-zero. In that case, it jumps to the address given in KIL.

```
/*3B8*/       long _sfp;     /* system frame pointer  */
```

The system frame pointer is used by the "FE" monitor command. The FE command creates a list of expanded commands and puts them out at the end of memory. It then moves the end of memory down (EUM) so that other programs executing will not overwrite these commands. The SFP is used as part of the process of retrieving the commands.

```
/*3BC*/       char _svf;      /* save flag--68881 support (x881)*/
```

Some programs may require special hardware registers to be saved when a context switch occurs. If this is the case, the program must execute an X881 PDOS call (which sets the SVF flag to -1). On a task swap, if the SVF flag is set, PDOS calls the BIOS routine B_SAV. By default this is used to save the floating point state of the 68881 hardware co-processor, but it could be altered by the user to save any important context. When a task is swapped in, the same flag is checked, and the BIOS routine B_RES is called to restore the context. Whenever a task exits to the monitor, the flag is cleared.

```
/*3BD*/       char _iff;      /* RESERVED FOR INTERNAL PDOS USE  */
```

This flag is reserved for PDOS use only.

```
/*3BE*/       void *_trp[16];         /* user TRAP vectors          */
```

The sixteen TRAP instructions cause a software interrupt through a special set of vectors in low memory. PDOS tests the current task, and if the TRP vector corresponding to the trap number has a non-zero value, PDOS jumps to that address. The subroutine performs as if it had been called by a simple JSR instruction -- just exit by means of an RTS. The return address on the stack is the address of the instruction immediately after the TRAP instruction, so you may pass parameters to the trap handler by putting pointers in-line. Since a TRAP instruction is only 16 bits, it may act as a short subroutine call. Several instructions must be executed by PDOS in order to get it to the subroutine, so it is not a fast subroutine call.

```
/*3FE*/       long _zdv;      /* zero divide trap                */
```

The hardware zero divide trap works similarly to the TRAP vectors above. The user subroutine is called in user mode, as if a JSR had been issued.

```
/*402*/       long _chk;      /* CHCK instruction trap           */
```

Trap vector for the CHCK instruction -- see above.

```
/*406*/       long _trv;      /* TRAPV;instruction trap          */
```

Trap vector for the TRAPV instruction -- see above.

```
/*40A*/       long _trc;      /* trace vector                    */
```

If the trace trap bit is set in the status word, the system makes a call through this vector. The debugger uses this vector when tracing. In addition, SMARTBUG relies on this vector for the PB, XBUG, and breakpoint entries.

```
/*40E*/       long _fpa[2];  /* BASIC floating point accumulator*/
```

This flag is reserved for use in BASIC only.

```
/*416*/       void *_fpe;     /* fp error processor address      */
```

Reserved for use in BASIC only.

```
/*41A*/       char *_clp;     /* command line pointer            */
```

This pointer keeps track of how much of the command line has been seen by XGNP. Normally, the pointer indicates a location in the CLB, but it may be reset to point to any string in memory. The use of this pointer also depends on the value of the CMD variable. If CMD contains a null or a period, XGNP will parse no further. If CMD contains a space or a comma, XGNP will parse to the next delimiter, insert the delimiter in CMD, replace the delimiter in the string with a null, advance the CLP to point after the null, and return to the beginning of the parsed string. If the monitor finds a null in the CMD field, it reads a new line into the command line buffer (CLB) and resets the CLP to point to the CLB.

```
/*41E*/       char *_bum;     /* beginning of user memory        */
```

The beginning of user memory is defined as the first available memory that does not have information already loaded into it. When you load a program into memory, the BUM is set to the address after the last location loaded. It is thus the first memory available for variable allocation.

- XLDF -- resets the BUM to note the last address loaded.
- XGML -- returns the current value of the BUM as one of the parameters.
- XCTB --initializes the BUM for the new task.
- "ZM" -- resets the BUM to the start of task memory for the task.
- "SV" -- defaults to saving the memory from the start of task memory to the BUM.
- "LT 1" -- reports the current value of the BUM (second value after "TCB=").

```
/*422*/       char *_eum;     /* end user memory                 */
```

The end of user memory defines the end of the stack. Whenever a task enters the monitor state, PDOS resets the user stack pointer to the current value of the EUM minus 2. XGML returns a value called the "upper memory limit" equal to the EUM minus 128. XCTB initializes the EUM for the new task. "FE" loads its commands out at the end of memory and adjusts the EUM down to protect them. "ZM" uses the EUM to mark where to stop clearing memory. "LT" displays the current value of the EUM under the column "EM". The "FM" command without parameters looks for system available memory that is adjacent to the EUM. The "GM" command adjusts the EUM to incorporate the new memory. The "FM" command with a parameter lowers the EUM after the memory has been given to the system.

```
/*426*/       char *_ead;     /* entry address                    */
```

Normally, a PDOS task will start execution at the beginning of the task. One exception to this is a BASIC program that executes in the BASIC interpreter. Object files may have an embedded starting address that is different from the beginning of file, but SY files must begin execution at the task beginning. Most of the time the entry address is the same as the start of the task -- it is set up by the XCTB (create task) primitive and can be reset by the ZM (zero memory) monitor command. The XLDF (load file) primitive will set the EAD to whatever value is appropriate for the file loaded. The GO monitor command uses the current value of the EAD as the default address to start executing. The current value of the EAD is listed by typing **LT 1** at the monitor -- it is the fourth number after the "TCB=".

```
/*42A*/       char *_imp;     /* internal memory pointer          */
```

PDOS supports I/O redirection through internal pointers in the Task Control Block. The XGCR (get char) primitive scans three sources of input for data. If the pointer IMP is non-zero, a byte is retrieved from where it points and the pointer is incremented. If the byte thus retrieved is a null, the pointer is cleared and PDOS continues looking for data. The IMP pointer is the highest level of the input hierarchy that also includes the ACI file IDs and the PRT input port. Data from this source thus will supersede data waiting from the other sources. One good use for the IMP is to provide default responses for line-oriented inputs. If a program is going to gather data via one of the get line calls, it may put a pointer to the default string in the IMP. The get line call will then read from the pointer (echoing to the screen) and at the end of the string (the string must be terminated with a null not a carriage return), PDOS will wait for a terminating carriage return from the keyboard. The user then has the option of editing the default response to provide useful changes, or entering a carriage return to take the default.

```
/*42E*/       int  _aci;      /* assigned input file ID           */
/*430*/       int  _aci2;     /* second assigned input file ID    */
```

The second level in the input hierarchy is the AC file ID -- when console input is supplied from a disk file. The normal way to re-direct input to come from a file is to execute a procedure file (type AC) from the monitor. However, a program can also modify the ACI field of the TCB to get the same effect. The file must be opened using one of the standard PDOS open calls and the file ID is saved in the ACI field. When an error occurs on input from the file (usually end of file), PDOS closes the file and clears the ACI field. The monitor will close the file early in response to an "RC" command, or a break character (**Esc Ctrl C**) from the keyboard. The second ACI field is to allow for one level of nesting -- one procedure file can call a second, but that file cannot call a third. When input terminates from ACI, the ACI2 field is checked and popped into ACI if non-zero. When a procedure file calls another, the ACI is pushed into ACI2 (if the latter is non-zero).

```
/*432*/       int   _len;     /* last error number                  */
```

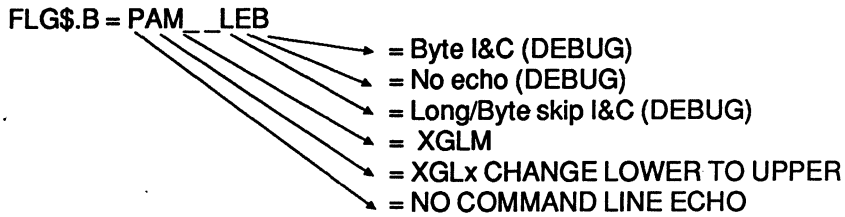
The function XLER loads an integer value into the LEN field of the TCB. The XERR function loads an error value into the LEN and exits. In the latter case, PDOS displays an error message to the screen. Programs may communicate by way of the LEN status -- generally a zero value indicates successful completion, while a non-zero value indicates some kind of error. In a procedure file, the special token "&0" is expanded to hold the current value of the LEN field. It is possible to write special programs that augment procedure file control by reading or modifying the LEN field.

```
/*434*/       int   _sfi;     /* spool file id                      */
```

PDOS output can be re-directed to a file with the SU command from the monitor. This command has two parameters -- a unit number and a file or port. If the second parameter is a port, the unit number is used to select which one of the fields U1P, U2P, U4P, or U8P should receive the specific port value. If the second parameter is a file, that file is opened and the file is placed in the SFI field. In that case, the unit number is placed in the SPU field. For any character output, the current value of the UNT variable is checked against the SPU field. If there are corresponding bits, the character is written to the file open on the SFI.

```
/*436*/       char  _flg;     /* task flags (bit 8=cmnd line echo)*/
```

The FLG field contains several "mode" bits that can be set or cleared to affect various PDOS functions. The bottom two bits are used by the debugger to vary the way memory inspect/change is done. If the least significant (bit 0) is set, the debugger will modify memory a byte at a time rather than a word at a time. If the next bit (bit 1) is set, the debugger will treat memory as "write-only" -- it will not read the locations but will write to them. This may be useful in working with some types of memory mapped I/O registers. The next three bits (2-4) are currently reserved for future expansion. Following that is a bit that is set upon entry to the XGLM function and cleared on exit of the function. The next bit (bit 6) if set, tells PDOS to convert lower case to upper case whenever one of the get line calls are performed. Since the monitor uses one of these functions, this is one way to obtain case folding on the command line. The most significant bit (bit 7) signals the PDOS monitor not to echo command lines to the monitor. The following diagram is extracted from MPDTCB:SR.



The FLG field is copied from the parent task to the child task when a task is created.

```
/*437*/        char _slv;      /* directory level                    */
```

The "LV" command sets a mask byte that indicates which files will be listed by the "LS" command when the level is not explicitly specified. It also dictates the default level when new files are created. The level specifier does not restrict access to files on levels, but it does allow you to order your disk files into logical groupings. The default level when PDOS comes up is level 1. Level 255 is a special case. If your task is set to level 255, files at all levels are visible to you. Files may be created at level 255, but it is not possible to list them by themselves -- listing the files at level 255 lists files at all levels.

The SLV field is copied from the parent task to the child task when a task is created.

```
/*438*/        char _fec;      /* reserved for PDOS internal use  */
```

This field is reserved for PDOS use only.

```
/*439*/        char _spare1; /* reserved for future use          */
```

This field is reserved for future use by PDOS.

```
/*43A*/      char _csc[2];  /* clear screen characters          */
```

The CSC field and the PSC field are designed to give some terminal independence to PDOS. If these two fields are properly initialized, the full-screen editor can run, and most other utilities that use terminal functions will work as well. The CSC field allows the user to specify up to four characters (two escapes and two other characters) to output when a program requires a clear screen sequence. The encoding is fairly simple -- load the two bytes with the characters to print. If the high bit is set on either byte, that byte is preceded by an Esc. Some terminals, then, require the sequence **Esc H Esc J** to clear the screen. The character "H" has a decimal value of 72 and a hexadecimal value of 48 in ASCII. Adding the escape bit makes it a C8. The "J" character is a 4A, which becomes a CA with the escape bit set. The entire sequence is C8CA. If the second byte is not necessary, leave it zero.

Some terminals are still so complex in their terminal sequences that this scheme is not enough. An example of this complexity is the ANSI standard terminal sequence. PDOS provides for this type of terminal by leaving a special call-out in the BIOS for terminal functions. If the CSC is zero, or if the first byte of the CSC is FF, PDOS calls the routine B$CLS in the BIOS to clear the screen. The default code in MBIOS for B$CLS performs the clear screen function for the ANSI terminal, but you may modify it to suit your own purposes. The B_CLS function needs to load the character sequence into memory using MOVE.B xx,(A3)+ instructions. A null character terminates the string. If B$CLS returns with a status of NE, PDOS simply outputs the string. Otherwise, the system expects register D0.W to contain a pair of bytes in the usual CSC format. It is possible to have code for several types of terminals in the B$CLS function by setting the first byte of the CSC to FF and using the second byte as a terminal type code.

There is a problem in associating the clear screen variable with a task rather than with the port. With virtual ports, where a task may move from one terminal to another -- it should retain the ability to clear the screen regardless of where it runs, however it does not. Also, a task may have more than one output unit active -- printing simultaneously on two or more terminals. In such a case, the task ought to be able to clear both screens with a single XCLS function, but it can't.

Refer to your *Installation and Systems Management* guide for more details about the clear screen call.

The CSC field is copied from the parent task to the child task when a task is created.

```
/*43C*/      char _psc[2];  /* position cursor characters      */
```

The PSC or position cursor sequence works with the CSC field to provide terminal independence. The characters stored here determine the lead-in sequence for the position cursor command; whether or not the row/column values are biased by a space; and whether the row or the column comes first. The two bytes of the PSC are normally output as lead-in characters for the function. If the high bit of the first character is set, the row and column values are biased by $20. (This means that position zero, zero -- the upper left corner of the screen -- will be addressed by outputting the lead-in characters followed by two spaces). If the high bit of the second character is set, the column is output first, followed by the row, otherwise, the row comes first.

As is the case with the CSC field, if the entire field is zero, or if the first byte is FF, PDOS calls the routine B_PSC in the BIOS to perform the cursor position sequence. B_PSC receives the row value in register D1.B and the column value in D2.B. Register A3 points to the buffer where the sequence should be deposited. Again, one might set the first byte of the sequence to FF and use the second byte as a function code. When B_PSC returns, if the status is NE then the string at (A3) is null terminated and sent to the terminal. Otherwise, PDOS finishes making the position cursor sequence, using D0 as the bias for the row and column, swapping them if the high bit is set on the second character of the PSC field, and storing the row and column two bytes beyond A3. The code is as follows:

```
      ADD.B   D1,D0          ;ADD ROW
       ROR.W  #8,D0
      ADD.B   D2,D0          ;ADD COLUMN
      TST.B   1+PSC$(A6)      ;SWAP?
        BPL.S @0006          ;N
       ROR.W  #8,D0          ;Y
      *
      @0006   ADDQ.W   #2,A3
      MOVE.W  D0,(A3)+       ;STORE POSITION CHARACTERS
```

One problem is that the output routine terminates on a null character. If your terminal requires that the row/column not be biased, but that the values go out directly, there may be a problem with addressing on row or column zero. In some cases, it might be possible to add $80 as a bias, since many terminals ignore the high bit. This would allow PDOS to distinguish between a terminating null and the 80 used to indicate a row or column zero. Another technique that might work would be to output the cursor positioning string within the B_PSC function, using the XPDC function. This function does not rely on a terminating null, but uses a count in register D7 to tell when to stop. In this case, the B_PSC would need to return a null string in the COB buffer (where A3 points) and a status of NE.

The PSC field is copied from the parent task to the child task when a task is created.

```
/*43E*/       char _sds[3];  /* alternate system disks          */
/*441*/       char _sdk;     /* system disk                     */
```

These four bytes indicate the "path" searched whenever any file name is specified for an open without an explicit disk designator. The search order is reversed from the storage order -- SDK is the first disk, followed by SDS[2], SDS[1], and SDS[0]. SDK is the "current" disk; the one used for files created without a disk designation, and the disk searched whenever an "LS" monitor command is given without a disk designation. The PDOS monitor outputs the contents of the SDS/SDK fields as the monitor prompt, using the B$MPT routine in MBIOS. This is entered via offset B_PDM in the BIOS. The prompt routine uses the convention that 255 is an illegal disk, and does not display that value. Thus, if only one or two numbers are given to the "SY" command, the remaining values in the SDS field are filled with FF or decimal 255. One odd consequence of this is that if you specify **SY 255**, the PDOS prompt is reduced to a single right angle bracket and there is NO default disk.

The SDS and SDK fields are copied from the parent task to the child task when a task is created.

```
/*442*/       void *_ext;    /* XEXT address                    */
```

Programs normally exit using the XEXT primitive. When PDOS performs this function, it checks the TCB variable EXT. By default, this field contains a zero, but if it contains a non-zero address, PDOS jumps to that address instead of taking a normal exit. It is thus possible for a program to specify special action to be taken on exit -- closing files and other termination sequences, for instance. It is very important that the termination routine specified by the EXT field clear the EXT field before it attempts termination, or the task will be held in a loop, continually executing the termination routine.

In the C standard library, the XEQ function loads another program image into a buffer and calls it as if it were a subroutine. XEQ alters the EXT vector to force the program to return to the caller instead of exiting. You may use the EXT vector to control the return of a task spawned with the XCTB call -- but only if you give XCTB the starting address. If you give XCTB a monitor command to execute in the form of a command string, PDOS loads the command into the command line buffer (CLB) and then does an XEXT, depending on the monitor, to parse the CLB for the next command. This means that your exit routine gets called BEFORE the task starts.

The EXT field is copied from the parent task to the child task when a task is created.

```
/*446*/       void *_err;     /* XERR address                    */
```

Most of the same things that apply to the EXT vector also apply to the ERR vector. This pointer determines where the program goes when it executes an XERR system call. If the ERR vector is zero, PDOS handles the error by displaying a message and exiting to the monitor. If the ERR field contains a non-zero address, PDOS jumps to that address instead. Although the XERR function is executed with the error number in register D0, PDOS passes it to your error trap in register D1. Since most programs can exit by either the XEXT instruction or the XERR instruction, you are generally advised to set both vectors to properly control program termination.

The ERR field is copied from the parent task to the child task when a task is created.

```
/*44A*/       char _cmd;      /* command line delimiter          */
```

The CMD field works with the CLP pointer to control the action of the XGNP (get next parameter) function. Since the monitor uses XGNP to parse the name of the next program to run, and since all programs use the XGNP function to collect command line arguments, this field can be useful to any system programs that affect program execution or parameter passing. The XGNP call can be used as a parser if its functions fit your needs.

Basically, the XGNP first examines CMD. If it contains a period or a null, XGNP does nothing, indicating no parameter available. If it contains a space or a comma, XGNP parses the string indicated by the CLP until it encounters a space, comma, period, or null. This delimiter is saved in the CMD field and a null placed in the string where it was found. Leading spaces on a parameter are ignored. An opening parenthesis disables the usual parsing for spaces, commas, or periods until a matching closing parenthesis is found. PDOS keeps a count of unclosed opening parentheses, so that they may be nested. It is thus possible to pass a group of parameters as a single parameter through XGNP.

```
/*44B*/       char _tid;      /* task ID                         */
```

The task number is also available in SYRAM and most PDOS functions use that value when they need it. This field is used primarily by application programs that need to know the current task number. PDOS does use the TID field in the following functions: the XCTB function uses the TID for the parent task field in the task list; the get line calls reference it for the &# symbol expansion in procedure files; and the monitor function "LT" determines which line gets the asterisk (indicating the current task) from the TID field.

```
/*44C*/       char _ecf;      /* echo flag                          */
```

The ECF flag is used by PDOS to disable all output without modifying the current value of the UNT variable. The ECF flag is normally set/cleared by the "EE" command from the monitor. The XERR primitive also clears the ECF flag so that output will be restored whenever an error occurs. Finally, when the monitor gets a command line that is not from a procedure file or an "FE" (For Every) frame, it clears the ECF flag. The output character routine in PDOS only looks at the high bit of the ECF to determine whether to allow output or not. This bit is set by the "EE" command whenever ANY non-zero value is loaded into the ECF.

The "GT" monitor command also sets the high bit to disable output while scanning for the label, but it restores the previous value of the ECF afterwards. The "LS" command tests the 1 bit of the ECF (set by **EE 2**). If that particular bit is set, the "LS" command appends the disk number to every file name and CLEARS the high bit for each line of output that has a file name. This makes possible a more condensed file listing -- one that does not have the usual header and footer that the "LS" command prints. Thus, **EE 2.LS ;@.EE 0** will not display the disk name, the directory size, or the summary information. It is possible to do a multiple disk listing by using the "FE", "LS", and "EE" commands together. For instance, to display all procedure files on disks 3-28 you might use the following command:

```
x>FE (3-28) EE 2[LS ;@/AC/&F]EE 0
```

The remaining bits of the ECF are undefined and reserved for future use.

```
/*44D*/       char _cnt;      /* output column counter              */
```

The CNT field in the TCB is used to keep track of the current print column on output. It is set directly by the XPSC primitive, cleared by printing a carriage return, decremented by printing a backspace, and incremented by printing a non-control character. Whenever PDOS expands tab characters to spaces, it references the current value of the CNT field. Similarly, the XTAB function uses the CNT field to determine how many spaces to print.

If a task has more than one output port (UNT not 1) then the CNT field will not be able to simultaneously maintain the correct value for all ports. This means that tabs may not expand correctly on both (or either!) ports, and the XTAB function may not perform correctly for both. There is a table in SYRAM that contains the current row and column position for every port -- this table is referenced by the XRCP function and is more likely to be accurate than the CNT field.

```
/*44E*/       char _mmf;      /* memory modified flag                */
```

The memory modified flag is used to tell what type of program executed previously, and if it is safe to re-enter it. This determines the proper action of the "GO" command from the monitor, and allows some programs to perform a different action on re-entry than on initial entry. The "GO" command checks the sign bit of the MMF. If it is set, and no starting address was given, the "GO" command simply exits. If the MMF has a zero or positive value, the "GO" command uses the EAD (entry address) as the default starting address. This gives the user the capability of leaving a program, executing a few monitor commands, and re-entering the program with memory intact. A few monitor commands, however, will alter the contents of task memory. Re-entering a program after one of these commands might cause a crash.

These commands (specifically "ZM," "FM," "TM" with a negative port, "FE," "DM," "TF," and "LL") use a potentially large amount of task memory for buffer space. Therefore they set the MMF flag to minus 1. The XCTB and XCHF calls both clear the MMF flag. XCHF is used by the monitor to start any new program, so with any program execution the MMF always starts off zero. BASIC, QLINK, and MEDIT all set special values in the MMF to let them distinguish between initial entry (when the MMF is zero) and a re-entry. The values used by these three programs are given in the file MPDTCB:SR and are as follows:

MMF$ = 1 = BASIC
2 = QLINK
3 = MEDIT

Other values for the MMF are reserved for future PDOS expansion.

```
/*44F*/       char _prt;      /* input port #                        */
```

The PRT field is used by the get character primitives (XGCR, XGCC, XCBC, XGCP, and XGCB) to select which input buffer to use for data. If the PRT has a value of zero, the task is a background task. By definition, background tasks may receive input from a procedure file or the IMP vector. If the task attempts monitor input with no input port, the task simply exits and gives up its memory to the system. If a program attempts character input with no input port, it receives an error 86 -- "Suspend on Port 0."

To avoid having multiple tasks trying to grab input from the same keyboard, XCTB keeps track of which ports have been allocated to tasks and which haven't. This record is kept in the SYRAM field, PATB. XCTB refuses to create a task with the same PRT value as another task. When a task exits with a non-zero value in the PRT, the corresponding PATB entry is cleared. If a task should clear its own PRT and then abort, the PATB entry for that task is still allocated, and PDOS will not allow you to create a task on that port.

Several additional uses are derived from the PRT field by PDOS functions. XSTM tests it to tell if a task is a background task. If so, and the task sends a task message with a negative destination, the message will be sent to the parent task. In this way, a task can create a background task for a particular function and that task can send a message to the parent on completion or other state change. If the task is not a background task and sends a task message to a negative destination, the message will come back to the originating task. This probably will be found and displayed by the monitor after the program exits.

Some PDOS primitives use the PRT as the default port if none is explicitly given. These are XSPF -- set port flag, XRPS -- read port status, and XRCP -- read cursor position. There is opportunity for error in this last case. A program may use the XRCP function to save the current cursor position, then position to a new location to write a message, and then re-position to the old position. You might, for instance, write a program to keep the current time displayed in one corner of the screen. If this program runs as a regular task it will have no problem, but if it runs as a background task (and does not explicitly specify the port to XRCP), it will not correctly read the cursor position. This is because XRCP uses the PRT value to index into the proper table, and the PRT value is zero for background tasks. Such a program should probably pick up the port number from the U1P field and use that in the specification of the port for XRCP.

One last PDOS function that makes extensive use of the PRT field is the "TM" command from the monitor. This function maps the current port to an alternate port, copying input from the current port to the output of the alternate port and vice-versa. It does this by storing one port number in the PRT and the other port number in the U1P and then reversing them. It is sometimes possible for a task to crash while in transparent mode, leaving the unfortunate user with the PRT driven by a modem port or some other inaccessible device. The MABORT program now searches for this situation and restores the PRT$ field for the task that last received the port.

```
/*450*/      char _spu;     /* spooling unit mask                    */
```

The SPU field is used to direct output to an output spool file. During character output, the UNT is compared against the SPU. If any bits correspond, the character is sent to the file ID in the SFI field. The SPU value supersedes the mapping to the U1P, U2P, U4P and U8P. So, if the SPU value is set to 2, unit 2 data will go to the spool file instead of to the U2P port. If the SPU is set to 16, however, it will not overlay any of the output ports. In that case, with valid port numbers in all four UxP fields, and a UNT value of 31 it is possible to direct character output to five different destinations!

The "SU" command from the monitor will set the SPU field if the second argument is a file name. If the second argument is a port number, it sets one or more of the UxP fields. SU 0 clears the SPU field and closes the file ID in the SFI field.

```
/*451*/       char _unt;      /* output unit mask                    */
```

The UNT field of the TCB directs character output to all, some, or none of the output sinks of the task. The output routine checks the SPU for correspondence with a copy of the UNT field. If any bits match, the byte is sent to the spool file and those bits are cleared from the copy of the UNT field. The first four bits of the copy of the UNT field are then checked one at a time. If the bit is set and the corresponding UxP field has a non-zero value the character is sent to that port. XCTB initializes the UNT field to 1.

```
/*452*/       char _u1p;      /* unit 1 port #                       */
/*453*/       char _u2p;      /* unit 2 port #                       */
/*454*/       char _u4p;      /* unit 4 port #                       */
/*455*/       char _u8p;      /* unit 8 port #                       */
```

These four fields determine the output port(s) connected to a task. If none of the fields contain a valid port number, the task does not perform terminal output -- at least not through PDOS. The U1P contains the primary output port number. When the task is created the UNT field is set to 1, making the U1P the only output port enabled. Generally, the U1P is the same as the PRT port, since most tasks use one port for both input and output. (Such does not have to be the case, however).

The U2P port is traditionally the port for printing listings. Many PDOS systems have a command in the start-up file to direct the U2P to the port occupied by the system printer. Earlier versions of PDOS did not allow the U2P values to be set by the "SU" command; rather a variation of the "BP" command (negating the port number) sets the port characteristics and assigns the port as the U2P port at the same time. This technique is disparaged now as being obscure, but it still works. The accepted technique is to type **SU 2,<port>**.

The TTA and TTS I/O drivers and the DN monitor comand all get their output unit from this field.

```
/*456*/ char _spare2[170];  /* reserved for system use             */
```

This block of memory contains special registers used by the debugger and a few temporary variables used by various PDOS primitives. Their assignment is subject to change without notice.

```
/*500*/       char _tbe[0];           /* task beginning              */
```

This "field" is actually not a field at all, but rather the beginning of the program space.

Further information may be found by studying the comments in the file MPDTCB:SR -- and by experimentation.

SYRAM

There are a number of fields in the TCB that may be of use to the advanced programmer. However, be warned that in the same way that the TCB determines the functioning of an individual task, so does the SYRAM block determine the functioning of an entire PDOS system. SYRAM is the variable space for the PDOS kernel. Within SYRAM are contained all the system parameters that must vary -- i.e., those that can't be coded into EPROM. The PDOS system itself is coded in position-independent assembly language. There is only one absolute value assumed in the entire system and that is the pointer to SYRAM (a value labeled B$SRAM and usually set by the xxDOS:GEN procedure file). Once an interrupt routine has found SYRAM it can find anything else in the PDOS system.

An application program generally has no need to modify variables in SYRAM. Programs changing SYRAM variables may crash the system and will probably not be portable to other PDOS systems. Future versions of PDOS may alter the structure of SYRAM, making things difficult for programs that depend on its present structure. The current structure of SYRAM is always defined in the assembly language module MSYRAM:SR. You should check the MSYRAM:SR file or the *Installation and Systems Management* guide for your system for any differences from the description given below.

SYRAM consists of three main parts. The first part contains fixed variables and tables of standard size which are pre-initialized by the BIOS. The second part of SYRAM also contains fixed variables and tables of standard size, but are pre-initialized by PDOS. These SYRAM offsets do not change and are included as assembler reserved words. The third part of SYRAM has the SYSGEN value-dependent tables, where the offset location of each table depends on a variable.

Many of the values in SYRAM are determined by conditional assembly symbols in the files MBIOS:SR and MSYRAM:SR. These may be set by modifying the actual source code of these files, or by defining the symbol when performing a sysgen. In the following discussion, if a variable table depends on such a symbol, the name of the symbol and the defining file are given.

Fixed Offset BIOS Initialized

```
/*000*/        char *_bios;          /* address of BIOS ROM   */
```

The first pointer in SYRAM indicates the start of the BIOS table. Since the BIOS code has (at least at the start) a fixed structure, this enables programs to get at the routines for the different types of I/O. For example, the driver file TTA picks up the BIOS pointer to get the BIOS UART table. After indexing to the appropriate table (based on the port type), it retrieves a pointer to the specific BIOS "putc" entry point. There are a number of fields in the BIOS that can be useful to a systems program. The MBIOS:SR file defines the BIOS structure.

```
/*004*/       char *_mail;            /* mail array address    */
```

In order for BASIC programs in different tasks to communicate, there needs to be a special memory area set aside outside of tasking memory. BASIC defines this area as an array, and initializes the first long word as an array descriptor. Programs in other languages may use the mail array to pass information back and forth, but if BASIC is in the system it would be safest to avoid the first long word.

The size of the mail array is determined by the MBIOS symbol "MSZ" The default size is 256 bytes. The mail array is allocated at the end of tasking memory.

```
/*008*/       int _rdkn;              /* RAM disk #            */
```

This variable holds the current number of the RAM disk. The RAM disk can be dynamically mapped to any number by way of the "RD" monitor command. The initial value of the RDKN field is set by the MBIOS symbol "RU". The default value is 8.

```
/*00A*/       int _rdks;              /* RAM disk size         */
```

The size of the RAM disk is given in multiples of 256 bytes. This is the size of a disk block under PDOS. The default size of the RAM disk is 255 blocks. The MBIOS symbol "RZ" sets the initial value of this parameter. Later, the second parameter of the "RD" monitor command can set it to any size desired.

```
/*00C*/       char *_rdka;            /* RAM disk address      */
```

The starting address of the RAM disk defines where in memory the RAM disk begins. The initial value of RDKA is set by the MBIOS symbol "RA". If the initial address is zero and the initial size is non-zero, PDOS allocates the RAM disk from the end of tasking memory and calculates the starting address after sizing RAM. You might want to set the RAM disk address if you have a separate memory card that you want to dedicated to your RAM disk. The RAM disk address may also be set by the third parameter of the "RD" monitor command.

```
/*010*/       char _bflg;             /* BASIC present flag    */
```

This flag is set during startup to the value of the FBA symbol (in MBIOS). It indicates whether or not BASIC was linked into the system. If there is no BASIC interpreter present, the "EX" command from the monitor returns an error 77, as does any attempt to run a file of type "EX" or "BX". The default depends on whether or not you have linked with a version of PDOS containing BASIC.

```
/*011*/      char _dflg;              /* directory flag      */
```

This flag is set during startup to the value of the FDR symbol (in MBIOS). Its value is zero unless the user has chosen to make file names local to the current directory level. Normally, any program can access any file on the specified disk without regard to the level number of the file. If the directory flag byte is set minus ($80), you can only access a file on a level different from your own by specifically specifying the level. Two files may have the same name on the same disk if the directory flag is set and they are on different levels.

```
/*012*/      int  _f681;              /* 68000/68010 flag    */
```

The 68000 and 68008 differ from the other processors in the information saved on the stack during an exception. The same version of PDOS runs on both the 68000 and the 68010, but to account for the difference in the exception handling, PDOS tests the processor during initialization to determine the processor type. The "F681" flag is then set to 0 to indicate a 68000 or to 2 to indicate a 68010. Any modifications to this flag will probably cause a crash.

```
/*014*/ char *_sram;          /* run module B$SRAM           */
```

The linker defines the symbol B$SRAM to be the pointer to the start of SYRAM. This is so that exception processors can load the pointer without referring to registers. In a ROM environment, it may be most convenient to put the pointer itself in SYRAM, since there may not be another place available. RUNGEN uses this value to produce run modules. In a RAM environment, the location of SYRAM itself may change from time to time, and putting B$SRAM inside SYRAM would be like locking the key inside the safe. Therefore, a fixed location is used in the xxDOS:GEN file to provide a pointer to SYRAM.

```
/*018*/ int spare1;           /* reserved for expansion      */
```

This location is reserved for future use by PDOS.

Fixed Offset PDOS Initialized

```
/*01A*/      int  _fcnt;              /* fine counter        */
/*01C*/      long _tics;              /* 32 bit counter      */
```

There are two counters in the PDOS system. Both are incremented once per clock tic (generally every hundredth of a second, but it may be different). The fine counter counts up to 1 second of time and is cleared. The tics counter runs endlessly and rolls over at the maximum 32-bit number.

```
/*020*/     char _smon;          /* month          */
/*021*/     char _sday;          /* day            */
/*022*/     char _syrs[2];       /* year           */
/*024*/     char _shrs;          /* hours          */
/*025*/     char _smin;          /* minutes        */
/*026*/     char _ssec[2];       /* seconds        */
```

The PDOS system clock keeps track of the current day, month, and year, as well as the hour, minute, and second. They are stored as 8-bit integers and incremented as is appropriate.

PDOS does not account for leap-year in its roll-over. This may mean that it is necessary to manually reset the clock every four years. PDOS also does not try to keep track of Daylight Savings Time. The SYRS and SSEC fields have an extra byte at the end in order to put the time and date onto 16-bit boundaries. These unused bytes are reserved for PDOS use.

```
/*028*/     char _patb[16];     /* input port allocation table*/
```

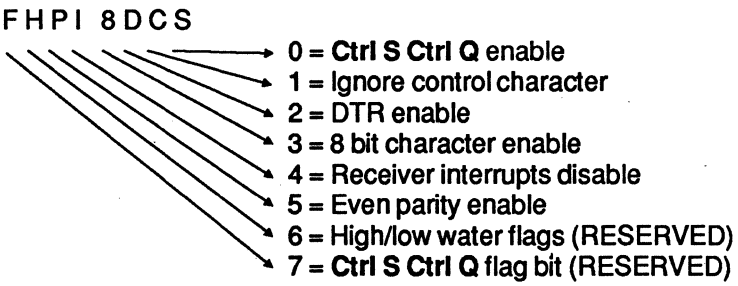
PDOS attempts to enforce a limitation that only one task can own the keyboard on any given port. The port allocation table serves this purpose by keeping track of what tasks have allocated what ports for input. The XCTB (create task) may request an input port for the new task. Before creating the task, PDOS checks the PATB to see if that port is already allocated. If the port is already allocated to another task, PDOS sets the new task's PRT value to zero. The task number is always saved as the binary complement of the actual task number. Thus, task 0 is saved as FF, task 1 as FE, etc.

```
/*038*/     char _brkf[16];       /* input break flags   */
```

Two characters in PDOS are considered "break characters". These are **Ctrl C** and **Esc** (decimal values 3 and 27). Whenever the character input routine detects that one of these characters has been pressed, the BRKF table entry corresponding to that port is set to a -1 (for an **Esc**) or +1 (for a **Ctrl C**). The XCBC (check for break character) primitive tests this table and returns a status indicating its value.

```
/*048*/     char _f8bt[16];              /* port flag bits  */
```

PDOS allows some control of characteristics of an I/O port through the "BP" monitor command and the XBCP primitive. In addition to setting the data rate on the port these calls allow the user to configure the port with various options. The F8BT table assigns an 8-bit status to each of the PDOS ports. Each status byte has the following structure:



Bit zero, if set, tells PDOS to use XON/XOFF handshaking on the port. If a **Ctrl S** is detected on input, PDOS stops output to that port until a **Ctrl Q** is seen. If PDOS gets behind on processing input and this bit is set, a **Ctrl S** will be transmitted to stop the other device from transmitting.

Bit one, if set, tells PDOS that the port is not to perform special input processing on the data stream. This means that **Esc** and **Ctrl C** characters are treated as regular data and do not set the break flag. The buffer clear character, **Ctrl X**, is also disabled by this bit.

The virtual port (window) switching character is also not disabled by this bit; you should not have this port enabled for virtual ports either, if you plan to send/receive binary data.

Bit two tells PDOS to perform hardware handshaking. If the hardware supports it, PDOS looks for a ready status on the DTR line before outputting, and will drop its own DTR signal when it needs to signal another computer to stop sending data. Check the xxBIOSU:SR file on your system to see what this bit does.

Bit three tells PDOS to send and receive eight bits of data at a time on the port, rather than seven. Some terminals may send or receive the eighth bit under special circumstances. Similarly, eight bit data transmission may be required for some communications protocols.

Bit four, if set, disables receiver interrupts on the port. PDOS allows non-interrupt received characters and the XGCR primitive "polls" the UART directly if the buffer is empty and this flag bit is set.

Bit five tells PDOS to enable parity on the port and to use even parity. PDOS does not support odd parity, although you might set the appropriate bits in your xxBIOSU:SR file and send odd when even is requested. The rest of PDOS does not use parity and provides no error handling; this is only a signal to the BIOS modules.

Bit six is reserved for internal use by PDOS. It signals that the internal buffer for this port is almost full. This is known as crossing the high water mark. If **Ctrl S Ctrl Q** handshaking is enabled, a **Ctrl S** is sent when this bit is set.

Bit seven indicates that a **Ctrl S** has been received and is waiting for a **Ctrl Q**. A problem almost certainly exists if both bits six and seven are set, since this indicates that the other device sent PDOS a **Ctrl S** to signal us to stop sending and then sent us too much data to hold.

```
/*058*/       char _utyp[16];                 /* port uart type  */
```

Although PDOS documentation refers to all types of character ports as UARTs (Universal Asynchronous Receiver/ Transmitters), a PDOS port may be one of a much larger group of devices. On some implementations of PDOS, a "port" is a memory-mapped graphic screen with associated keyboard. On another, the "UART" may actually be implemented with some sort of parallel printer port device. Each one of these different types of hardware requires a different handler subroutine on a given PDOS implementation. PDOS allows up to eight types of character ports, each with its own device service routine. These are named in MBIOS:SR as U$1DSR, U$2DSR, U$3DSR, and U$4DSR. Most PDOS implementations only have 1 or 2 device types. The UTYP table tells what set of device service routines each port should use.

```
/*068*/       char _urat[16];                 /* port rate table  */
```

The transmission speed in bits per second of a port (usually called "baud rate") is set with the BP command from the monitor or the XBCP primitive. The current speed is saved in the UART table (range = 0 to 8).

```
/*078*/       char _evtb[10];         /* 0-79 event table */
/*082*/       char _evto[2];          /* 80-95 output events   */
/*084*/       char _evti[2];          /* 96-111 input events   */
/*086*/       char _evts[2];          /* 112-127 system events */
```

Logical events on PDOS are stored in SYRAM in a table of bits. If the bit is set, the corresponding event is set, and vice versa. The bits for events 0-127 are stored in these tables.

```
/*088*/       char _ev128[16];        /* task 128 events     */
```

Event 128 is a special event, and local to each individual task. This means that there is a unique bit allocated for every task in PDOS. The EV128 table contains these bits. Normally, the local events are set by the local task, or are set in response to a delay event initiated by the XDEV primitive. It is possible, however, for a program to set the local event of another task by directly modifying the corresponding bit in the EV128 field. The code below shows an example:

```
*       SET THE LOCAL EVENT OF TASK N
        OPT     PDOS
START   XGML                    ;MAKE SURE A5 PROPERLY POINTS TO SYRAM
        XPMC    PROMPT
        XGLU                    ;READ IT
        XCDB                    ;CONVERT TO BINARY-->D1

         MOVE.W D1,D0           ;COPY NUMBER
        LSR     #3,D1           ;TASK NUMBER / 8 (BYTE INDEX)
        ADDI.W  #EVTS.+2,D1     ;BIAS BY BEGINNING OF EVENT TABLE
        NOT.B   D0              ;COMPLEMENT TO REVERSE BIT INDEX
        BSET    D0,0(A5,D1.W)   ;SET THE BIT
        XEXT

PROMPT  DC.B    13,10,'Enter task number:',0
        EVEN
        END     START
```

```
/*098*/        long _evtm[4];          /* events 112-115 timers  */
```

Events 112-115 are special because they are automatically set every so many clock tics. Event 112 occurs every 1/5 of a second; event 113 marks the second interval; event 114 happens on the ten second mark and event 115 every 20 seconds. The EVTM array contains the individual counters for these four timers, as they count up the required number of clock tics for the specified interval.

```
/*0A8*/        long _bclk;             /* clock adjust constant   */
```

The standard PDOS clock allows for timing in terms of 100 tics per second, 128 tics per second, or some other fairly small number that is convenient. Sometimes, however, adjusting the number of tics per second still results in a system clock that runs too fast or too slow, due to a crystal that has an odd oscillation period or some other hardware peculiarity. In that case, PDOS allows a small adjustment to be made to the clock every second, when the B$LED routine is called. This code is part of the BIOS routines that the user may customize. A typical set of code appears below:

```
B$LED   MOVE.L  B_CLK(A0),D0    ;ADJUST CLOCK?
  BEQ.S @0002                   ;N
ADD.L   D0,BCLK.(A5)            ;Y, ADJUST COUNT, CARRY?
  BCC.S @0002                   ;N
 ADDQ.W #1,FCNT.(A5)            ;Y, UP COUNTER
```

In this code, the BIOS field B_CLK is added every second to the SYRAM counter BCLK. until the result overflows the 32-bit field. Then, the fine counter is incremented. This allows for a fairly small adjustment to be made to the fine counter to keep the PDOS system clock current.

```
/*0AC*/        char *_tltp;            /* task list pointer     */
```

This points to the entry in the task list that corresponds to the current task.

```
/*0B0*/        char *_utcb;            /* user tcb ptr          */
```

This points to the TCB (task control block) of the current task. The information also exists in the task list entry for this task, but it is copied out here for convenience in access.

```
/*0B4*/      int  _suim;          /* reserved for PDOS use */
```

This field is reserved for PDOS use only.

```
/*0B6*/      int  _usim;          /* reserved for PDOS use */
```

This field is reserved for PDOS use only.

```
/*0B8*/      char _sptn;          /* reserved for PDOS use */
```

This field is reserved for PDOS use only.

```
/*0B9*/      char _utim;          /* user task time       */
```

This field is initialized from the entry in the task list. Every clock tic, it is decremented until it goes to zero, indicating that the task's time slice is up and it needs to swap.

```
/*0BA*/      char _tpry;          /* task priority       */
```

This field is initialized from the entry in the task list. This field is also used in the monitor, where the current task priority is used as the default priority for the "CT" command, and one less than the current priority is used as the task priority for the "@" command.

```
/*0BB*/      char _tskn;          /* current task number */
```

This is the task number of the currently executing task. PDOS uses it heavily. It is also available in the TCB as the TID$ field.

```
/*0BC*/      char spare1;         /* reserved            */
```

This byte is reserved for future use by PDOS.

```
/*0BD*/      char _tqux           /*task queue offset flag  */
```

For fastest interrupt response, the time critical task suspends on either a logical or physical event. the associated interrupt service routine should acknowledge the hardware, set the event bit directly, and then load the task number (0-127) into the TQUX byte of SYRAM. The ISR then exits with an XRTE primitive. The PDOS task scheduler will immediately schedule the designated task.

```
/*0BE*/      char _tlck;          /* task lock flag      */
```

This flag is set by the lock task primitive (XLKT) and unlocked by the unlock task primitive (XULT). It also may be set by using the TAS.B TLCK.(A5) instruction. When it is set, PDOS will not schedule any other task. In other words, scheduling is disabled. Several critical functions in PDOS are protected by locking the task until they are finished.

```
/*0BF*/       char _rflg;              /* task reschedule flag    */
```

This field is set internally whenever scheduling is attempted when the task is locked. It indicates that PDOS should immediately re-schedule when the task unlocks. Normally, this field is set to FF, with 0 indicating re-schedule.

```
/*0C0*/       char _e122;              /* batch task #            */
```

Normally, the PDOS monitor creates a separate task whenever you precede a command line with "@". This task is by default 32K and executes at a priority 1 lower than the current task. If the E122 field is contains the number of a PDOS task, however, (task 0 doesn't count), the monitor will send the command line in a task message to that task and set event 122. The indicated task serves as a batch processor, waiting on event 122, and when it wakes up, getting a task message with XGTM. The batch processor may then execute the monitor command in a "background" type of mode. You may write your own batch processor to take advantage of this feature.

```
/*0C1*/       char _e123;              /* spooler task #          */
```

The monitor command "CF" normally just executes an XCPY (copy files) command from the file named as the first argument to the file named as the second. If the E123 field of SYRAM is non-zero (has a valid task number), the monitor sends the command line to that task via XSTM and sets event 123. A background spooler task may then put its own task ID in the E123 field and suspend on event 123. When it wakes up, the command line is obtained by executing the get message primitive, XGTM. The proper disposition of the command line depends on the spooler task. You may write your own spooler to take advantage of this feature.

```
/*0C2*/       char _e124;              /* reserved for PDOS use */
/*0C3*/       char _e125;
```

These two fields are reserved for future use by PDOS.

```
/*0C4*/       long _cksm;              /* reserved for PDOS use */
```

This field is reserved for future use by PDOS.

```
/*0C8*/       int  _pnod;              /* pnet node #             */
```

The byte at SYRAM offset $0C9 is initialized by PDOS start-up to the node character passed from the BIOS in the upper byte of D7.L This defaults to a nul ($00) which means that it is not a multi-processor system. If this byte is non-zero, the monitor prompt outputs the byte as a character followed by a blank. It can be set at DOSGEN time with /NODE=$xx. See the MBIOS:SR file.

```
/*0CA*/ char _bser[6];                /* reserved for PDOS use  */
/*0D0*/ char _iler[6];                /* reserved for PDOS use  */
```

These two fields are reserved for PDOS use only.

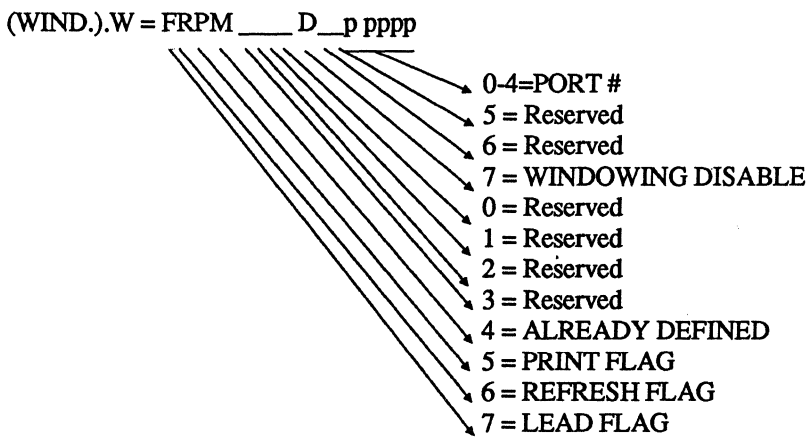
```
/*0D6*/ char _ccnt[16];               /* control C count      */
```

Starting with PDOS version 3.0 there has been a system utility called MABORT. This utility runs in the background at a high priority, but only executes a few instructions before swapping. It reschedules every second, and when it runs, it tests the **Ctrl C** count for every port. If any port has received multiple **Ctrl C** characters in excess of the limit (default limit is 2, but it can be set to any number when the MABORT task is created), the task mapped to that port for input is aborted. The functioning of the MABORT task is described in the *PDOS Monitor, Editor, Utilities* manual. PDOS takes care of incrementing the **Ctrl C** count every time one is received. The CCNT field has one byte available for every port -- PDOS input routines take care of incrementing the count. Some utilities, such as MEDITCON and MEDIT, disable the **Ctrl C** count abort function temporarily by loading the port's byte with $80. This minus count will not abort the task. Of course, if the control character disable bit is set on the port, the **Ctrl C** is ignored and the CCNT remains the same.

```
/*0E6*/       char * wind;            /* window IDs           */
```

Starting with PDOS 3.2, the system utility called WIND1 works with PDOS to give you multiple logical ports to a physical port. The WIND1 program is described in the *PDOS Monitor, Editor, Utilities* manual. When the WIND1 program begins, it allocates a buffer within its own task space and saves a pointer to that buffer in the WIND field of SYRAM. A non-zero value in the WIND field, then serves as an indicator that windowing is enabled. When this is the case, every character input or output through a port undergoes additional processing. When data comes in to or goes out from a port, that port is checked to see if it is mapped to a logical port. If so, the characters are routed appropriately into the logical port or out the physical port. In addition, a memory image of mapped logical ports is kept within a buffer in the WIND1 task. When the user maps his physical port to a different logical port, the WIND1 task refreshes the screen to show the current display of that logical port.

The WIND field points to an array of 15 words (one for each physical port) each with the following format:



The low order 4 bits indicate the logical port that is mapped onto the physical port. The following code segment illustrates (in C) how a program might cause its own screen to be refreshed.

```
refresh(port)
int port;
{
#include <SYRAM:H>
    int *ptr = _syram->_wind;     /* pick up the pointer to window table.*/
    while ((*ptr & 0xf) != port) /* scan the list to find the */
       ptr++;                              /*   current port.  */
    *ptr |= 0x4000;                        /* set refresh bit */
    xsef(127);                             /* Wake WIND1 to refresh */
}
```

```
/*0EA*/       char *_wadr;            /* window addresses    */
```

The WADR pointer in SYRAM points to the buffer in the WIND1 task where the screen images are saved. This pointer is initialized by the WIND1 program and used by PDOS whenever characters are output to a port with windows enabled.

```
/*0EE*/       char *_chin;            /* input stream        */
/*0F2*/       char *_chot;            /* output stream       */
```

A hook in PDOS 3.2 is the character input/output traps. Whenever PDOS receives a character or sends a character to a port, it tests the CHIN or CHOT vector. If the field is non-zero, it makes a subroutine call to the address saved there. The following code shows how data for a particular port might be saved away in a buffer for a monitoring program to test. One use for these traps is a communications analyzer program that watches all data through a modem port and displays it on a PDOS terminal.

To use the following code segment, the CHIN trap and CHOT traps must be initialized with the addresses of the .CHIN and .CHOT routines. The data goes into the .CHINBUF queue, the first three words of which tell the input, output, and size of the queue.

```
SECTION 2
EXTN     .PORT,.CHINBUF,.CHOTBUF,.CHIN,.CHOT

BSIZE   EQU     256                     /* must be power of 2 */
.PORT           DS.L    1               /* declare the PORT variable*/
.CHINBUF        DS.W    3               /* PUT, GET, COUNT */
        DS.B    BSIZE                   /* BUFFER */
.CHOTBUF        DS.W    3
        DS.B    BSIZE
SECTION 0

***********************************
*       INTERCEPT INPUT STREAM
*       D0.B = CHAR
*       D2.B = PORT
 *
.CHIN
        CMP.B   .PORT+3,D1              /* FOR US? */
        BNE.S   @0099
        MOVE.L  A0,-(A7)                /* FREE UP REGISTER */
        MOVEA.L #.CHINBUF,A0            /* POINT TO STRUCTURE */
        BSR.S   PUTBYTE                 /* SAVE A BYTE AWAY */
        MOVEA.L (A7)+,A0                /* RESTORE REGISTER A0 */
@0099   RTS

 ***********************************
*       INTERCEPT OUTPUT STREAM
*       D0.B = CHAR
*       D1.B = PORT
 *
.CHOT
        CMP.B   .PORT+3,D1
         BNE.S   @0099
        MOVE.L  A0,-(A7)                /* FREE UP REGISTER */
        MOVEA.L #.CHOTBUF,A0            /* POINT TO STRUCTURE */
        BSR.S   PUTBYTE                 /* SAVE A BYTE AWAY */
        MOVEA.L (A7)+,A0                /* RESTORE REGISTER A0 */
@0099   RTS

 ***********************************
*       SAVE AWAY A BYTE
*       D0.B = CHAR
*       A0-> 0: PUT INDEX.W
*            2: GET INDEX.W
*            4: COUNT.W
*            6: BUFFER(0..BSIZE-1)
 PUTBYTE
        MOVE.L  D1,-(A7)                /* FREE UP A REGISTER */
        MOVE.W  (A0),D1                 /* PUT INDEX */
        MOVE.B  D0,6(A0,D1.W)           /* STORE THE BYTE */
         ADDQ.B #1,D1                   /* INCREMENT THE POINTER*/
        AND.W    #BSIZE-1,D1            /* WRAP IF NECESSARY */
        MOVE.W   D1,(A0)                /* SAVE IT */
        ADDQ.W  #1,4(A0)                /* INCREMENT COUNT */
        MOVE.L  (A7)+,D1                /* RESTORE REGISTER */
        RTS

        END
```

```
/*0F6*/       char *_iord;          /* I/O redirect        */
```

The IORD table is initially loaded with a pointer to a field later in the SYRAM table called the RDTB. The RDTB table contains one byte for every port on the system. Before data is output to a port, the corresponding byte in the re-direct table is checked. If a non-negative byte is there, the data is placed in the INPUT buffer of the indicated port -- thus output to one port is re-directed to become input data to another port.

```
/*0FA*/       char _fect;     /* file expand count   */
```

This byte is the [number of sectors-1] added to the end of a file when it is expanded. The default is $01, which means that files are extended by two sectors. This value is also used as the initial number of sectors for defining non-contiguous files. It can be altered at DOSGEN by setting B.FEC to the value [count-1] when assembling xxBIOS:SR.

```
/*0FB*/       char _pidn;     /*processor ident byte*/
```

PDOS 3.3a and newer sets this byte to a value so programs can tell which 68000 family processor is currently running. The values possible for this byte are as follows:

=0	not PDOS 3.3a or newer
=1	68000 or 68008
=2	68010 or 68012
=3	68020 without 68881
=4	68020 with 68881

This byte alters the message output by the ID monitor command.

```
/*0FC*/       char *_begn;    /* abs address of kernel entries */
```

This long word points to the absolute entry table of the PDOS kernel or the label K1$BEGN. The dispatch table contains various entry points into PDOS which are described in the section following.

```
/*100*/       int _rwcl[13];          /* port row/col 1..15     */
```

Whenever PDOS performs a position cursor (XPSC) or moves the cursor by printing a character, a line-feed, a carriage return, or a backspace, it records the current position of the cursor in the RWCL table. One entry is saved for each logical port. When WIND1 refreshes the screen, it places the cursor where the RWCL entry for the port indicates. The XRCP read cursor position primitive refers to this table.

```
/*11C*/       char *_opip[15];    /* reserved for PDOS use */
```

This field is reserved for PDOS use only.

```
/*158*/       char *_uart[17];   /* UART base addresses 1..15   */
```

On a 68000 system, all I/O is handled through memory-mapped addresses. This table tells the base address for every port on the system. It contains the fourth parameter for the "BP" command.

```
/*198*/       long _mapb;              /* memory map bias     */
```

This pointer indicates the start of the memory map used by PDOS to allocate and deallocate task memory.

Variable Offset

There are more tables and buffers in MSYRAM, but their location and size change from one installation to another, according to DOSGEN parameters. They are noted in the MSYRAM:SR file, but user programs do not generally access them. If an application needs to access one of these tables, the offsets and sizes are available in the B$BIOS table at fixed offsets. The address of this table is the first location in SYRAM and the current offsets are as follows:

MSYRAM Switches

The system RAM is defined by the source module MSYRAM:SR. This section of random access memory contains tasking, file, message, memory, timing, and scheduling information.

SYRAM configuration:

- NT {Task table size}
- NM {# Task messages}
- TZ {Task message size}
- ND {# Delayed events}
- NC {# Channel buffers}
- NF {# File slots}
- NU {# Input buffers}
- IZ {Input buffer size}
- MSZ{Maximum memory size}

The SYRAM configuration section describes the PDOS system variable definitions. Various queues and stacks are defined such as the maximum number of tasks, buffers, slots, delay lists, and bit map. All parameters have default values.

NT

TASK TABLE SIZE. The size of the task table in SYRAM determines how many tasks PDOS supports. Default NT = 32.

NM

TASK MESSAGES. The task messge buffers are queued, intertask communication buffers whose size can vary from 4 to 127 bytes. Default NM=32.

TZ

TASK MESSAGE SIZE. The task message size can vary from 4 to 127 bytes. Default TZ=64.

ND

DELAYED EVENTS. The maximum number of events being delayed is set by this value. Default ND=32.

NC

CHANNEL BUFFERS. Disk sector access is cached through the channel buffers. If more files are opened than there are available channel buffers, then the least used buffers are written out to disk until accessed again. Default NC=8.

NF

FILE SLOTS. The NF parameter specifies the maximum number of files that can be opened concurrently . Default NF=32.

NU

INPUT BUFFERS. The number of ports specifies how many type-ahead buffers to allocate. Default NU=15.

IZ

INPUT BUFFER SIZE. The IZ variable determines the input buffer size (how many characters can be stored in the type-ahead buffers). IZ ranges from 1 to 7 and is used as the power of 2 of the buffer size. Thus, the buffer size ranges from 2 to 128 as IZ goes from 1 to 7. Default IZ=6.

IZ = 1 ---> 2	IZ = 5 ---> 32
IZ = 2 ---> 4	IZ = 6 ---> 64
IZ = 3 ---> 8	IZ = 7 ---> 128 (maximum>
IZ = 4 ---> 16	

MZ

MAXIMUM MEMORY SIZE. The memory bit map specifies how much memory is available to PDOS. Each bit represents 2k bytes of memory. If bit map size is less than 2048, then it is the number of k bytes. Otherwise, it is the actual number of bytes known to PDOS.

Dispatch Table

A location-independent dispatch table is the first location of the kernel as is referenced by label K1$BEGN. From application programs, K1$BEGN may be referenced through the SYRAM offset BEGN.(A5). This table provides an easier interface for cross development routines. The entry points to this table are defined below:

```
*****************************************************
*     DISPATCH TABLE FOR CROSS DEVELOPMENT
*
              DC.L    $FFFFFFFF       ;CROSS DEV SP
              DC.L    $FFFFFFFF       ;CROSS DEV PC
              DC.L    '3.3a'          ;REVISION IDNT
*
K1$BEGN       BRA.L   K1$STRT         ;$00 = KERNEL COLD START
              BRA.L   K1$CLKI         ;$04 = CLOCK INTERRUPT
              BRA.L   K1$XSWP         ;$08 = TASK SWAP
              BRA.L   K1$SERR         ;$0C = SYSTEM ERROR ENTRY
              BRA.L   K2$PINT         ;$10 = PORT LOOKER
              BRA.L   K2$CHRI         ;$14 = PORT CHAR IN ENTRY
              BRA.L   K2$CHAR         ;$18 = INSERT CHAR TO BUFFER
              BRA.L   K2$AOEV         ;$1C = ACK OUTPUT EVENT
              BRA.L   K1$SVEC         ;$20 = SET EXCPT VECTOR
              BRA.L   D$INT           ;$24 = DEBUGGER INIT
              BRA.L   K1$ISWP         ;$28 = INTERRUPT SWAPPER
              BRA.L   M$AERM          ;$2C = XLATE ERR MSG, NO OUT
```

The long word preceding the K1$BEGN table is the kernel revision number, which could be used by applications to determine with which version of PDOS it is running. The next two preceding long words of $FFFFFFFF are to allow the PDOS object to be located at the beginning of the ROMs on the target system. To make calls to the routines or subroutines, the program should first save any working registers and initialize any required inputs. Then get a pointer to SYRAM, in A5 for example, and get the address of the K1$BEGN table from SYRAM into another address register, such as A4. Then go to the routine with either a JMP or JSR instruction to an offset of A4, as below:

```
      MOVEA.L B$SRAM,A5
      MOVEA.L BEGN.(A5),A4
      JSR     $18(A4)          ;insert character into buffer
```

A detailed description follows:

K1$STRT

Kernel Cold Start
Entry point

```
JMP $00(A4)
```

Inputs D4.L = B.BAS Memory bit map base address
D5.W = Overriding BAUD rate (-1 = use R$TASK table value)
D6.L = B.VEC Exception vector base address
D7.L = Node.B / Auto.B / $00.B / SDK$.B
(A3) = BINTB Vector table
(A4) = B$BIOS table
(A6) = Start of tasking memory
(A7) = End of tasking memory
First SYRAM locations must also be set
Interrupts must be off
Outputs <None>
Called Once from MBIOS cold start up.

K1$CLKI

Clock Interrupt
Entry Point

```
JMP $04(A4)
```

Inputs (A7) >> Status register word, old PC (& exception word)
Supervisor mode, just as if the timer has interrupted
Outputs <None>
Called Directly send system timer interrupt to this entry with an entry in BINTB table of MBIOS, B$ACK will acknowledge the interrupt.

K1$XSWP

Task Swap
Entry point

```
JMP $08(A4)
```

Inputs (A7) >> D0-A6, SR.W, PC.L
(A5) = SYRAM
TQUX.(A5) = # of task to wake up (optional)
Supervisor mode
If called from interrupt routine, need to re-enable interrupt mask to pre-int level
Outputs <None>
Called When XSWP primitive is executed or as exit from user interrupt routine for fast task wake up.

K1$SERR

User System Error Entry
Entry point

```
JMP $0C(A4)
```

Inputs (A7) = DC.L (MESSAGE
DC.W LADR,R/W,I/N,CODE
DC.L ACCESS ADDRESS
DC.W INSTRUCTION REGISTER
DC.W STATUS REGISTER
DC.L PROGRAM COUNTER
Supervisor mode
Outputs <None>
Called By user added error trap, acts just like PDOS exception.

K2$PINT

All-Port Looker
Entry point

```
JMP $10(A4)
```

Inputs (A7) >> Status register word, old PC (& exception word)
Supervisor mode, just as if the UART has interrupted
Ouputs <None>
Called Directly send UART receiver interrupt to this entry with an entry in BINTB table of MBIOS; causes a general port look on all UART types, calls get character for each type so BIOSU code will acknowledge interrupt and get character.

K2$CHRI

External Port Character Input
Entry point

```
JMP $14(A4)
```

Inputs (A7) >> D0-A6,SR.W, PC.L
D0.B = CHARACTER
(A0) = UART BASE ADDRESS
(A5) = B$SRAM
Outputs <None>
Called From K2$PINT general looker after character is found or directly from UART receiver interrupt using BINTB entry.

K2$CHAR

Insert Character to Buffer
Subroutine

```
JSR $18(A4)
```

Inputs D0.B = CHARACTER
(A0) = UART BASE ADDRESS
(A5) = SYRAM
Outputs D2.W = LOGICAL PORT #
(A0) = PHYSICAL UART BASE ADDRESS
(A1) = PHYSICAL FLAGS ADDRESS
(A2) = PHYSICAL DSR ADDRESS
.EQ. = HIGH WATER ((A1)=D1.W=FLAGS)
.VS. = Type-ahead buffer OVERFLOW
Called By K2$CHRI to store chracter into buffer; may be used by keyboard interrupts to load multiple characters into the buffer with just one keystroke (e.g. function keys).

K2$AOEV

Ack Output Event
Subroutine

```
JSR $1C(A4)
```

Inputs (A0) = UART BASE ADDRESS
Outputs <None>
Called From user printer interrupt to set output event associated with the port, regardless of BAUD PORT binding.

K1$SVEC

Set/Read Exception Vector
Subroutine

```
JSR $20(A4)
```

Inputs (A5) = SYRAM
F681. (A5) = 0/2 for processor type
D0.B = VECTOR # (2-255)
(A0) = NEW ROUTINE OR $0000 (for read only)
Outputs (A0) = OLD ROUTINE address
Called During K2$STRT kernel cold start and by XVEC primitive. May be called by BIOS for processor independent BIOS implementations or used in applications as substitute for XVEC primitive.

D$INT

Debugger Initialize
Subroutine

```
JSR $24(A4)
```

Inputs <None>
Outputs Alters debug area of TCB
Called By XCTB when new tasks are created, or whenever task size is altered with FM or GM monitor commands. The INSTALL utility also uses this subroutine.

K1$ISWP

Interrupt Swapper Entry Point

```
JMP $28(A4)
```

Inputs (A6) >> A5.L
SR.W
PC.L
Supervisor mode
TQUX.(A5) = # or task to wake up (optional)

Outputs <None>

Called As an exit from user interrupt service routines for a fast "swap to task." A5 is assumed to be pushed on the stack and the interrupt device has been acknowledged. This routine pops A5, pushes all registers in the stack, re-enables interrupts and enters K1$XSWP, just like XRTE, but without double register pushing/rolling.

M$AERM

Translate Error Message, No Out Subroutine

```
JSR $2C(A4)
```

Inputs D1.W = Error number
(A1) = Buffer to receive error message

Outputs (A1) = Error message

Called By ER monitor command or by user utility to get translation of PDOS error message for the system on which it is running.

System Services

System services are those functions that a task requires of the operating system while entered in the task list. These requirements range from timing and interrupt handling to task coordination and resource allocation.

PDOS provides many time-oriented functions which key off of the system hardware interval timer. The current time of day and date are maintained with fine adjustment parameters. A 32-bit counter is used for various delta time functions such as task scheduling and event delays.

Hardware interrupts are processed by the kernel BIOS or passed to user tasks. Tasks can be suspended pending the occurrence of an interrupt and then be rescheduled when the interrupt occurs. Interrupts such as the interval timer and character input or output are handled by the kernel itself.

Task coordination is an integral part of realtime applications since many functions are too large or complex for any single task. The PDOS kernel uses common or shared data areas, called mailboxes, along with a table of pre-assigned bit variables, called events, to synchronize tasks. A task can place a message in a mailbox and suspend itself on an event waiting for a reply. The destination task is signaled by the event, looks in the mailbox, responds through the mailbox, and resets the event signaling the reply.

System resources include the processor itself, system memory, and support peripherals. The PDOS kernel provides assembly primitives to create and delete tasks from the task list. Memory is allocated and deallocated as required. Peripherals are generally a function of the file manager but are assigned and released via system events. Device drivers coordinate related I/O functions, interrupts, and error conditions. All of these functions are available to user tasks and thus tasks may spawn tasks and dynamically control their operating environment.

Support Utilities

Other support utilities contained within the PDOS kernel include number conversion, command line decoding, date and time conversions, and message processing routines. Facilities are also provided for locking a task in the run state during critical code execution.

PDOS Character I/O

The flow of character data through PDOS is the most visible function of the operating system. Character buffering or type-ahead assures the user that each keyboard entry is logged, even when the application is not looking for characters. Character output is normally performed through program control (polled I/O).

Input and output occurs through logical port numbers. A logical port is bound to a physical UART (Universal Asynchronous Receiver /Transmitter) by the baud port commands. Only one task is assigned to an input port at any one time while many tasks may share the same output port. It is then the responsibility of each task to coordinate all outputs.

PDOS Character Input

PDOS character inputs come from four sources: 1) user memory; 2) a PDOS file; 3) a polled I/O driver; or 4) a system input port buffer. The source is dictated by input variables within the task control block. Input variables are the Input Message Pointer (IMP$(A6)), Assigned Console Input (ACI$(A6)), and input port number (PRT$(A6)).

```
          OPT     PDOS          ;GET TCB VARIABLES
          LEA.L   CMMD(PC),A1   ;POINT TO COMMAND
          MOVE.L  A1,IMP$(A6)   ;SET INPUT POINTER
          ....
CMMD      DC.B    'MESSAGE',0
          EVEN
```

When a request is made by a task for a character and IMP$(A6) is nonzero, then a character is retrieved from the memory location pointed to by IMP$(A6). IMP$(A6) is incremented after each character. This continues until a null byte is encountered, at which time IMP$(A6) is set to zero.

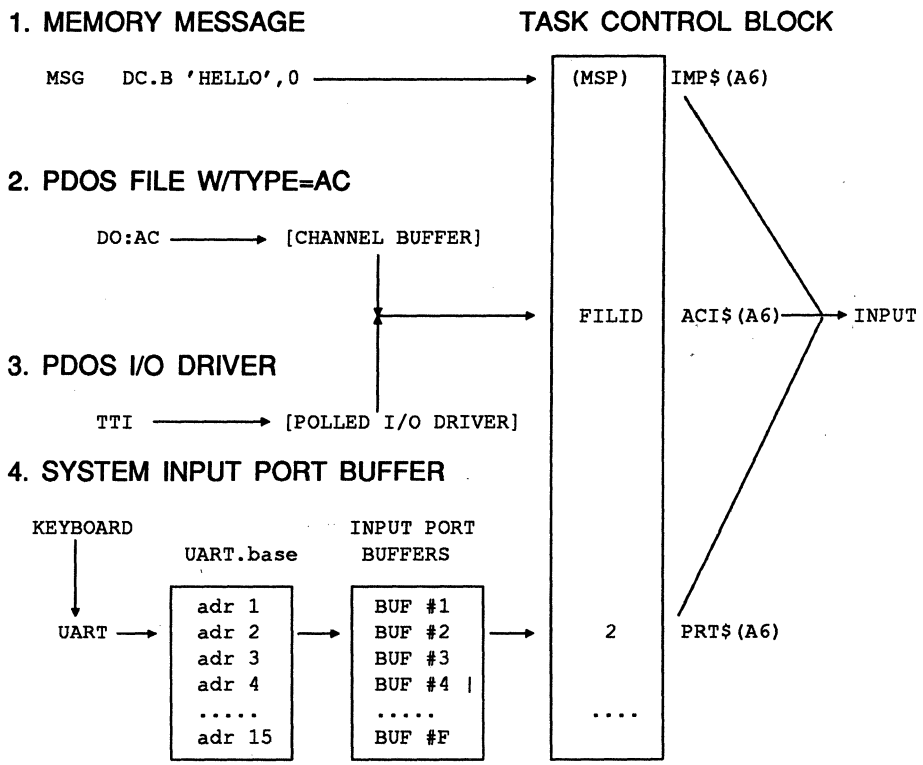
```
          OPT     PDOS          ;GET TCB VARIABLES
          LEA.L   FILEN(PC),A1  ;POINT TO FILE NAME
          XSOP
            BNE.S ERROR
          MOVE.W  D1,ACI$(A6)   ;SET CONSOLE INPUTS
          ....
FILEN     DC.B    'INDATA',0
          EVEN
```

If IMP$(A6) is zero and ACI$(A6) is nonzero, then a request is made to the file manager to read one character from the file assigned to ACI$(A6). The character then comes from a disk file or an I/O device driver. This continues until an error occurs (such as an end-of-file) at which time the file is closed and ACI$(A6) is cleared.

```
          OPT     PDOS          ;GET TCB VARIABLES
          MOVEQ.L #3,D1         ;READ CHARACTERS FROM
          MOVE.B  D1,PRT$(A6)   ;  PORT #3
          ....
```

If both IMP$(A6) and ACI$(A6) are zero, then the logical input port buffer selected by PRT$(A6), is checked for a character. If the buffer is empty, then the task is automatically suspended until a character interrupt occurs.

PDOS character input flow is summarized below:



UART.base binds a physical UART to a logical port number.

UART baud rate, address, and type are defined by the "BP" and "baud" commands (XBCP primitive).

XGCC gets characters from input port buffers only.

PDOS Character Output

PDOS character outputs are directed to various destinations according to output variables in the task control block. Output variables are the output unit (UNT$(A6)), spooling unit (SPU$(A6)), spooling file ID (SFI$(A6)), and output port variables U1P$(A6), U2P$(A6), U4P$(A6), and U8P$(A6). The output unit selects the different destinations. (Do not confuse output units with disk unit numbers).

```
          OPT     PDOS          ;GET TCB VARIABLES
          LEA.L   FILEN(PC),A1  ;GET FILE NAME
          XSOP                  ;OPEN FILE
            BNE.S ERROR
          MOVE.W  D1,SFI$(A6)   ;SET SPOOL FILE ID
          MOVEQ.L #0,D1         ;CLEAR COUNTER
          MOVE.B  #4,SPU$(A6)   ;SET SPOOL UNIT TO 4
*
LOOP      MOVE.B  D1,UNT$(A6)   ;SELECT UNIT
          XCBM    MES01         ;CONVERT NUMBER
          XPLC                  ;OUTPUT MESSAGE
          ADDQ.W  #1,D1         ;INCREMENT D1
          CMPI.W  #8,D1         ;8 TIMES?
            BLT.S LOOP          ;N
          ....                  ;Y

FILEN     DC.B    'OFILE',0     ;OUTPUT FILE NAME
MES01     DC.B    'OUTPUT MESSAGE #'.0
          EVEN

UNIT 1 =          OUTPUT MESSAGE #1
                  OUTPUT MESSAGE #3
                  OUTPUT MESSAGE #5
                  OUTPUT MESSAGE #7

UNIT 2 =          OUTPUT MESSAGE #2
                  OUTPUT MESSAGE #3
                  OUTPUT MESSAGE #6
                  OUTPUT MESSAGE #7

OFILE =           OUTPUT MESSAGE #4
                  OUTPUT MESSAGE #5
                  OUTPUT MESSAGE #6
                  OUTPUT MESSAGE #7
```

When an output primitive is called, the task output unit is ANDed with the task spooling output unit. If the result is nonzero, then the character is directed to the file manager and written to the file specified by SFI$(A6). The output unit is then masked with the complement of the spooling unit and passed to the UART character output processor.

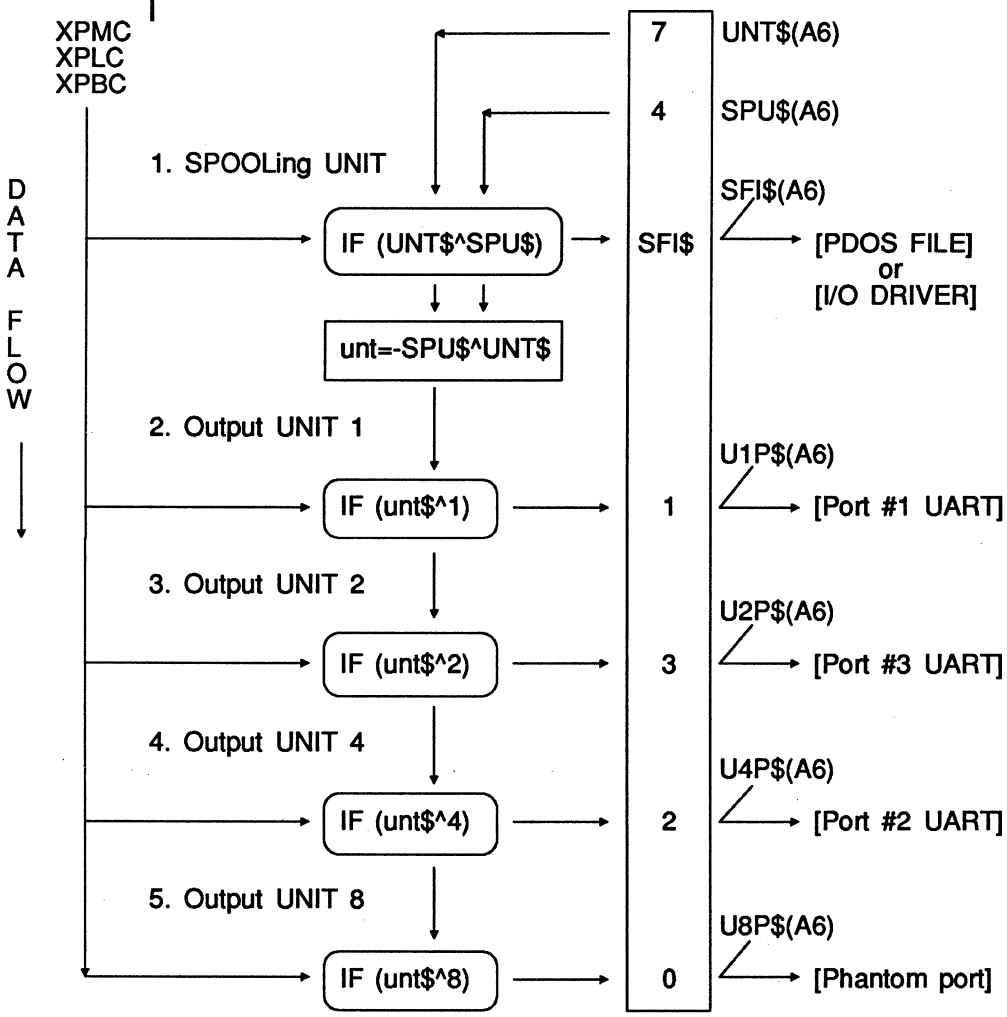
Units 1, 2, 4, and 8 are special output numbers. Unit 1 is the console output port assigned when the task was created. Units 2, 4, and 8 are an optional output ports that correspond to TCB variables U2P$, U4P$, and U8P$. They are assigned by the spool unit command (SU) or baud port (BP) command.

If the 1 bit (LSB) is set in the masked output unit (UNT$(A6)), then the character is directed to port U1P$(A6). Likewise, if bits 2, 3, or 4 are is set in the masked output unit, then the character is output to the U2P$(A6), U4P$(A6), or U8P$(A6) ports.

In summary, the bit positions of the output unit are used to direct output to various destinations. More than one destination can be specified. Bits 1 through 4 are predefined according to U1P$, U2P$, U4P$ and U2P$ variables within the task control block. Other unit bits are used for outputs to files and device drivers. Thus, if SPU$(A6)=4 and UNT$(A6)=7, then output would be directed to the file manager via SFI$(A6) and to two UARTs as specified in U1P$(A6) and U2P$(A6).

SPU$(A6) = 0000 0000 0000 0100
UNT$(A6) = 0000 0000 0000 0111

File SFI$(A6)
Port U2P$(A6)
Port U1P$(A6)



UNIT 1 = (-SPU$ ^ UNT$) ^ 1
UNIT 2 = (-SPU$ ^ UNT$) ^ 2
UNIT 4 = (-SPU$ ^ UNT$) ^ 4
UNIT 8 = (-SPU$ ^ UNT$) ^ 8
PDOS FILE = (SPU$ ^ UNT$)

Events

Tasks synchronize with each other through events which are single bit flags that may be set or cleared. Events are classified as either logical or physical. Physical events refer to a special byte and bit in memory. Logical events are translated to physical events with the XTLP primitive.

Logical events are not address dependent and referred to by 1-128. There are four types of logical events in PDOS: software, software resetting, system, and local. System events are further divided into output, input, timing, driver, and system resource events. System events are predefined software resetting events that are set during PDOS initialization. Event 128 is local to each task and is used as a delay event.

1-63

Logical events 1 through 63 are software events. They are set and reset by tasks and not changed by PDOS task scheduling. A task can suspend itself pending a software event and then be rescheduled when the event is set. One task must take the responsibility of resetting the event for the sequence to occur again.

64-80

Software resetting events. Logical events 64 through 80 are like the normal software events except that PDOS resets the event whenever a task suspended on that event is rescheduled. Thus, one and only one task is rescheduled when the event occurs.

These events are set and reset by the Send Message Pointer (XSMP) and Get Message Pointer (XGMP) primitives.

81-95

Logical events 81 through 95 correspond to output ports 1 through 15. A task suspends itself on an output event after transmitting a character through a UART. When the transmit character complete interrupt occurs, the event is set and the corresponding suspended task continues execution.

Output port events are only supported though the xxBIOSU routines. See your *Installation and Systems Management* guide for implementation details.

96-111

Logical events 96 through 111 correspond to input ports 0 through 15. A task suspends itself on an input event if a request is made for a character and the buffer is empty. Whenever a character comes into an interrupt driven input port buffer, the corresponding event is set.

112-115

Logical events 112 through 115 are timing events and are set automatically by the PDOS clock module according to intervals defined in the PDOS Basic I/O module (BIOS). Event 112 is measured in tics, while events 113, 114, and 115 are in seconds. The maximum time interval for event 112 is 497 days. Events 113, 114, and 115 have a maximum interval of 4,294,967,300 seconds or approximately 136 years. A task suspended on one of these events is regularly scheduled on a tic or second boundary.

112 = 1/5 second event
113 = 1 second event
114 = 10 second event
115 = 20 second event

116-127

Logical events 116 through 127 are for system resource allocation. Drivers and other utilities requiring ownership of a system resource synchronize on these events. These events are initially set by PDOS, indicating the resource is available. One and only one task at a time is allowed access to the resource. When the task is finished with the resource, it must reset the event thus allowing other tasks to gain access.

116 = Reserved
117 = Reserved
118 = Reserved
119 = Reserved
120 = Level 2 lock
121 = Level 3 lock
122 = Batch event
123 = Spooler event
124 = Abort task event
125 = Reserved
126 = Reserved
127 = Virtual ports (windows)

128

Logical event 128 is local to each task. Unlike other events, it can only be set by a delay primitive (XDEV). It is automatically reset by the scheduling of a task suspended on event 128.

Task Communication

Many different methods are available for intertask communication in PDOS. Most involve a mailbox technique where semaphores are used to control message traffic. Specially designed memory areas such as MAIL, COM, and event flags allow high level program communications. PDOS currently maintains 32 message buffers for queued message communications between tasks or console terminals. More sophisticated methods require program arbitrators and message buffers.

Logical event flags are system memory bits, common to all tasks. They are used in connection with task suspension or other mailbox functions. Events 1 through 63 are for software communication flags. Events 64 through 127 automatically reset when a suspended task is rescheduled. Events 81 through 95 are output events; 96 through 111 are input events; 112 through 115 are timing events; and 116 through 127 are system events. Event 128 is local to each task and cannot be used to communicate between tasks.

```
EVENT 30

IF EVF[30]
```

Physical event flags are user-defined, arbitrary memory bits, which may be common among multiple processors. These may also be DONE bits in the status register of a disk controller. Use the primitive XDPE to delay the event, XSOE to suspend on the event, and assembly instructions to test, set or clear physical events.

PDOS maintains 32 64-byte message buffers for intertask communication. A message consists of up to 64 bytes plus a destination task number. More than one message may be sent to any task. The messages are retrieved and displayed on the console terminal whenever the destination task issues a PDOS prompt or by executing a Get Task Message primitive (XGTM). The displayed message indicates the source task number. The BASIC verbs SENDM and GETM may also be used to pass data between tasks.

PDOS supports shorter message pointer transfers between tasks with the Send Message Pointer (XSMP) and Get Message Pointer (XGMP) primitives. When a pointer is sent, event [destination message slot # + 64] is set. When a message pointer is retrieved, the corresponding event is cleared. These messages are not queued, but are much faster for intertask message passing than the queued 64-byte messages.

The FM monitor command is used to permanently allocate system memory for non-tasking data or program storage. Memory allocated in this way can be used for mailbox buffers as well as handshaking semaphores or assembly programs. (See the *PDOS Monitor, Editor, Utilities* manual and the FM monitor command.)

Task Suspension

Any task can be suspended pending one or two events. Logical events (1-127) are system memory bits common to all tasks. Event 128 is local to each task. A suspended task does not receive any CPU cycles until one of the desired events occurs. A task is suspended from BASIC by using the WAIT command, or from an assembly language, C, FORTRAN, or Pascal program by the XSUI primitive. A suspended task is indicated in the List Task (LT) command by the event number(s) being listed under the "Event" heading.

```
x>LT
Task   Prt Tm  Event   Map Size      PC    ...
*0/0   64  2            0   384  00001D08 ...
 1/0   64  2   99       0   20   00001B42 ...
x>
```

When one of the events occurs, the task is rescheduled and resumes execution. If the event is set by the XSEF primitive, then an immediate context switch occurs. If a high priority task is waiting for the event, it is immediately rescheduled, overriding any current task (unless locked). If the event is set with a XSEV primitive, then the task begins execution during the normal swapping function of PDOS.

In the case of physical events, XSOE is used to suspend the task. Tasks suspended on physical events will have the status 1/-1 under the event header when the LT command is typed at the monitor.

High Priority Tasks

A high priority task is defined as a task in the execution list which is exempt from round robin scheduling. This means the task will continue to execute until it suspends itself (due to I/O or if an XSUI command is executed), or a higher priority task becomes ready. Task priority is listed by the LT (List Task) command under the "PRT" heading. A task priority can be altered with the "TP" command.

High priority tasks are useful in writing user interrupt handlers where immediate and fast response is required.

PDOS Exception Handling

When an exception occurs, the processor saves the exception registers, PC and SR, in the debug area of the TCB so that the debugger can be used after an exception. The task last error number LEN(A6) is loaded with 87. The exception processor also exits with an XERR on error 87.

The level 2 and 3 file manager locks are cleared when a task encounters an exception that returns the task back to the PDOS monitor.

The following describes how error exception processing can be interpreted:

Example

```
BUS ERR   @00005C50 54492004 08000000 0165
D0: 0000000D 08000000 00000000 00000000   00000000 00000000 00000000 00000002
A0: 00005C8C 00007381 0000754C 08000000   00004438 0000B000 0034D800 004477FE
```

Definition

```
ERROR_TYPE   @PROG_CNTR   INST_REG/STAT_REG   ACCESS_ADR   SPEC_STAT
DATA REGISTERS D0-D7
ADDRESS REGISTERS A0-A7
```

Summary

ERROR_TYPE:

This message indicates the type of exception that occurred. The types of exceptions that may occur are summarized later.

PROG_CNTR:

This is the value of the Program Counter at the time of the exception. For all exceptions other than bus and address errors, this value represents the address of the next instructions. For bus and address errors, this value does not necessarily point to the instruction that was executing when the error occurred, but may be advanced up to five words, due to the instruction prefetch mechanism.

INST_REG/STAT_REG:

The high order word is the Instruction Register. It is valid only for bus and address errors and will be zero for any other type of exception. This is an internal register that contains the current instruction being processed. Due to the prefetch mechanism, this is not the instruction that caused the error, but from one up to five words away.

The low order word is the Status Register. This represents the current value of the status register.

ACCESS_ADR:

This is the Access Address. It is valid only for bus and address errors and will be zero for any other type of exception. It represents the address that the processor was attempting to access when the error occurred.

SPEC_STAT:

This is the Special Status word. For bus and address error exceptions, it indicates the current processor cycle (i.e. read/write, CPU function codes, instruction/note). For other exception types, the SPEC_STAT field shows the exception frame type and vector offset that is pushed to the stack (for 68010 and 68020 CPUs only).

Error Types

BUS ERR:

A Bus Error occurs when the BERR line on the VMEbus is asserted by a card or the on-board bus timer expires (indicating this memory location does not exist).

ADR ERR:

An Address Error occurs when an attempt is made to access a word or long word at an odd address (on memory that does not support unaligned transfers).

ILLG:

An Illegal Instruction occurs when an unknown instruction is encountered.

ZDIV:

A Zero Divide occurs when the divide instruction encounters a divisor of zero.

CHCK:

A Check occurs in conjunction with the CHK (check register against bounds) instruction.

OVFL:

An Overflow occurs with either a TRAPV (trap on overflow), a TRAPcc (trap on condition), or a cpTRAPcc (trap on coprocessor condition).

PRIV:

A Privilege Violation occurs when an attempt is made to execute a privileged instruction in user mode.

TRCE:

A Trace exception occurs when a trace vector has not been defined in the TCB (task control block) and trace mode is entered by the user program. This does not include the use of trace in the debugger.

FLIN:

A Line-F Emulator exception occurs when an opcode beginning with "F" is executed. PDOS does not use any Line-F codes. All PDOS primitives are implemented with Line-A codes.

SPUR:

A Spurious exception occurs with either a bad interrupt cycle or a reserved interrupt vector (i.e. vectors 12 and 15).

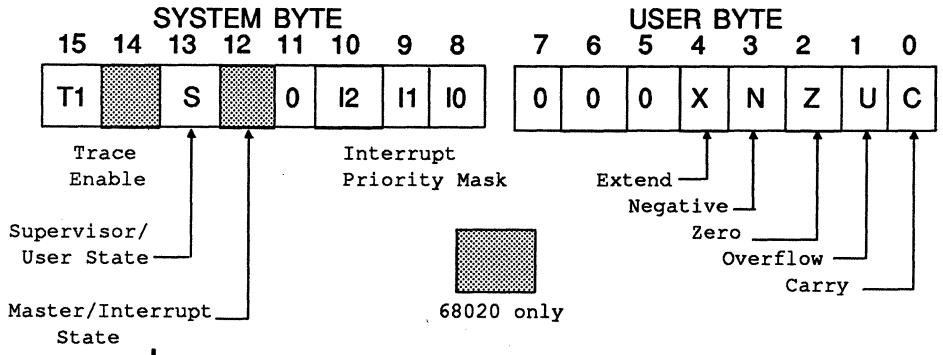
cpER:

A Coprocessor Protocol Violation occurs when an invalid coprocessor response is generated.

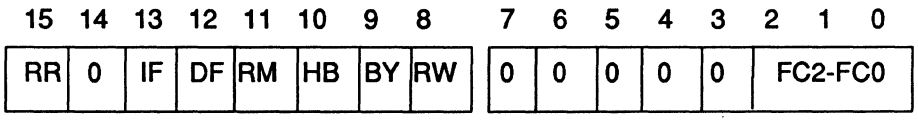
FRMT:

A Format Error occurs in conjunction with an invalid cpRESTORE instruction or a bad exception stack frame.

STATUS REGISTER



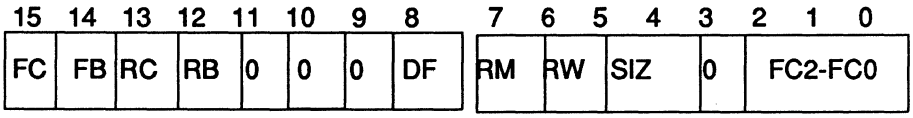
SPECIAL STATUS WORD FOR THE 68010



RR = Rerun flag (0 = rerun instruction)
IF = Instruction fetch cycle
DF = Data fetch cycle
RM = Read-modify-write cycle
HB = High byte transfer
BY = Byte transfer (0 = word transfer)
RW = Read/write cycle (0 = write)
FCx= Function code during faulted access

FCI = User data
FC2 = User program
FC5 = Supervisor data
FC6 = Supervisor program
FC7 = Interrupt acknowledge
All other FCs are undefined

SPECIAL STATUS WORD FOR THE 68020



FC = Fault on Stage C of instruction pipe
FB = Fault on Stage B of instruction pipe
RC = Rerun flag for stage C *
RB = Rerun flag for stage B *
DF = Fault/rerun flag for data cycle *
RM = Read-modify-write data cycle
RW = Read/write data cycle (0 = write)
SIZ= Size code for data cycle
FCx=Function code during faulted access
* 0 = do not rerun bus cycle

FC1 = User data
FC2 = User program
FC5 = Supervisor data
FC6 = Supervisor program
FC7 = CPU space
All other FCs are undefined

File Management

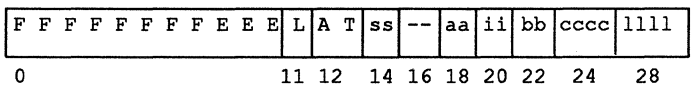
The PDOS file management module supports sequential, random, read only, and shared access to named files on a secondary storage device. These low overhead file primitives use a linked, random access file structure and a logical sector bit map for allocation of secondary storage. No file compaction is ever required. Files are time stamped with date of creation and last update. Default PDOS configurations allow up to 32 files to be open simultaneously; however, PDOS may be configured for up to 127 files. Complete device independence is achieved through read and write logical sector primitives.

File Storage

A file is a named string of characters on a secondary storage device. A group of file names is associated together in a file directory. File directories are referenced by a disk number. This number is logically associated with a physical secondary storage device by the read/write sector primitives. All data transfers to and from a disk number are blocked into 256-byte records called sectors.

A file directory entry contains the file name, directory level, the number of sectors allocated, the number of bytes used, a start sector number, and dates of creation and last update.

(each char represents a byte)




```
F = File Name                    8 characters
E = File Extension               3 characters
L = Directory Level              0-255
A = File Attribute               $80 = AC - Procedure file
                                 $40 = BN - Binary file
                                 $20 = OB - 68000 object module
                                 $10 = SY - System module
                                 $08 = BX - BASIC Token file
                                 $04 = EX - BASIC ASCII File
                                 $02 = TX - ASCII text file
                                 $01 = DR - Driver
T = File Type                    $80 = +  - File altered
                                 $04 = /C - Contiguous file
                                 $02 = /* - Delete protect
                                 $01 = /** - Write protect
s = Start Sector Number          Logical start sector
a = Sectors Allocated to File    Sectors allocated
i = Sector Index of EOF          Sectors used
b = Bytes in EOF Sector          0-252
c = Date/Time Created            hr*256+sc, (yr*16+mn)*32+dy
l = Date File Last Changed          "              "
```

A file is opened for sequential, random, shared random, or read only access. A file type of "DR" designates the file to be a system I/O driver. A driver consists of up to 252 bytes of position independent binary code. It is loaded into the channel buffer whenever opened. The buffer then becomes an assembly program that is executed when referenced by I/O calls.

A sector bit map is maintained for each disk number. Associated with each sector on the logical disk is a bit which indicates if the sector is allocated or free. Using this bit map, the file manager allocates (sets to 1) and deallocates (sets to 0) sectors when creating, expanding, and deleting files. Bad sectors are permanently allocated. When a file is first defined, two sectors are initially allocated to that file and hence, the minimum file size is two sectors.

A PDOS file is accessed through an I/O channel called a file slot. Each file slot consists of a 38-byte status area and an associated 256-byte sector buffer. Data movement is always to and from the sector buffer according to a file pointer maintained in the status area. Any reference to data outside the sector buffer requires the buffer to be written to the disk (if it was altered) and the new sector to be read into the buffer. The file manager maintains current file information in the file slot status area such as the file pointer, current sector in memory, end-of-file sector number, buffer in memory flag, and other critical disk parameters required for program-file interaction.

PDOS defaults to 32 files that may be open at a time though it may be configured to allow for up to 127. Keeping all sector buffers resident would require prohibitive amounts of system memory. Therefore, only eight sector buffers are actually memory resident at a time. The file manager allocates these buffers to the most recently accessed file slots. Every time a file slot accesses data within its sector buffer, PDOS checks to see if the sector is currently in memory. If it is, the file slot number is rolled to the top of the most recently accessed queue. If the buffer has been previously rolled out to disk, then the most recently accessed queue is rolled down and the new file slot number is placed on top. The file slot number rolled out the bottom references the fourth last accessed buffer which is then written out to the disk. The resulting free buffer is then allocated to the calling file slot and the former data restored.

Files requiring frequent access generally have faster access times than those files which are seldom accessed. However, all file slots have regular access to buffer data.

PDOS allocates disk storage to files in sector increments. All sectors are both forward and backward linked. This facilitates the allocation and deallocation of sectors as well as random or sequential movement through the file.

PDOS files are accessed in either sequential or random access mode. Essentially, the only difference between the two modes is how the end-of-file pointers are handled when the file is closed. If a file has been altered, sequential mode updates the EOF pointer in the disk file directory according to the current file byte pointer, whereas the random mode only updates the EOF pointer if the file has been extended.

Two additional variations of the random access mode allow for shared file and read only file access. A file which has been opened for shared access can be referenced by two or more different tasks at the same time. Only one file slot and one file pointer are used no matter how many tasks open the file. Hence, it is the responsibility of each user task to ensure data integrity by using the lock file or lock process commands. The file must be closed by all tasks when the processing is completed.

A read only random access to a file is independent of any other access to that file. A new file slot is always allocated when the file is read only opened and a write to the file is not permitted.

File Names

PDOS file names consist of an alphabetic character (A-Z or a-z) followed by up to seven additional characters. An optional one to three character extension is separated from the file name by a colon (:). Other optional parameters include a semi-colon (;) followed by a file directory level and a slash (/) followed by a disk number. The file directory level is a number ranging from 0 to 255. The disk number ranges from 0 to 255.

Legal file names:

FILE
A1234567:890;255/127
PROGRAM/3
FILE2;10

A file typed as a system I/O device driver (DR) has entry points directly into the channel buffer for OPEN, CLOSE, READ, WRITE, and POSITION commands.

If the file name is preceded by a "#", the file is created (if undefined) on all open commands except for read only open. When passing a file name to a system primitive, the character string begins on a byte boundary and is terminated with a null.

```
x>CF TEMP,#TEMP2/5
```

```
FILEN   DC.B    'FILE1/4',0
```

Special characters such as a period or a space may be used in file names. However, such characters may restrict their access. The command line interpreter uses spaces and periods for parsing a command line.

Directory Levels

Each PDOS disk directory is partitioned into 256 directory levels. Each file resides on a specific level, which facilitates selected directory listings. You might put system commands on level 0, procedure files on level 1, object files on level 10, listing files on level 11, and source files on level 20. Level 255 is global and references all levels.

```
x>LV
Level=1
x>LV 10
Was 1
```

A current directory level is maintained and used as the default level in defining a file or listing the directory when no directory level is specified.

Disk Numbers

A disk number is used to reference a physical secondary storage device and facilitates hardware independence. All data transfers to and from a disk are blocked into 256-byte records called sectors.

The range of disk numbers is from 0 to 255. Several disk numbers may share the same secondary storage device. Each disk can have a maximum of 65280 sectors or 16,711,680 bytes.

A default disk number is assigned to each executing task and stored in the task control block. This disk number is referred to as the system disk and any file name which does not specifically reference a disk number defaults to this parameter.

```
0>SY 1,0
Was 0
1,0>SY
Disk=1,0
1,0>_
```

PDOS supports multiple disk directory searches. Up to four disk devices can be associated with each task. When a file is referenced, each directory is searched (in order) until the file is found.

```
1>SY 1,2,3,4
Was 1
1,2,3,4>_
```

Some utility programs make use of the system disk for temporary file storage. By not specifying the disk parameter, the program becomes device independent and defaults to the current system disk.

When a task is created, the parent task's disk number(s) and directory level are copied into the task control block of the new task.

File Attributes

Associated with each file is a file attribute. File attributes consist of a file type, storage method, and protection flags. These parameters are maintained in the file directory and used by the PDOS monitor and file manager.

The file type is used by the PDOS monitor in processing the the file. For instance, a file typed as "EX" (a PDOS BASIC file), calls the BASIC interpreter which loads the file and begins execution with the first line number. A file typed as "OB" (a 68000 object module), calls the relocating loader to load the object into memory. If a start address tag is included at the end of the file, the module is immediately executed at that address. Otherwise, PDOS enters the program at the first loaded address.

The following are legal PDOS file types:

AC Assign console. A file typed "AC" specifies to the PDOS monitor that all subsequent requests for console character inputs are intercepted and the character obtained from the assigned file.

BN Binary file. A "BN" file type has no significance to PDOS but aids in file classification.

OB 68000 tag object file. Output from the MASM 68000 assembler is in tagged object form. The tag directs the PDOS monitor to load the file into memory (if there was a starting address tag) and execute the program.

SY System file. An "SY" file is generated from an "OB" file. MC68000 object is condensed into a memory image by the "SYFILE" utility. The first location of a system file is the program entry address.

BX PDOS BASIC binary file. A BASIC program stored using the "SAVEB" command is written to a file in pseudo-source token format. Such a file requires less memory than the ASCII LIST format and loads much faster. Subsequent reference to the file name via the PDOS monitor automatically restores the tokens for the BASIC interpreter and begins execution.

EX PDOS BASIC file. A BASIC program stored using the "SAVE" command is written to a file in ASCII or LIST format. Subsequent file reference via the PDOS monitor automatically causes the BASIC interpreter to load the file and begin execution.

TX ASCII text file. A "TX" type classifies a file as one containing ASCII character text.

DR I/O driver. A "DR" file type indicates that the file data is an I/O driver program and is executed when referenced. An I/O driver must be copied with the TF monitor command or MTRANS utility instead of the CF monitor command.

A PDOS file is physically stored in contiguous sectors whenever possible. A non-contiguous structure results from file expansions where no contiguous sectors are available. Contiguous files have random access times far superior to non-contiguous files. A contiguous file is indicated in the directory listing by the letter "C" following the file type.

File protection flags determine which commands are legal when accessing the file. A file can be delete and/or write protected.

File storage method and protection flags are summarized as follows:

C Contiguous file. A contiguous file is organized on the disk with all sectors logically sequential and ordered. Random access in a contiguous file is much faster than in a non-contiguous file since the forward/backward links are not required for positioning.

***** Delete protect. A file which has one asterisk as an attribute cannot be deleted from the disk until the attribute is changed.

****** Delete and write protect. A file which has two asterisks as an attribute cannot be deleted nor written to. Hence, READ, POSITION, REWIND, OPEN, and CLOSE are the only legal file operations.

+ File altered. A file which has a plus sign as an attribute has been altered. It is cleared whenever the IA monitor command is executed on the file.

Time Stamping

When PDOS is first initialized, the system prompts for a date and time. These values are then maintained by the system clock and are used for time stamping file updates, assembly listings, and other user defined functions.

```
PDOS/68000 R3.3
ERII, Copyright 1983-88
xxxxx BIOS          xx:68000
DATE=00-???-00 10-DEC-87
TIME=00:00:00 10:30
```

When a file is first created or defined, the current date and time is stored with the disk directory entry. This time stamping appears in the "DATE CREATED" section of a directory listing. From then on, the creation date and time are not changed.

When a file has been opened, altered, and then closed, the current date and time are written to the "LAST UPDATE" section of the disk directory entry. The time stamp indicates when the file was last altered by any user.

Ports, Units, and Disks

The terms ports, units, and disks are often confused and as such are explained here:

Ports

Ports are logical input channels and are referenced by numbers 0 through 15. Associated with each port is an interrupt driven input buffer. The BAUD PORT (BP) command binds a physical UART to a buffer.

Units

A unit is an output gating variable. Each bit of the variable directs character output to a different source. Bit 1 (LSB) is associated with U1P$(A6) output port. Likewise, bit 2 is associated with U2P$(A6) output port. The "SU" and "SPOOL" commands bind the other bits to the PDOS file structure.

Disks

A disk is a logical reference to a secondary storage device. Disk numbers range from 0 to 255. Several disk numbers may reference the same physical device. The system BIOS deciphers what the disk number means.

PDOS BIOS

The PDOS Basic I/O Subsystem (BIOS) configures the PDOS environment for different types of hardware peripherals. This includes UARTs, mappers, system LEDs, read/write sector primitives, and disk motor control. Other functions of the BIOS include startup parameters such as auto-start, PDOS prompts, default disk, RAM disk size and location, interrupt vector generation and processing, and MAIL array size.

For a list of the current configurable parameters along with their defaults for your system, refer to the MBIOS:SR file or your *Installation and Systems Management* guide.

The BIOS is linked with the PDOS kernel and UART module to form an execution module. The monitor and file manager are added to complete a PDOS system.

All PDOS hardware dependence is confined to the following three modules:

xxBIOS, which contains CPU-related parameters such as cold startup code, exception vector table, exception vector setup, DIP switches, memory mapper, clock acknowledgment, etc.

xxBIOSU, which has all terminal I/O routines interfacing to various UARTs.

xxBIOSW, which has the read and write logical sector routines. Another file, xxPARM:SR, is closely associated with the BIOS, is included when assembling the three BIOS modules, and defines various hardware addresses, offsets, and low parameter RAM locations used by the BIOS.

To make a PDOS port to new hardware, you need to write the above four files (xxBIOS:SR, xxBIOSU:SR, xxBIOSW:SR, and xxPARM:SR). You replace the "xx" characters with a mnemonic for the CPU card. The xxBIOS:SR file contains the first location executed (entry point) after a cold start. As it is assembled, it includes the xxPARM:SR file and the general MBIOS:SR file. xxBIOS:SR contains CPU-related parameters such as exception vector setup, DIP switch settings, memory mapper, and clock initialization and interrupt acknowledge routines.

xxBIOSU:SR contains all terminal I/O routines interfacing to various UARTs in the system, including input, output, port configuration, and XON/XOFF handshaking. xxBIOSW:SR has the read and write logical sector routines, along with controller initialize and Winchester partition routines. xxPARM:SR is an "include" file that defines some low parameter RAM locations that the BIOS uses as well as the board, chip, and register addresses peculiar to the hardware.

The PDOS BIOS module is composed of the user BIOS module (xxBIOS:SR) and a common PDOS BIOS module (MBIOS:SR). The user BIOS module is composed of the task startup table (R$TASK) and various routines called by the PDOS common BIOS module and the PDOS kernel. These routines are optional and are only included when needed.

xxBIOS:SR - User BIOS Module

The user BIOS module (xxBIOS:SR) consists of tables and routines specific to the system hardware. It contains the cold startup initialization subroutines, the BINTB exception vector table, the task startup table (R$TASK), the kernel subroutines, and the BIOS table. The BIOS table B$BIOS and certain kernel subroutines, along with some generic cold startup code, are contained in the MBIOS:SR file. The remaining routines, which are located in xxBIOS:SR, will have to be written by you if you are porting to new hardware.

The user BIOS module is organized as follows:

B$STRT - Cold start entry address & constants
B$TASK - Task startup table (ports & tasks)
B$CPU - Set CPU dependent parameters (subroutine)
B$RAM - Fix top of RAM (subroutine)
B$RSW - Read system switches (subroutine)

The following optional subroutines:
B$ACK - Acknowledge clock interrupt
B$LED - Blink LED & adjust clock
B$MAP - Load system map constant
B$SAV - Save hardware registers
B$RES - Restore hardware registers
etc.

BINTB - Interrupt vector table

Text from the generic BIOS file MBIOS:SR is then INCLUDEd at the end of this user BIOS module.

Task Startup Table

R$TASK is the task startup table, and consists of two parts. First, the ports to be bauded into the system are listed and then the tasks to be started are included.

Cold Startup Subroutines

Three optional subroutines are called from MBIOS:SR during cold startup: BCPU, BRAM and B$RSW. B$CPU is called right after MBIOS sets up a stack and decides if it is a 68000 or a 68010 chip. B$CPU performs various functions, depending on the hardware, including any of the following:

- Pause to allow the hardware to settle
- Initialize the module control registers
- Count the cards in the system using bus errors
- Perform any "one time only" initialization of any cards or chips that need it
- Write zeroes to RAM to initialize parity
- Set up mapper MMU
- Initialize the timer chip

After MBIOS sizes RAM in a non-destructive way, it calls B$RAM to allow you to reserve some RAM for disk buffers or whatever. A6 and A7 point to the beginning and end of tasking RAM and you can take some off either end, usually the TOP end.

Just before MBIOS enters the PDOS kernel startup, it calls B$RSW to allow you to alter the default baud rate, system disk, and autostart flag.

Kernel Subroutines

The B$BIOS table in MBIOS has some optional subroutines, which are called by the PDOS kernel at times to perform a BIOS-related function. If a particular routine is not included, i.e. the label is undefined in the BIOS, then the subroutine simply returns to the kernel with RTS. Except as noted, preserve all registers used within these routines.

B$ACK System clock interrupt acknowledge
B$CTB System create task
B$KTB System kill task
B$LED System blink LED
B$MAP System schedule task (load map)
B$PRT System protect
B$PSC Position cursor
B$CLS Clear screen
B$IRD Init RAM Disk
B$SAV Save on stack (68881 FPC)
B$RES Restore from stack (68881 FPC)
B$CMD Pre-process monitor commands
B$CLO Close file, flush sector cache

The following paragraphs describe the environment and purpose of the MBIOS subroutines:

B$ACK

System clock interrupt acknowledge routine is called from MPDOSK1 during each timer interrupt and is therefore in supervisor mode. B$ACK should "turn off" the timer interrupt line to the CPU by performing an acknowledge. Input registers are (A0)=B$BIOS table and (A5)=SYRAM.

B$CTB

System create task routine is called from MPDOSK1 in user mode just before each task goes to execute its program. The task number is in D0.B, (A1)=B$BIOS table, (A0)=routine address, and A5-A7 have the standard values. This routine, used with B$KTB below, could be used to keep track of active/inactive tasks, restrict access to disks, etc.

B$KTB

System kill task routine is called from MPDOSK1 in user mode after the task's files have been closed and before frames are popped of or the memory is freed back to the pool. The task number is in D0.B, (A0)=B$BIOS table, and (A5)=SYRAM.

B$LED

Blink LED routine is called from MPDOSK1 once each second to toggle a heartbeat LED. The other main function of B$LED, even used on systems with no LED, is to adust the PDOS clock. Input registers are D0.B=seconds (0-59), (A0)=B$BIOS table, and (A5)=SYRAM.

B$MAP

System load map register routine is called from MPDOSK1 in supervisor mode whenever a new task is rescheduled. It was originally intended to perform a mapper load, but can now be used as a task monitor to see which tasks are getting rescheduled most often, etc. The MAP word associated with the task is in D0.W, (A0)=B$BIOS table, (A5)=SYRAM, and (A6)=TCB.

B$PRT

System protect routine is not called by PDOS, but used to be called by a pair of utilities to write-protect and un-protect the memory area where PDOS resides.

B$PSC

Position cursor routine is called by MPDOSK2 in supervisor mode to output the expanded screen addressing string required by ANSI terminals. B$PSC is called only if the word at PSC$(A6)=0 or if the byte PSC$(A6)=$FF, D1.B=row (y-position),D2.B=column (x-position), (A0)=B$BIOS table, (A5)=SYRAM, and (A3)=character buffer for output. Two possible functions are performed by B$PSC:

- Build the expanded address string in (A3)+, returning (A3) pointing after the last byte of string, the RTS with SR=.NE., or
- Set word (A3) to an encoded simple position and RTS with SR=.EQ.

B$CLS

Clear screen routine is called by MPDOSK2 in supervisor mode to output the expanded screen clear string requiirred by ANSI terminals. B$CLS is called only if the word at CLS$(A6)=0 or if the byte CLS$(A6)=$FF, (A0)=B$BIOS table, (A5)=SYRAM, and (A3)=character buffer for output. Two possible functions are performed by B$CLS:

- Build the expanded clear string in (A3)+, returning (A3) pointing after last byte of string, the RTS with SR=.NE., or
- Set word (A3) to an encoded simple position and RTS with SR=.EQ.

B$IRD

Initialize RAM disk routine is called by MPDOSM in user mode to initialize the RAM disk whenever the RD monitor command is executed with the initialize option. D6.W=# of directory entries, D7.W=# of PDOS sectors, (A0)=base address of disk, and (A4)=B$BIOS table.

B$SAV

Save on stack routine is called by the PDOS kernel before every task context switch if the task save flag (SVF$) is set. Address register A1 contains the user stack pointer (USP) which is saved on the supervisor stack immediately on return. Address register A5 points to SYRAM, and register A0 points to the BIOS table.

B$RES

Restore from stack routine is called by the PDOS kernel after every task context switch if the task save flag (SVF$) is set. Address register A1 contains the user stack pointer (USP) which is restored to the 68000 USP register immediately on return. Address register A5 points to SYRAM, and register A0 points to the BIOS table.

B$CMD

Command line routine is called by MPDOSM in user mode just before a command line is processed. It can be used to pre-process or translate user command names into two-letter PDOS commands. For example, FILES or DIR could be translated into LS. (A1)=input command line and (A2)=B$BIOS table. The altered command line should be pointed to as well by (A1). Another possible function of B$CMD is to expand the resident monitor commands. Here B$CMD parses the command line for the new commands and then, instead of returning to the regular parser (RTS), simply execute the new command code, and exit with an XEXT or XERR primitive.

B$CLO

Close file routine is called by MPDOSF file manager in user mode whenever a file has been closed, right after the directory entry has been updated on the disk. B$CLO can be utilized by a user-supplied BIOS-resident sector caching scheme to flush the buffers to disk. D0.B is the disk number of the file, D1.W is the last sector written, (A0)=B$BIOS, and A5-A7 the usual values.

The only ones that the xxBIOS:SR files usually define are B$ACK and B$LED. B$ACK just acknowledges or resets the timer chip interrupt logic. B$LED may blink an LED if the hardware provides one and perform a general clock fine adjustment. The others are for custom use by those who may wish to keep track of tasks, protect the operating system, or use a complicated terminal type (e.g. ANSI).

Exception Vector Table

BINTB is the exception vector table whose entries define the hardware dependent vectors that are to be loaded into the 68000's exception table from $000 - $3FF. Entries in this table consist of a word exception offset followed by a long word routine offset address. The timer interrupt vector address must be set to call K1$CLKI entry into MPDOS:OBJ. UARTs can be set to call either K2$PINT, a general port looker, or K2$CHRI, a character found entry. The table is terminated with a zero word.

```
BINTB   DC.W    $100
        DC.L    LABEL-B$BIOS
        ...
        DC.W    0
```

The actual exception vector table is fixed at address $0000 on 68000 systems, but it can be located anywhere in memory on systems that use the 68010 or 68020 processor. The 68010/20 Vector Base Register, or VBR, can be accessed with the MOVEC instruction and is added to all exception table accesses. PDOS allows 68010/20 systems to have a non-zero VBR by setting the constant B.VEC to the required VBR address when assembling xxBIOS:SR. The BINTB table still contains the $000 to $3FC offsets from the VBR, and not the actual address of the exceptions.

Other possible uses for the BINTB entries might be pre-processors for the bus error or line-A exceptions. The bus error pre-processor may want to re-try the instruction that caused the bus error (68010/20) at fixed number of times before calling the PDOS K1$BERR entry. Or bus error could bring in a new page under a user-implemented page-fault memory managed system.

The line-A pre-processor could add new user-defined line-A instructions to the fixed list of PDOS primitives. To do this, add a BINTB table entry and an xxBIOS:SR routine which decodes the line-A, and goes either to PDOS or to the new processor.

BIOS Example

The following is an example of a user BIOS module.

```
TTL     xxBIOS:SR - 68K xxBIOS
*       xxBIOS:SR       11/17/86
*************************************************************************
*                                                                       *
*       xx    xx  xx    xx  xx    xx  xx    xx  xx    xx  xx    xx      *
*        xx  xx    xx  xx    xx  xx    xx  xx    xx  xx    xx  xx       *
*         xxxx      xxxx      xxxx      xxxx      xxxx      xxxx        *
*           xx        xx        xx        xx        xx        xx        *
*         xxxx      xxxx      xxxx      xxxx      xxxx      xxxx        *
*        xx  xx    xx  xx    xx  xx    xx  xx    xx  xx    xx  xx       *
*       xx    xx  xx    xx  xx    xx  xx    xx  xx    xx  xx    xx      *
*                                                                       *
*               BBBBBBBB   IIIIII    OOOOOOO     SSSSSS                 *
*               BB     BB    II     OO     OO   SS                     *
*               BB     BB    II    OO       OO  SS                     *
*               BBBBBBBB     II    OO       OO   SSSSSS                *
*               BB     BB    II    OO       OO        SS               *
*               BB     BB    II     OO     OO         SS               *
*               BBBBBBBB   IIIIII    OOOOOOO    SSSSSSS                *
*                                                                       *
*=======================================================================*
*=      REVISION SCHEDULE MODULE: xxBIOS
*=
xxBIOS IDNT    3.2     BIOS                IDNT label appears in QLINK map
*=
*=======================================================================*
*
       IFUDF   RF      :RF     EQU 0           ;RUN MODULE FLAG
       IFUDF   TPS     :TPS    EQU 100         ;TICS/SECOND
       IFUDF   CLKADJ  :CLKADJ EQU 0           ;CLOCK ADJUST
*
```

- RF = Run module flag
- TPS = System tics per second
- CLKADJ = Clock adjustment factor

```
        OPT     ARS,CRE
        SECTION 14
        PAGE
```

- MASM options for short absolute references and cross reference.
- BIOSes are SECTION 14 code.

Define task startup table external for run module assembly. Also add EPROM 68000 startup vector.

```
*************************************************
*       RUN MODULE SECTION
*
        IFNE    RF
        XREF    R$TASK,S$PROM
        DC.L    SYZ.+S$SRAM     ;SUPERVISOR STACK POINTER
        DC.L    BSTRT           ;STARTUP VECTOR
        ENDC
*
*************************************************
*       PDOS ENTRY POINT
*
        XDEF    B$STRT          ;BIOS STARTUP ENTRY POINT
        XREF    B$SRAM          ;ADDRESS OF SYRAM POINTER
        XREF    S$SRAM          ;SYSTEM RAM
*
```

- B$STRT = PDOS cold start entry address
- B$SRAM = System RAM variable
- S$SRAM = System RAM (defined at link time)
- PDID = 'PDOS'
- SYID = System identification

```
B$STRT BRA.L    BSTRT           ;BOOT EPROM START
       DC.L     PDID            ;PDOS BOOT IDENTIFICATION
       DC.W     SYID            ;SYSTEM ID
B.SRAM DC.L     S$SRAM          ;SYRAM ADDRESS
       XREF     U.1ADR,U.1TYP
       XREF     U.2ADR,U.2TYP
*
*************************************************
*       TASK STARTUP TABLE (NON-RUN MODULE)
*
       IFEQ     RF
       XDEF     R$TASK

*
R$TASK DC.B     1,U.1TYP,BIBR,%0000     ;PORT #1
       DC.L     U.1ADR
       DC.B     2,U.2TYP,BIBR,%0000     ;PORT #2
       DC.L     U.2ADR
       DC.W     0                       ;END-OF-TABLE
*
```

- U.xADR = UART base address
- U.xTYP = UART type
- R$TASK = Task Startup Table

```
*       TASK #0
*

        DC.B    64              ;PRIORITY
        DC.B    TT              ;TASK TIME
        DC.L    0               ;DSEG SIZE
        DC.W    0               ;MAP
        DC.L    *-*             ;PSEG START (0=MBEGN)
        DC.W    1               ;PORT #

        Insert other startup tasks here

        DC.W    0               ;END OF TABLE
        ENDC

*
BMES01  DC.B    $0A,$0D,'xxBIOS ',$DATE,0
        EVEN


************************************************
*       CPU DEPENDENT PARAMETERS
*
B.PTMSK         EQU     $2500       ;PORT DISABLE INT MASK
SYID    EQU     'xx'                ;SYSTEM ID WORD
*
```

- 64 = Task priority (1-255)
- TT = Task time slice
- 0 = RAM size (0=use all)
- 0 = Mapper constant
- *-* = Task entry address (=Monitor)
- 1 = Task port number
- BMES01 = BIOS startup message
- B.PTMSK = Disable all port interrupts

```
************************************************
*       CPU DEPENDENT SETUP ROUTINES
*
B$CPU   EQU     *               ;CPU SETUP
RTS
*
```

The B$CPU routine initializes the system. This may include the system clock, memory mapper, interrupts, controllers or any other CPU dependent parameters.

```
************************************************
*       FIX TOP OF RAM
*
B$RAM   EQU     *               ;RAM FIX
B$RAM In: (A2) = Top of RAM
                 (A4) = BIOS table
                 (A5) = SYRAM
                 (A7) = (Top of RAM)-4 (RTS)

        RTS
*
```

The B$RAM routine is called after memory has been sized. It is here that the top of memory (A7) can be adjusted.

```
*************************************************
*       READ SWITCHES
*
B$RSW   EQU     *               ;READ SWITCHES
B$RSW In: D4.L = SYRAM (B.BAS) bit map base (=0)
                 D5.W = Baud rate (-1=none)
                 D6.L = B.VEC=vector base register (=0)
                 D7.L = NODE.B/ASF.B/FLG$.B/SDK$.B
                 (A3) = Interrupt vector table (BINTB)
                 (A4) = BIOS table (B$BIOS)
                 (A6) = Start of tasking memory
                 (A7) = End of tasking memory
        RTS
```

The B$RSW routine is called just before entering the PDOS kernel. It is here that system switches can be read and the initial baud rate (D5.W), auto-start flag (ASF.B), or system disk (SDK$.B) adjusted.

```
*************************************************
*       ACKNOWLEDGE CLOCK INTERRUPT
*
B$ACK   EQU     *               ;ACKNOWLEDGE CLOCK
RTS
*
```

- B$ACK = Acknowledge clock interrupt

```
*************************************************
*       BLINK LED & ADJUST CLOCK
*
B$LED   EQU     *               ;BLINK LED
        MOVE.L  B_CLK(A0),D0    ;ADJUST CLOCK?
          BEQ.S @0002           ;N
        ADD.L   D0,BCLK.(A5)    ;Y, ADJUST COUNT, CARRY?
          BCC.S @0002           ;N
        ADDQ.W  #1,FCNT.(A5)    ;Y, UP COUNTER
*
@0002   RTS                     ;RETURN
*
```

- B$LED = Blink LED & adjust clock

```
*************************************************
*       LOAD SYSTEM MAP CONSTANT
*
B$MAP   EQU     *               ;LOAD MAP CONSTANT
RTS
*
```

- B$MAP = Load system map constant

```
*************************************************
*       SAVE 68881 REGISTERS ON USER STACK
*
        OPT     P=68020,P=68881,OLD
*
B$SAV   FSAVE   -(A1)
        FMOVEM.X FP0-FP7,-(A1)  ;SAVE 68881 REGS FP0-FP1
        FMOVE.L FPCR/FPSR,-(A1) ;SAVE STATUS REGISTER
        RTS                     ;RETURN
*
```

- **B$SAV - Save on stack**

```
*************************************************
*       RESTORE FROM STACK
*
B$RES   FMOVE.L (A1)+,FPCR/FPSR
        FMOVEM.X (A1)+,FP0-FP7
        FRESTORE (A1)+
        RTS                     ;RETURN
        PAGE
```

- **B$RES - Restore from stack**

All system-dependent exception vectors are built from three-word entries of the following format:

```
*************************************************
*       xxBIOS INTERRUPT STRUCTURE
*
BINTB   EQU     *               ;INTERRUPT TABLE
        DC.W    $007C           ;INT LEVEL 7 PROCESSOR
        DC.L    BINT7-B$BIOS
        DC.W    <address>
        DC.L    <routine>-B$BIOS ;ADDITIONAL VECTORS
        ....
        DC.W    0               ;END-OF-TABLE
*
```

Finally, the common MBIOS:SR module is included to complete the BIOS module.

```
        INCLUDE MBIOS:SR     ;INCLUDE COMMON BIOS MODULE
        END
```

MBIOS:SR - Common BIOS Module

The common BIOS module (MBIOS:SR) is included at the end of the user BIOS module. It has many default equates that also can be adjusted at assembly time. The BIOS configuration table (B$BIOS) drives the PDOS system and is pointed to by the first long word of SYRAM.

The MBIOS:SR file contains the user-changeable BIOS table, through which the PDOS kernel communicates with the UARTs, disks and hardware. The table includes pointers to the UART BRA.S tables, the disk read and write routines, the task startup table, along with various constants for timed events and default clear screen codes.

BIOS Table

At the end of the B$BIOS table of the BIOS, there are some system configuration-dependent parameters at specific offsets in the table. These values, which vary from system to system, may be needed by applications that access variable SYRAM tables, perform battery clock or bootstrap functions, service multiprocessor systems, or install drivers. The following table defines the PDOS system parameters which are available here. First is listed the variable's label name, the B$BIOS table offset, and the size of the value (word or long - W/L).Then the file where the variable comes from is listed, along with whether the value is a number or a SYRAM offset (V/O); followed by a definition of the variable:

Name	Offset	W/L	Source	V/O	Definition
TBE$	$A0	Word	MPDOS	Value	Task control block size in bytes (must always = $500)
MAPB.	$A2	Word	SYRAM	Offset	System memory bit map base address SYRAM offset. The actual table, MAPS., follows this entry.
NMB.	$A4	Word	SYRAM	Value	Map size, or # of bytes in MAPS. table
PORT.	$A6	Word	SYRAM	Offset	Input character buffers
NPS.	$A8	Word	SYRAM	Value	Number of I/O ports
NCP	$AA	Word	SYRAM	Value	Number of characters/port
IOUT	$AC	Word	SYRAM	Offset	Output character buffers
TQUE	$AE	Word	SYRAM	Offset	Task queue
TLST	$B0	Word	SYRAM	Offset	Task list
NTB.	$B2	Word	SYRAM	Value	Maximum number of tasks
TBZ.	$B4	Word	SYRAM	Value	Task list entry size
TMTF.	$B6	Word	SYRAM	Offset	To/from/index table
TMBF.	$B8	Word	SYRAM	Offset	Task message buffers
NTM.	$BA	Word	SYRAM	Value	Number task messages
IMZ.	$BC	Word	SYRAM	Value	Task message size
TMSP.	$BE	Word	SYRAM	Offset	Task message pointers
NTP.	$C0	Word	SYRAM	Value	Number of task message pointers
DEVT.	$C2	Word	SYRAM	Offset	Delay event list
NEV.	$C4	Word	SYRAM	Value	Number of delay events
XCHB.	$C6	Word	SYRAM	Offset	Channel buffers
NCB.	$C8	Word	SYRAM	Value	Number of buffers
XFSL.	$CA	Word	SYRAM	Offset	File slots
NFS.	$CC	Word	SYRAM	Value	Number of file slots
TSEV.	$CE	Word	SYRAM	Offset	Task schedule event code table
RDKL.	$D0	Word	SYRAM	Offset	RAM disk list
NRD.	$D2	Word	SYRAM	Value	Number of RAM disks in list
DRVL.	$D4	Word	SYRAM	Offset	Installed driver list
UTLL.	$D6	Word	SYRAM	Offset	Installed utility list
B.NUT	$D8	Word	BIOS	Value	Number of UART types
	$DA	Word			<Spare>
	$DC	Word			<Spare>
	$DE	Word			<Spare

The following are used by the multi-processing library:

VME	$E0	Long	BIOS	Value	Base of VMEbus
RVA	$E4	Long	BIOS	Value	RAM address from VMEbus
GRA	$E8	Long	BIOS	Value	Global RAM address

The following may be used by the MMKBT PDOS utility:

PLA	$EC	Long	BIOS	Value	Default PDOS load address
BTS	$F0	Word	BIOS	Value	Bootstrap sct (-1=none)

The following may be used by the MTIME PDOS utility:

CKT	$F2	Word	BIOS	Value	Battery clock type
CKA	$FA	Long	BIOS	Value	Battery clock address

MBIOS Switches

MBIOS:SR contains some user-alterable, cold start-up code which initializes the hardware, sizes memory, sets up the RAM disk, and loads registers, then branches to the generic PDOS kernel startup entry point.

AS=Auto Start	Execute SY$STRT on boot up (=0, no autostart)
BR = 0	Initial baud rate
HR undefined	Highest memory address (else size using bus errors)
ANS=1	ANSI 3.64 PSC/CSC
FDR=0	Directory flag
TPS=100	Tics/second (system dependent)
SD=0	Default disk number
LV=1	Default file directory level
CPSP Long=$AA00	Clear screen and position
$9B3D	cursor code
EV112=TPS/5	Event 112
EV113=1	Event 113
EV114=10	Event 114
EV115=20	Event 115
MSZ=256	Mail array size
RU=8	RAM disk unit
RZ=255	RAM disk size
RE=16	Number of directory entries
RA=0	RAM disk address
IRD=1	RAM disk initialization

The switches are described in detail following. Most of them have default values.

AS

AUTO START. This switch determines whether or not the SY$STRT (auto start) file is to be executed on startup. If AS=0, then the SY$STRT file is not executed. If it is non-zero, it is executed. Default=1.

BR

INITIAL BAUD RATE. This is a number from 0 to 8 which represents the initial baud rate for the character I/O ports. The default is 0.

0=19200	5=600
1=9600	6=300
2=4800	7=110
3=2400	8 =38400
4=1200	

HR

HIGHEST MEMORY ADDRESS. The high memory address variable determines whether memory is sized (HR undefined) or fixed (HR=top address). Default is undefined.

ANS

ANSI 3.64 PSC/CSC. If this switch is equal to 1, then the BIOS subroutine for clear screen and position cursor for ANSI 3.64 terminal support is included. Default=1.

FDR

DIRECTORY FLAG. The directory flag determines the mode of access for the file manager. When the flag is zero (plus byte), all levels are global; When the flag is set to $80 (minus byte), then files are unique to each directory level. The only exception is level 0 which is global to all. Default is 0 for soft level partitioning.

/FDR=0	global levels
/FDR=$80	unique levels

TPS

TICS/SECOND. The tics/second variable sets the number of clock interrupts that are equivalent to one second. Default is system dependent. You cannot vary this switch without altering B$CPU clock chip initialization.

SD

DEFAULT DISK #. The default disk number determines which disk number is selected when no disk is specified by a filename. Default=0 and may be altered by the B$RSW routine.

LV

DEFAULT LEVEL #. The default level number determines which level in a disk directory is selected when no level is specified. Default=1.

CPSC

LONG CLEAR SCREEN AND POSITION CURSOR CODE. The clear screen codes are used by the XCLS primitive. Default=$AA00. The position cursor codes are used by the XPSC primitive. Default=$093D. Set CPSC=$AA00093D.

EV112

EVENT 112. The event 112 variable is decremented every clock interrupt. Default=TPS/20.

EV113

EVENT 113. The event 113 variable is decremented every second. Default=1.

EV114

EVENT 114. The event 114 variable is decremented every second. Default=10.

EV115

EVENT 115. The event 115 variable is decremented every second. Default=20.

MSZ

MAIL ARRAY SIZE. The mail array size is in bytes. Default=256 and is best as a multiple of 256.

RU

RAM DISK UNIT. The RAM disk is selected by "RU= ". Default=8.

RZ

RAM DISK SIZE. The size of the RAM disk determines how much memory to allocate. If RZ=0, then no RAM disk is selected. Default=255.

RE

OF DIRECTORY ENTRIES. The number of directory entries in the RAM disk is selected by "RE= ". Default=32.

RA

RAM DISK ADDRESS. The address variable determines where the RAM disk is located. If RA=0, then the RAM disk is allocated off the top of memory. Otherwise, the parameter indicates the memory address of a RAM disk. Default=0.

IRD

RAM DISK INITIALIZATION. The RAM disk is initialized by the PDOS BIOS by setting IRD=1 (default).

The following switches are also found in the MBIOS module:

TT=1 Default task time for tasks created with the CT monitor command.

Command line control keys and symbols:

Changing the following default switch settings will alter the way the PDOS utilities and other commands work.

B.ADD	**Ctrl A**	Recall last line
B.LFT	**Ctrl H**	Move left
B.RGT	**Ctrl L**	Move right
B.DRT	**Ctrl D**	Delete character under cursor
B.DLT	**Del**	Delete left
B.BRK	**Ctrl C**	PDOS break
B.CLR	**Ctrl X**	Clear buffer and window control character (B.WND)
B.CMD	.	Command delimiter
B.EXT	:	File extension
B.LEV	;	File level
B.DSK	/	File disk
B.WC1	*	Character wild card
B.WC2	@	Field wild card

B.SZ1=4 Auto-create task size.

B.SZ2=32 Create task (CT) task size.

B.TEV=64 Toggle event number.

B.BAS=0 Memory bit map base address.

B.VEC=0 Vector base register.

B.NUT=4 Number of UART types.

The following switches contain clock and bootstrap values:

PLA=$800 PDOS load address.

BTS=-1 Bootstrap sector.

CKT=-1 Battery clock type. (-1=none).

CKA=-1 Battery clock address.

The following switches contain multi-processor values:

VME=0 Base of VMEbus.

RVA=-1 RAM address from the VMEbus.

GRA=-1 Global RAM address.

NODE=0 Node letter.

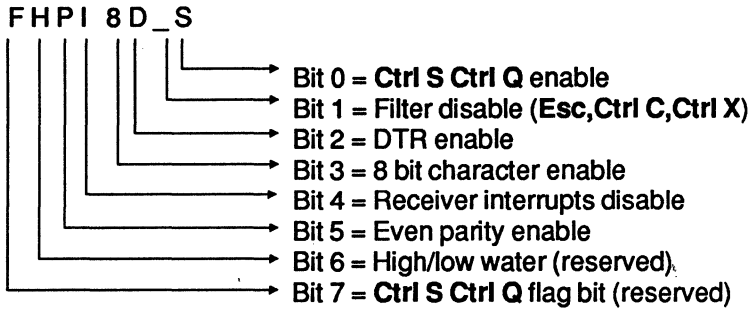
MBIOS:SR also has the monitor prompt routine B$MPT. The default subroutine just outputs the disk numbers and an angle bracket (0>). The following two DOSGEN flags can be defined to alter the prompt:

- LF is the Level Flag. If set non-zero, it outputs a semi-colon and the current level before the angle bracket.
- WF is the Window Flag. If set non-zero, it outputs the windowing port number and a colon before the disk numbers.

xxBIOSU - UARTs

PDOS UARTs are grouped into types. UARTs of the same type are differentiated by separate base addresses. The baud port function of PDOS binds a port to a particular type and base address. UARTs of the same type may be either chips of the same kind (i.e. having the same register offsets from the base address), or chips of various kinds which are on the same card (e.g. UARTs and parallel printer port all on a CPU card). Each UART type has a Device Service Routine (DSR) associated with it. Whenever PDOS interacts with a UART, all communication is accomplished through the DSR of the UART's type. The address of the DSR for a particular UART type is found in the B$BIOS table. The DSR address points to a structured table of entries, which perform the various functions needed. Pointers to DSRs for up to 8 (default=4) different UART types are maintained in the B$BIOS table.

In addition to type and address, a UART also has a flag byte associated with it. This byte is maintained in the SYRAM table F8BT. and the bits are defined as follows:



The DSR address points to a table of branch short (BRA.S) instructions, which are called from MPDOSK2 with a JSR. This means that they all exit with an RTS. To preserve the table structure, all branches MUST be short. Eliminate range errors with additional BRA.L links and not by altering BRA.S to BRA.L.

```
U$xDSR  BRA.S   UxDG             ;GET A CHARACTER
        BRA.S   UxDP             ;PUT A CHARACTER
        BRA.S   UxDB             ;BAUD THE PORT
        BRA.S   UxDR             ;RESET THE PORT
        BRA.S   UxDS             ;READ PORT STATUS
        BRA.S   UxHW             ;HIGH WATER
        BRA.S   UxLW             ;LOW WATER
        DC.B    'U0'             ;(optional) INSTALLABLE KEYWORD
        BRA.S   UxDI             ;(optional) INSTALL CARD
        DC.B    'Type x UART',0  ;(optional) CARD IDENTIFIER
        EVEN
```

The first seven routines are required for all DSRs, but only DSRs which are meant to be installable require the keyword, install card code, and card identifier message. The UART Device Service Routines can come from three different places.

- Included in the standard system xxBIOS:SR file.
- Linked installable DSR at DOSGEN time.
- Installed from a disk file after PDOS is running, by setting the DSR address into the B$BIOS table.

Each DSR subroutine should preserve all registers, except as noted in the following detailed description:

UxDG

Get Character

Inputs Supervisor mode (interrupts disabled)
Outputs D0.B = Character
A0.L = UART base address
SR = .EQ. >> character found
.NE. >> no character found
.CS. >> character found but ignore
Called By MPDOSK2 during K2$PINT interrupt service, must check all UARTs of this type for a received character. Can destroy A0 and D0. You must load up both D0 and A0 if a character is found. Also may be called from xxBOOT program, or from XGCR if receiver interrupts are disabled.

UxDP

Put Character

Inputs D0.B = Character
D1.L = Output EVENT.W | PORT FLAGS.W
A0.L = UART base address
Supervisor mode (interrupts enabled)
Outputs SR = .EQ. >> Character was output
.NE. >> Character was not output
Called By MPDOSK2 to output a single character, returning in status whether or not the character was sent. After 10 .NE. returns as a single character, MPDOSK2 executes an XSWP. Output event ranges from 80-95, and the port flags are defined above. May also be called from xxBOOT program or TTA drivers.

UxDB

Baud Port

Inputs D0.W = Baud rate (0-8)
D1.L = Output EVENT.W | PORT FLAGS.W
A0.L = UART base address
Supervisor mode (interrupts disabled or enabled)
Outputs SR = .EQ. >> UART successfully bauded
.NE. >> Baud was unsuccessful
Called By MPDOSK1 and K2 to configure a PDOS port. It must determine if the base address is legal for this type, and then set the data bits, stop bits, baud rate, parity, and enable receiver interrupts, if requested. It then returns in status whether or not the port was bauded. Output event ranges from 80-95, and the port flags are defined above. May also be called from xxBOOT program.

UxDR

Reset Port

Inputs A0.L = UART base address
Supervisor mode (interrupts enabled)
Outputs SR = .EQ. >> UART successfully reset
.NE. >> Reset was unsuccessful
Called By MPDOSK2 when you baud a port that has already been defined (in case the port is being bauded to a different UART).

UxDS

Read Port Status

Inputs A0.L = UART base address
Supervisor mode (interrupts enabled)
Outputs D0.W = UART status
Called By MPDOSK2 when executing XRPS read port status primitive.

UxHW

Signal High Water

Inputs D1.W = Port flags
A0.L = UART base address
Supervisor mode (interrupts disabled)
Outputs <none>
Called By MPDOSK2 when a received character passed the 2/3 full mark. Based upon the flag byte, it should stop the external device from sending characters either by setting a hardware bit or by sending a **Ctrl S** (XOFF).

UxHW

Signal Low Water

Inputs D1.W = Port flags
A0.L = UART base address
Outputs <none>
Called By MPDOSK2 when a character is taken from a type-ahead buffer and passes the 1/3 full mark. Based upon the flag byte, it should re-enable the external device to sending characters either by resetting a hardware bit or by sending a **Ctrl Q** (XON).

UxDI

Install Card (optional)

Inputs (A1) = K1$BEGN kernel routine table
A2.L = 0 or new card base address
(A5) = SYRAM
Supervisor mode
Outputs D1.W = # of cards found
Called By MPDOSK1 (with interrupts disabled) before bauding the R$TASK table ports and by the INSTALL utility (with interrupts enabled). Preserve D1 and A4 for K1 call. See examples for setting interrupt vector, counting cards, etc.

Interrupt Inputs

All UART receivers should produce interrupts for input. You can have the port type-ahead buffers filled with polling by disabling receiver interrupts in the flag byte. UART interrupts can be pointed to the internal PDOS ISR, called K2$PINT. This routine calls all the UART types' "get character" entries until one returns .EQ. with the character. If the UARTs can generate individual interrupt vectors, individual routines can be written for each port to get the character, store some registers and enter the PDOS ISR at K2$CHRI right after the .EQ. return from the "get character" polling of K2$PINT. This provides much faster response.

Parallel Port Interrupts

PDOS is designed for polled character output (in other words, it is not designed for efficient interrupt driven output). However, try to implement interrupts on parallel ports whenever possible.

xxBIOSW - Read/Write Disk DSRs

The read/write sector routines are supplied in the xxBIOSW:SR module. Four entries are supplied for read, write, initialize, and check for floppy motor off. An additional entry is for an error message list used by the PDOS monitor module to report disk errors.

The entry points are as follows:

W$XWSE - Write sector
W$XRSE - Read sector
W$XDIT - Initialize disks
W$XDOF - Check for disk off
W$ERM - Error message list

An annotated example follows for the xxBIOSW module.

```
 TTL     xxBIOSW:SR - 68K R/W SECTOR BIOS
 *       xxBIOSW:SR      05/07/84
*******************************************************************************
 *
 * DDDDDDD   IIII   SSSSSS   KK    KK        RRRRRRRR \\  WW                  WW
 * DD    DD   II   SS        KK   KK         RR      RR \\  WW                WW
 * DD     DD  II   SS        KK KK           RR      RR  \\  WW              WW
 * DD     DD  II    SSSSSS   KKKKK           RRRRRRRR     \\  WW    WWW     WW
 * DD     DD  II         SS  KK  KK         RR   RR        \\  WW  WW WW  WW
 * DD    DD   II         SS  KK   KK        RR    RR        \\  WWWW   WWWW
 * DDDDDDD   IIII  SSSSSSS   KK    KK       RR     RR        \\  WW     WW
 *
 *=*****************************************************************************
 *=      REVISION SCHEDULE MODULE: SBIOSW
 *=
 *=
 xxBIOSW         IDNT    3.0     M68000 PDOS
 *=
 *=*****************************************************************************
 *       PDOS R/W SECTOR MODULE
 *
        XDEF    W$XWSE,W$XRSE
        XDEF    W$XDIT,W$XDOF
        XDEF    W$ERM

 ************************************************
 *       INITIALIZE DISKS
 *
 W$XDIT EQU      *               ;INITIALIZE DISKS
        RTS
 *
```

The disk controllers are initialized. Any memory tables or communication variables are also set to a known state.

```
 ************************************************
 *       DISK OFF ROUTINE
 *
 W$XDOF EQU      *               ;DISK OFF
        RTS
 *
```

This routine is called once every second from the PDOS kernel. It is intended for controllers of 5 1/4" floppies where the motor is turned off after a certain length of time with no access.

```
 ************************************************
 *       WRITE SECTOR
 *
 *       IN:     D0.W = DISK #
 *               D1.W = LOGICAL SECTOR #
 *               (A2) = BUFFER ADDRESS
 *       OUT:      SR = EQ...WRITE COMPLETE
 *                      NE...D0.L = ERROR
 *
 W$XWSE EQU      *                       ;WRITE SECTOR
        MOVEQ.L #0,D0                   ;SET STATUS .EQ.
        RTS
```

The write sector routine outputs the logical 256-byte sector pointed to by address register A2 to the disk. Data register D0.W selects the disk number and register D1.W is the logical sector number. The status is returned EQUAL if the operation completed with no error. Otherwise, a NOT EQUAL status is returned with D0.L containing the error number.

```
************************************************
*       READ SECTOR
*
*       IN:     D0.W = DISK UNIT #
*               D1.L = LOGICAL SECTOR #
*               (A2) = BUFFER ADDRESS
*       OUT:      SR = EQ...WRITE COMPLETE
*                      NE...D0.L = ERROR
*
W$XRSE  EQU     *               ;READ SECTOR
        MOVEQ.L #0,D0           ;SET STATUS .EQ.
        RTS
*
```

The read sector routine reads the logical 256-byte sector from a disk into the memory buffer pointed to by address register A2. Data register D0.W selects the disk number and register D1.W is the logical sector number. The status is returned EQUAL if the operation completed with no error. Otherwise, a NOT EQUAL status is returned with D0.L containing the error number.

```
******************************************************
*       COMMON ERROR NUMBERS
*
ERR100  MOVEQ.L #100,D0         ;ILLEGAL DISK #
        RTS                     ;RETURN .NE.
*
ERR101  MOVEQ.L #101,D0         ;SECTOR TOO LARGE
        RTS                     ;RETURN .NE.
*
************************************************
*       ERROR MESSAGE LIST
*
W$ERM   DC.W    100                             ;Error list bias
        DC.B    'Illega',-'l','drive',0         ;100 Common errors
        DC.B    'Secto',-'r','to',-'o','big',0  ;101
        ....

        DC.B    -1                              ;End
```

Disk Read/Write

The inputs to the disk read and write routines, W$XRSE and W$XWSE, are as follows:

D0.B = logical disk number
D1.W = logical sector number
(A2) = data destination (read) or source (write) address

The output from the routines are status .EQ. if the operation was successful. If it was not, an .NE. status is returned and D0.L is the error number. Disk error numbers range from 100 to 199, with corresponding messages in the W$ERM table.

Cold Startup Initialize

The kernel startup calls the disk initialize routine, W$XDIT, before interrupts are enabled. This routine should preserve all registers and perform all controller initialization. When implementing the PDOS disk standard, the XDIT routine should call each disk controller's routine, if that controller is in the system (as indicated by the flags set by B$CPU). The registers A4, A5, and A6 are set up as parameter RAM pointers for storing the disk partition information, as follows:

```
*****************************************
*       GENERAL DISK INITIALIZE:
*         Based upon P$V320 & P$RWIN flags,
*         init the installed controller(s) and load
*         up the parameter RAM as you find drives.
*****************************************
*
RL      REG     D0-A6
*
W$XDIT  MOVEM.L RL,-(A7)
        LEA.L   P$PARM,A4
        MOVEA.L A4,A6           ;SAVE P$FPARM
        CLR.L   (A4)+           ;NO FLOPPY FOR NOW
        CLR.L   (A4)+
        MOVEA.L A4,A5           ;SAVE P$WPARM
        CLR.L   (A4)+
        CLR.L   (A4)+
        CLR.L   (A4)+
        CLR.L   (A4)+           ;(A4) POINTS TO DRIVE 0 PARM AREA
        TST.B   P$V320          ;IS V320 IN SYSTEM?
          BEQ.S @010            ;N
        BSR.L   W$XDITV         ;Y, DO V320 INIT
*
@010    TST.B   P$RWIN          ;N, RWIN1 IN?
          BEQ.S @020            ;N
        BSR.L   W$XDITR         ;Y, DO RWIN1 INIT
*
@020    MOVE.L  P$PARM+4,D0     ;IS FLOPPY 1 DEFINED?
          BEQ.S @030            ;N
        MOVEA.L D0,A0           ;Y, GET POINTER
        ADDQ.W  #1,PART$(A0)    ;SET AS DISK #1
*
@030    MOVEM.L (A7)+,RL
        RTS                     ;AND RETURN
```

Kernel Subroutine

The PDOS system clock interrupt calls the disk off routine, W$XDOF, once a second so that the BIOSW module can perform custom functions for the floppy select and motors, Winchester, or buffer timers.

Error Message Table

The error message table, W$ERM, is used by the PDOS monitor to help the user interpret disk error numbers. This table begins with a word indicating the error number that corresponds to the first message in the table that follows. Then the messages continue, separated by nulls, and terminated by an $FF byte. Any error numbers that have no message must have a zero in the table to keep messages that follow in sync. The BIOSW module must end on an even address.

```
*************************************
*       ERROR MESSAGE LIST
*
W$ERM   DC.W    100
        DC.B    'Illegal drive',0       ;100 Common errors
        DC.B    'Sector too big',0      ;101
        DC.B    'FDC Not ready',0       ;102 No retry (Floppy errors)
        DC.B    'FDC No ID AM detect',0 ;103 Retry
        DC.B    'FDC Write protect',0   ;104 No retry
        DC.B    'FDC Wrong sector',0    ;105 Retry
        DC.B    'FDC Wrong cylinder',0  ;106 Recal, retry
        . . .
        DC.B    'HDC Wrong track',0     ;121
        DC.B    -1
        EVEN
```

Interrupts

Disk controllers implement various interrupt methods. Some are usable while others are useless. From a multi-tasking perspective, it is best to have a task waiting for a disk controller suspend itself until the disk is done. Of course, the task should also suspend on a delayed local event as a timeout precaution. If interrupts are used, then a flag should be provided to turn the use of interrupts on and off. Code should be written either to use disk interrupts or just to poll the controller registers. Both must usually be provided since the disk initialize code doesn't have interrupts enabled and boot ROMs (which share the same disk code) do not have the PDOS "suspend" primitives available.

If you decide not to use interrupts, you should at least "XSWP" swap your task out during long loops that wait for the disk.

As a convention, event 119 is reserved for disk suspension. For increased speed, it is reset and set directly in SYRAM, not using the PDOS event primitives. The disk controller interrupt service routines usually just set event 119 and get the controller to stop interrupting, leaving other error checking to the main routine. These interrupt routine labels are XDEFed out to the xxBIOS:SR module, which includes them as entries in the BINTB exception vector table.

PDOS Winchester Standard

The PDOS Winchester standard keeps all the information about the Winchester drive on the Winchester drive. This allows you to do the following:

- Use a drive with any number of heads and cylinders
- Divide up the drive into any combination of large and small partitions
- Automatically skip all tracks with manufacturing defects

The PDOS Winchester standard information is contained in a block of data that resides in one or two sectors (usually sector 0) of physical track 0 on each Winchester drive in the system. The Drive Data Block (DDB) consists of three parts:

- The drive parameters
- A variable length partition definition table
- A variable length bad track list

These tables are built and written to the drive by the xxFRMT utility. They are then read into the parameter RAM area by the xxBIOSW disk initialize subroutine, W$XDIT, and subsequently used by the read/write sector code, W$XRSE and W$XWSE, in the xxBIOSW disk module.

The following discussion of the PDOS Winchester standard uses a strict definition of terms. These definitions are found in the glossary.

System Independent Drive Parameters

To allow the use of any size Winchester drive in the PDOS system, the drive parameters are read in from the drive itself. These include the number of heads and cylinders. During disk initialization, if a SCSI (SASI) controller is used in the system, either a "Set Drive Parameters" or an "Initialize Drive Characteristics" command is sent to the SCSI controller using the number of heads and cylinders specified in the disk's header sector. Thus, any drive in any PDOS system could actually have any number of heads or cylinders, limited only by the controller or hardware.

Disk Partitions on Drive Header

Each PDOS Winchester standard drive has all the necessary disk partition information in the header data. There is a three-word entry for each partition of the drive, consisting of a PDOS disk number, a logical base track, and a logical top track number. The PDOS read/write sector routines in xxBIOSW try to match the requested logical disk number to the disk number associated with a disk partition on an installed Winchester drive. The partition's associated base and top tracks are used to bias the requested PDOS sector number to an actual physical or SCSI logical block number. The number of partitions possible on any one drive or system may be limited by any of the following:

- The amount of data read in by W$XDIT
- The data written out by xxFRMT
- The amount of room in low parameter RAM

See the source code or the *Installation and Systems Management* guide for effective limits.

Bad Track Mapping

Following the partition information in the drive's header is an optional bad track list. This table consists of word entries in increasing order of physical track numbers that should not be accessed (skip them). The logical track number is incremented one for each bad track that is numbered lower than or equal to the requested track. The result is a mapped physical track that corresponds to the requested logical track number, where the physical track number is greater than or equal to the logical track number.

Drive Data Blocks (DDBs)

Each PDOS system allocates, in its system parameter RAM, a table of six Drive Data Block addresses -- two for floppy drives and four for Winchester drives. The addresses of the Drive Data Blocks are stored by the xxBIOSW disk initialize routine, W$XDIT, when PDOS first starts up. The actual DDBs are usually stored in the system's parameter RAM area by W$XDIT immediately following the six addresses of the P$PARM table.

If more than one type of disk controller is possible in a particular system, then the general W$XDIT routine calls the individual XDIT routines for each controller installed. These routines usually initialize the controller, and then loop through all possible drive select codes, looking for drives (floppy or Winchester) that may be attached.

As a floppy disk drive is found, its DDB is stored in one of the first two addresses. Each floppy Drive Data Block is built without accessing the drive, using default parameters, since the floppy drives are common to each system, have only one partition, and don't have bad tracks. If there is only one floppy controller in a system, the only difference between the F0 and F1 tables is usually the drive select byte and the disk number, which are set to 0 and 1, respectively.

As Winchesters are found installed (no read error), then W$XDIT determines if the header data is actually PDOS Winchester standard information. The test for this is that the first four bytes of the header information are "ME4U" and the next word, signifying the number of heads on the drive, is from one through 16. If it is okay, then the data is moved into a DDB in system parameter RAM and the address is saved in the next available P$PARM table location. If the drive is installed but the header data is not PDOS Winchester standard information, then W$XDIT moves down some default drive data into the DDB in P$PARM.

The four Winchester Drive Data Blocks are filled as W$XDIT finds them in the system, altering the controller number and drive select bytes to match where the drive is found. The first Winchester's Drive Data Block is usually read into the system's parameter RAM area by W$XDIT immediately following the two floppy DDBs. It is referred to as drive "W0", but it may be attached to any controller with any drive select jumper. The Drive Data Block for drive "W1" would follow the "W0" bad track table, and so on. You must be sure that the parameter RAM definition file, xx$PARM:SR, and the system memory map allocate enough room for all the drives that may be installed in the system.

PDOS Disk Numbering

PDOS disk numbers 0 and 1 are reserved for floppy drives; disk numbers 2 and above are for Winchester partitions. These Winchester partitions, numbered 2-99, are biased by one track worth of sectors (e.g. 32, 33, 34, 38, or 64). To access sectors in the first track, or base track, of the partition, you use the PDOS disk number plus 100. For example, reading from disk 102 accesses the unbiased disk 2 sectors. If there are 32 sectors per track, then disk 2, sector 0 accesses the same sector as disk 102, sector 32. All of the disk accesses for disks 2-99 and 102-199 use the bad track table of the corresponding drive to offset requested tracks.

The PDOS Winchester standard also defines a way to access all the sectors on a drive, ignoring the bad track table remapping feature. This is needed by the "verify" process in the xxFRMT utility -- to check all the sectors on a track to find new bad tracks. PDOS disk numbers 200-209 are mapped to the physical sectors of drive W0, numbers 210-219 are mapped to drive W1, and so on. Disk 200, sectors 0 through 65535 (0 to $FFFF) access Winchester drive W0 physical sectors 0 through 65535. Disk 201, sectors 0 through 65535 access Winchester drive W0 physical sectors 65536 through 131071 ($10000 to $1FFFF). This pattern continues until disk 209 maps to sectors $90000 to $9FFFF. This will accommodate drives up to 168 Mbytes, formatted. If larger disks must be accessed, then you must alter the xxBIOSW:SR code so that the xxFRMT utility can verify the entire drive. This could be done by consolidating drives: 200-219 are drive W0, 220-239 are drive W1.

Currently disk numbers and partitions for each drive are defined by the format utility, xxFRMT. The partitions on each drive get consecutive disk numbers, starting at a specified number, and skipping the standard RAM disk number, 8. Normally the first partition on drive W0 is assigned PDOS disk number 2. The first partition on drive S1 would normally be assigned the next PDOS disk number higher than the last disk number on drive W0, etc.

PDOS Disk Layout

The following disk sector listings define the PDOS disk formats including the header sector, directory entries, and data storage.

```
x>MDDUMP
68K PDOS Disk Dump/Alter Utility
  Disk # = 0
  To alter sector, enter "A"; to exit, enter "Z"
  Start Sector = 0
  End Sector = 2
```

Sector 0 contains the directory header and first sector bit map.

- 0940 = Boot sector
- 006D = # of files
- 88 = # of boot sectors
- 00800 = Boot address
- 0940 = # of PDOS sectors
- A55A = PDOS ID
- FFFF = Sides/density
- 1 = Allocated
- 0 = Free

```
Sector/Disk=$0000 (0)/0
000-00F   53 41 47 45 20 50 44 4F 53 20 32 2E 36 64 00 00 FORCE PDOS 3.3..
             Disk name
010-01F   09 40 00 6D 88 00 08 00 00 80 09 40 A5 5A FF FF .@.m.......@%Z..
020-02F   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
030-03F   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
040-04F   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
050-05F   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
060-06F   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
070-07F   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
080-08F   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
090-09F   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0A0-0AF   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0B0-0BF   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0C0-0CF   FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ................
0D0-0DF   FF F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .p..............
0E0-0EF   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0F0-0FF   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

..............
```

Sector 2 is the first directory sector.

- 41....00 = File name
- 00 00 00 = File extension
- 05 = Directory level
- 0800 = Type
- 0012 = Start sector
- 0000 = Free
- 0012 = Sectors allocated
- 0012 = EOF sector index
- 009A = # of bytes in last sector
- 101FA8A2 = Date created
- 101FA8A2 = Date last updated

```
Sector/Disk=$0002 (2)/0
 000-00F   41 4D 41 5A 49 4E 47 00 00 00 00 05 08 00 00 12 AMAZING.........
 010-01F   00 00 00 12 00 12 00 9A 10 1F A8 A2 10 1F A8 A2 ..........("..("
 020-02F   41 53 4D 00 00 00 00 00 00 00 00 00 80 00 00 25 ASM............%
 030-03F   00 00 00 00 00 00 00 2E 10 1F A8 A2 10 1F A8 A2 ..........("..("
 040-04F   42 30 31 00 00 00 00 00 00 00 00 0A 20 00 00 26 B01.......... ..&
 050-05F   00 00 00 01 00 01 00 58 10 1F A8 A2 10 1F A8 A2 .......X..("..("
 060-06F   42 30 31 00 00 00 00 00 53 52 00 0A 02 00 00 28 B01.....SR.....(
 070-07F   00 00 00 04 00 04 00 55 10 1F A8 A2 10 1F A8 A2 .......U..("..("
 080-08F   42 30 32 00 00 00 00 00 00 00 00 0A 20 00 00 2D B02.......... ..-
 090-09F   00 00 00 01 00 01 00 5B 10 1F A8 A2 10 1F A8 A2 .......[..("..("
 0A0-0AF   42 30 32 00 00 00 00 00 53 52 00 0A 02 00 00 2F B02.....SR...../
 0B0-0BF   00 00 00 04 00 04 00 3D 10 1F A8 A2 10 1F A8 A2 .......=..("..("
 0C0-0CF   42 30 33 00 00 00 00 00 00 00 00 0A 20 00 00 34 B03.......... ..4
 0D0-0DF   00 00 00 01 00 01 00 5B 10 1F A8 A2 10 1F A8 A2 .......[..("..("
 0E0-0EF   42 30 33 00 00 00 00 00 53 52 00 0A 02 00 00 36 B03.....SR.....6
 0F0-0FF   00 00 00 04 00 04 00 3F 10 1F A8 A2 10 1F A8 A2 .......?..("..("
```

- 0013 = Forward link
- 0000 = Backward link (null)

```
To alter sector, enter "A"; to exit, enter "Z"
  Start Sector = $12
  End Sector = $13

 Sector/Disk=$0012 (18)/0
 000-00F   00 13 00 00 FF FF FF FF 00 00 0D 0E 00 00 04 DC ................\
 010-01F   00 00 00 54 00 00 00 68 23 14 41 4D 41 5A 49 4E ...T...h#.AMAZIN
 020-02F   47 20 50 52 4F 47 52 41 4D 00 00 00 1C 14 53 45 G PROGRAM.....SE
 030-03F   45 44 3D 00 0B 63 07 1A 63 5C 00 2E 07 08 5C 0D ED=..c..c\....\.
 040-04F   17 4E 06 63 00 00 08 63 06 5C 0D 17 4E 00 1C 14 .N.c...c.\..N...
 050-05F   57 48 41 54 20 41 52 45 20 59 4F 55 52 20 57 49 WHAT ARE YOUR WI
 060-06F   44 54 48 20 41 4E 44 20 4C 45 4E 47 54 48 00 0A DTH AND LENGTH..
 070-07F   64 0A 65 00 23 14 50 4C 45 41 53 45 20 57 41 49 d.e.#.PLEASE WAI
 080-08F   54 2E 2E 2E 2E 00 0B 00 10 64 5C 01 30 65 5C 01 T.........d\.0e\.
 090-09F   30 18 66 0A 64 5C 01 30 65 5C 01 30 18 67 0A 64 0.f.d\.0e\.0.g.d
 0A0-0AF   65 32 17 68 00 00 08 69 06 5C 00 07 08 6A 06 5C e2.h...i.\...j.\
 0B0-0BF   00 07 08 6B 06 60 64 32 5C 01 30 17 40 00 08 6C ...k.`d2\.0.@..l
 0C0-0CF   06 5C 01 07 08 6B 5C 01 18 67 06 6C 07 08 6C 06 .\...k\..g.l..l.
 0D0-0DF   6C 5C 01 30 07 08 6D 06 6B 07 08 6E 06 5C 01 00 l\.0..m.k..n.\..
 0E0-0EF   06 6F 06 5C 01 01 64 07 06 70 06 5C 01 01 65 00 .o.\..d..p.\..e.
 0F0-0FF   08 6F 70 18 66 06 5C 01 00 00 1F 70 07 1F 6F 00 .op.f.\....p..o.
```

- 0014 = Forward link
- 0012 = Backward link (null)

```
Sector/Disk=$0013 (19)/0
000-00F   00 14 00 12 01 5D 00 00 01 04 00 00 1D 71 00 00 .....]........q..
010-01F   1A 6D 64 2E 07 08 6D 06 6D 5C 01 30 07 01 5D 00 .md...m.m\.0..].
020-02F   00 00 FA 00 08 6D 06 5C 01 07 08 6E 06 6E 5C 01 ..z..m.\...n.n\.
030-03F   30 07 1A 6E 65 2C 07 08 6E 06 5C 01 00 00 1A 6D 0..ne,..n.\....m
040-04F   6E 18 67 5C 00 29 07 01 5D 00 00 00 C8 00 1A 6D n.g\.)..]...H..m
050-05F   5C 01 31 5C 00 29 07 01 5D 00 00 02 12 00 1A 6D \.1\.)..]......m
060-06F   5C 01 31 6E 18 67 5C 00 2E 07 01 5D 00 00 02 12 \.1n.g\....]....
070-07F   00 00 1A 6E 5C 01 31 5C 00 29 07 01 5D 00 00 01 ...n\.1\.)..]...
080-08F   86 00 1A 6D 6E 5C 01 31 18 67 5C 00 2E 07 01 5D ...mn\.1.g\....]
090-09F   00 00 01 86 00 00 1A 6D 64 29 07 01 5D 00 00 01 .......md)..]...
0A0-0AF   4A 00 1A 6D 5C 01 30 6E 18 67 5C 00 2E 07 01 5D J..m\.0n.g\....]
0B0-0BF   00 00 01 4A 00 00 21 60 5C 03 32 5C 01 30 07 01 ...J..!`\.2\.0..
0C0-0CF   5D 00 00 03 16 0A 5D 00 00 03 34 0A 5D 00 00 03 ]......]...4.]...
0D0-0DF   5C 00 1A 6E 65 2E 07 01 5D 00 00 01 54 00 1A 6A \..ne...]...T..j
0E0-0EF   5C 01 29 07 01 5D 00 00 01 72 00 00 08 69 06 5C \.)..]...r...i.\
0F0-0FF   01 07 01 5D 00 00 01 68 00 00 1A 6D 6E 5C 01 30 ...]...h...mn\.0
 To alter sector, enter "A"; to exit, enter "Z"
 Start Sector =
```

PDOS I/O Drivers

PDOS I/O drivers are an extension of the PDOS file system. If a file's attribute is "DR", then the PDOS file manager expects the file to be an I/O driver program instead of data.

Driver Entry Points

PDOS I/O drivers are an extension of the PDOS file system. An I/O driver is designated by the "DR" file type. I/O driver files contain position independent (self-relocating) code rather than data.

When an I/O driver is opened, closed, read from, written to, or positioned, the PDOS file manager branches into the channel buffer at specific entry points. This requires that the first twelve bytes of the file be reserved for branch instructions and that the driver code and variables be no more than 240 bytes in length to be dynamically loaded. Otherwise, the driver must first be installed into the system via the INSTALL utility. The following driver entry points must be at the beginning of each driver module:

```
        SECTION 0
        DC.W    $A55B           ;DRIVER ID
DROP    BRA.S   OPEN            ; 2 OPEN
DRCL    BRA.S   CLOS            ; 4 CLOSE
DRRD    BRA.S   READ            ; 6 READ
DRWR    BRA.S   WRIT            ; 8 WRITE
DRPS    BRS.S   POST            ;10 POSITION
```

The driver must be written in position independent or self-relocating 68000 assembly code. This simply means that while the code is relocatable, there can be no relocatable tags within the object file.

```
        SECTION 0
DTTX    DC.W    $A55B       ;BEGINNING OF DRIVER
        ....

        ADDQ.W   #1,CNT(A2) ;INCREMENT COUNT
        LEA.L   BUF(A2),A0  ;POINT TO BUFFER
        MOVE.L  A0,VAR(A2)
        ....

VAR      EQU *-DTTX+4
        DC.L    0
        ....

         OFFSET *-DTTX+4
CNT     DC.W    0
BUF      DS.B   10
```

A common way to make the code self-relocating is to generate a base address and then reference each constant within the program as a displacement beyond the base address. PDOS passes the base address of the driver buffer in address register A2. This can be conveniently used as the base register for variables defined as the label minus the start address plus four. The former makes the label absolute (relocatable-relocatable=absolute) and the latter skips the file links.

Using Driver Registers

The PDOS file manager passes all parameters in registers to I/O drivers. All registers are available for use by the driver except address registers A4 through A7.

The driver executes in supervisor mode. The return address is already on the system stack. The status register passes the error conditions back to the PDOS file manager. An "EQ" status indicates that no error occurred. A "NE" status specifies an error with the error number returned in data register D0.

The data and address registers of the file manager call are located on the stack immediately following the return address, where D0 is 4(A7), D1 is 8(A7), and so on. This is useful for passing the number of bytes on the end of file to the D3.L of the file manager call. See the input driver example.

If the driver alters constants within the buffer, then the file altered bit must be set in the file slot so that the buffer is correctly restored when rolled to the disk. This is done by executing the instruction "ORI.W #$8000,12(A4)" or "TAS.B 12 (A4)".

The following table describes the register usage for each driver entry point:

OPEN: D7.W = Channel status
(A2) = Driver base + 4
(A4) = File slot
(A5) = SYSRAM
(A6) = Task TCB
(A7) = Return address

CLOSE: D7.W = Channel status
(A2) = Driver base + 4
(A4) = File slot
(A5) = SYSRAM
(A6) = Task TCB
(A7) = Return address

READ: D5.L = Character count (-1 = Line operation)
D7.W = Channel status
(A2) = Driver base + 4
(A3) = Memory buffer
(A4) = File slot
(A5) = SYSRAM
(A6) = Task TCB
(A7) = Return address
3*4+4 (A7) = Return EOF bytes to D3.L

WRITE: D5.L = Character count (-1 = Line operation)
D7.W = Channel status
(A2) = Driver base + 4
(A3) = Memory buffer
(A4) = File slot
(A5) = SYSRAM
(A6) = Task TCB
(A7) = Return address

POSITION: D5.L = Character position
D7.W = Channel status
(A2) = Driver base + 4
(A4) = File slot
(A5) = SYSRAM
(A6) = Task TCB
(A7) = Return address

Driver Generation

A PDOS driver is generated using conventional PDOS utilities. The procedure is as follows:

- Assemble the source file using MASM assembler.
- Change the old driver file type to "SY" (if defined).
- Use the MSYFL utility to create a binary image. If the section 0 length (E tag) exceeds $00FC, the driver must be installed before it may be used.
- Set the new driver file type as "DR".

Example:

```
>MASM TTO:SR,#TTO:RB
68K PDOS Assembler R3.3
ERII, Copyright 1987
SRC=TTO:SR
OBJ=TTO:RB
LST=
ERR=
XRF=
END OF PASS 1
END OF PASS 2
>SA TTO,SY
>MSYFL TTO:RB,TTO
68K PDOS SY File Maker Utility
  Source file = TTO:RB
  Destination File = TTO
   SECTION LENGTH = E0000000CA
    Entry Address = 00000000
>SA TTO,DR
>CF LIST,TTO
```

Restrictions

Note the following restrictions when adding an I/O driver to PDOS:

- Drivers must be written in self-relocating, address independent 68000 assembly language.
- The driver identification constant $A55B must be the first word of the driver.
- Driver entry points must immediately follow the driver identification word.
- A driver MUST NOT make any console or file I/O system calls.
- A driver is exited via an "RTS" instruction. A "NE" status condition indicates a driver error with data register D0 passing the error number.
- Drivers execute in supervisor mode.
- Address registers A4, A5, A6, and A7 must be preserved.
- Drivers longer than 252 bytes must be installed before they may be used.
- Drivers to be ROMed must not use destructive code modification techniques. In addition, ROMable program sections should reside in section 0 or 14. RAM data should use section 1.

PDOS Output Driver Example

The following program is an example of a PDOS I/O driver. The output is to the logical port number found in the TCB variable U1P$. This driver may be optionally installed, but not ROMed.

```
TTO:SR - 68K PDOS TTO DRIVER            68020 PDOS Assembler 10-Dec-86
PAGE: 1         14:44 17-Dec-86         FILE: TTO:SR,WINI #2


 2                              *    TTO:SR         10/02/87
 3
*********************************************************************************
 4                              *                                                 *
 5                              *          66   888  K   K    PPPP  DDDD   OOO   SSS      *
 6                              *         6    8   8 K  K     P   P  D  D O   O S   S     *
 7                              *        6     8   8 K K      P   P  D  D O   O S         *
 8                              *        6666   888  KK       PPPP   D  D O   O  SSS      *
 9                              *        6   6 8   8 K K      P      D  D O   O     S     *
10                              *        6   6 8   8 K  K     P      D  D O   O S   S     *
11                              *         666   888  K   K    P      DDDD   OOO   SSS      *
12                              *                                                 *
13                              *    TTTTT TTTTT  OOO     DDDD  RRRR  III V   V EEEEE RRRR *
14                              *      T     T   O   O     D  D R   R  I  V   V E     R   R *
15                              *      T     T   O   O     D  D R   R  I  V   V E     R   R *
16                              *      T     T   O   O     D  D RRRR   I   V V  EEEE  RRRR *
17                              *      T     T   O   O     D  D R R    I   V V  E     R R  *
18                              *      T     T   O   O     D  D R  R   I    V   E     R  R *
19                              *      T     T    OOO      DDDD  R   R III   V   EEEEE R  R *
20                              *                                                 *
21
*=*****************************************************************************
22                              *    Eyring Research Inst.  Copyright 1983,1987.
23                              *    ALL RIGHTS RESERVED.
24                              *=
25                              *=        Module Name: TTO
26                              *=             Author: Paul Roper
27                              *=     Revision History:
28                              *=
29                              *=   02/11/86 2.0   Fixed XON/XOFF look before calling put
30                              *=   06/20/86 3.0   Fixed for upper D1.L=output event#-printers
31                              *=   10/01/86 3.3   PDOS 3.3
32  0/00000000:                 TTO  IDNT    3.0      68K PDOS TTO DRIVER
33                              *=
34
*=*****************************************************************************
35                              *
36                              *  This driver will output files to the terminal. It outputs
37                              *  the file data to the Unit 1 Port (U1P$) of the task that
38                              *  opened it. It filters the output stream by ignoring LF,
39                              *  converting Cr characters to Cr LF pairs, keeping an inde
40                              *  pendent column counter and expanding Tab to column positions
41                              *  (multiples of 8), using blanks. BS backspace characters
42                              *  decrement the counter. Output events, XON/XOFF, and DTR
43                              *  line checks are all supported.
44                              *    D5.L = Character count (-1 = Line)
45                              *    D7.W = Channel status
46                              *    (A2) = Driver base + 4
47                              *    (A3) = Memory buffer
48                              *    (A4) = File slot
49                              *    (A5) = SYSRAM
50                              *    (A6) = Task TCB
51                              *    (A7) = Return address
52                              *
53                    00001400       OPT      PDOS,CRE
54                    0000007C BURT  EQU      $007C          ;BIOS UART TBL
55                              *
56  0/00000000:       0/00000000       SECTION 0
```

```
TTO:SR - 68K PDOS TTO DRIVER            68020 PDOS Assembler 10-Dec-86
PAGE: 2         14:44 17-Dec-86         FILE: TTO:SR,WINI #2


 1  0/00000000:A55B             DTTO DC.W    $A55B          ;DRIVER ID
 2  0/00000002:600E             DROP BRA.S   OPEN           ; 2 OPEN
 3  0/00000004:6050             DRCL BRA.S   CLOS           ; 4 CLOSE
 4  0/00000006:6006             DRRD BRA.S   READ           ; 6 READ
 5  0/00000008:6050             DRWR BRA.S   WRIT           ; 8 WRITE
 6  0/0000000A:7046             DRPS MOVEQ.L  #70,D0        ;10 POSITION ERROR
 7  0/0000000C:4E75                  RTS
 8                              *
 9  0/0000000E:7050             READ MOVEQ.L #80,D0          ;ERROR 80, DRIVER ERROR
10  0/00000010:4E75                  RTS
11                              *
12  0/00000012:006C8000000C     OPEN ORI.W   #$8000,12(A4)  ;FILE ALTERED
13  0/00000018:422A00EA              CLR.B    CCNT(A2)       ;CLEAR COUNTER
14  0/0000001C:4241                   CLR.W  D1              ;D1=PORT #
15  0/0000001E:122E0452              MOVE.B   U1P$(A6),D1    ;D1=PORT #
16  0/00000022:7650                   MOVEQ.L       #80,D3
17  0/00000024:D601                  ADD.B    D1,D3
18  0/00000026:354300E8               MOVE.W  D3,OUTE(A2)     ;D3=OUTPUT EVENT #
19  0/0000002A:16351058              MOVE.B  UTYP.(A5,D1.W),D3 ;D3=UART TYPE
20  0/0000002E:154300EB              MOVE.B   D3,TYPE(A2)     ;SAVE FOR FUTURE
21  0/00000032:D643                  ADD.W    D3,D3           ;POINT TO DSR
22  0/00000034:2055                  MOVEA.L  (A5),A0
23  0/00000036:D0F0301E              ADDA.W   BURT(A0,D3.W),A0
24  0/0000003A:5448                  ADDQ.W   #2,A0           ;A0=PUTC ENTRY
25  0/0000003C:254800D0              MOVE.L   A0,PUTC(A2)     ;SAVE PUTC ADR
26  0/00000040:E549                  LSL.W    #2,D1           ;SAVE BASE ADR
27  0/00000042:41ED0158              LEA.L    UART.(A5),A0
28  0/00000046:2570100000E0          MOVE.L   0(A0,D1.W),PADR(A2)
29  0/0000004C:E449                   LSR.W  #2,D1           ;SAVE FLAGS
30  0/0000004E:48751048              PEA.l    F8BT.(A5,D1.W) ;PUSH POINTER TO FLAGS
31  0/00000052:255F00E4              MOVE.L   (A7)+,FADR(A2) ;SAVE PTR
32                              *
33  0/00000056:4240             CLOS CLR.W   D0              ;RETURN .EQ.
34  0/00000058:4E75                  RTS
35                              *



36                              ****************************************
37                              *    WRITE CHARACTERS
38                              *
39  0/0000005A:006C8000000C     WRIT ORI.W    #$8000,12(A4) ;N, ALTERED
40                              *
41  0/00000060:7000             WRIT02       MOVEQ.L #0,D0   ;GET CHARACTER
42  0/00000062:101B                  MOVE.B   (A3)+,D0       ;DONE?
43  0/00000064:6604                    BNE.S  WRIT04         ;N
44  0/00000066:4A85                  TST.L    D5             ;Y, WRITE LINE?
45  0/00000068:6BEC                    BMI.S CLOS            ;Y, DONE
46                              *
47  0/0000006A:0C000008         WRIT04       CMPI.B  #$08,D0  ;BACKSPACE?
48  0/0000006E:6604                    BNE.S  WRIT06         ;N
49  0/00000070:532A00EA              SUBQ.B   #1,CCNT(A2)    ;Y
50                              *
51  0/00000074:0C000009         WRIT06       CMPI.B  #$09,D0 ;OK, TAB?
52  0/00000078:6614                    BNE.S WRIT08          ;N
53  0/0000007A:7020                  MOVEQ.L #' ',D0         ;Y
54  0/0000007C:7207                  MOVEQ.L #7,D1           ;GET MASK
55  0/0000007E:C22A00EA              AND.B    CCNT(A2),D1    ;GET COUNTER
56  0/00000082:5F01                  SUBQ.B  #7,D1           ;TAB BOUNDARY?
```

```
TTO:SR - 68K PDOS TTO DRIVER            68020 PDOS Assembler 10-Dec-86
PAGE: 3         14:44 17-Dec-86         FILE: TTO:SR,WINI #2


 1  0/00000084:6708                    BEQ.S WRIT08          ;Y
 2  0/00000086:534B                  SUBQ.W  #1,A3           ;N, DO AGAIN
 3  0/00000088:4A85                  TST.L   D5              ;WRITE LINE?
 4  0/0000008A:6B02                    BMI.S  WRIT08         ;Y
 5  0/0000008C:5285                  ADDQ.L   #1,D5          ;N, BACKUP
 6                                 *
 7  0/0000008E:0C00000A            WRIT08      CMPI.B  #$0A,D0  ;LF?
 8  0/00000092:6742                    BEQ.S WRIT16          ;Y, IGNORE
 9  0/00000094:0C00000D              CMPI.B  #$0D,D0         ;N, CR?
10  0/00000098:6608                    BNE.S WRIT10          ;N
11  0/0000009A:422A00EA              CLR.B   CCNT(A2)        ;Y, CLEAR CCNT
12  0/0000009E:303C0A0D              MOVE.W   #$0A0D,D0      ;CHANGE TO CRLF
13                                 *
14  0/000000A2:0C000020            WRIT10      CMPI.B  #' ',D0  ;CONTROL?
15  0/000000A6:6D04                    BLT.S  WRIT12         ;Y
16  0/000000A8:522A00EA              ADDQ.B   #1,CCNT(A2)    ;N, UP COUNT
17                                 *
18  0/000000AC:4A2A00EB            WRIT12      TST.B   TYPE(A2) ;DEFINED TYPE?
19  0/000000B0:67A4                    BEQ.S  CLOS           ;N, SKIP IT
20  0/000000B2:222A00E8               MOVE.L OUTE(A2),D1     ;GET OUT EFVENT TO D1
21  0/000000B6:206A00E4              MOVEA.L FADR(A2),A0     ;GET PTR TO FLGS
22  0/000000BA:1210                  MOVE.B   (A0),D1        ;TEST FLAG EACH TIME
23  0/000000BC:08010000               BTST.L #0,D1           ;^S^Q CHECK?
24  0/000000C0:6704                    BEQ.S WRIT14          ;N
25  0/000000C2:4A01                  TST.B   D1              ;Y, ^S STOP SET?
26  0/000000C4:6BE6                    BMI.S  WRIT12         ;Y, WAIT HERE
27                                 *
28  0/000000C6:206A00E0            WRIT14      MOVEA.L  PADR(A2),A0    ;UART BASE ADR
29  0/000000CA:4EB900000000          DC.W     $4EB9,0,0      ;JSR PUTC.L
30                     000000D0 PUTC  EQU    *-DTTO          ;RETRY?
31  0/000000D0:66DA                    BNE.S WRIT12          ;Y
32  0/000000D2:E048                   LSR.W  #8,D0           ;N, 2 CHARS?
33  0/000000D4:66D6                    BNE.S WRIT12          ;Y
34                                 *
35  0/000000D6:5385                WRIT16      SUBQ.L   #1,D5  ;DONE?
36  0/000000D8:6686                    BNE.S  WRIT02         ;N
37                                 *    BRA      CLOS2        ;Y
38  0/000000DA:4E75                  RTS                     ;Y, RETURN .EQ.
39                                 *

40                                 ****************************************
41                                 *    DRIVER VARIABLES
42                                 *
43  0/000000DC:         000000E0       OFFSET   *-DTTO+4
44    000000E0:00000000            PADR DC.L    0             ;BASE ADR
45    000000E4:00000000            FADR DC.L    0             ;UART FLAGS ADDRESS
46    000000E8:0000                OUTE DC.W     0            ;OUTPUT EVENT #
47    000000EA:00                  CCNT DC.B    0             ;COLUMN COUNT
48    000000EB:00                  TYPE DC.B    0             ;PORT TYPE
49    000000EC:                        EVEN
50                                 *
51                                 ****************************************
52                                 *    DRIVER LENGTH CHECK
53                                 *
54    000000EC:                        IFLT     256-(TYPE+1)
55                                     FAIL     ** DRIVER LENGTH ERROR! **
56                                      ENDC
57    000000EC:       0/00000000       END      DTTO
```

PDOS Input Driver Example

```
TTI:SR - 68K PDOS TTI DRIVER            68020 PDOS Assembler 10-Dec-86
PAGE: 1         14:45 17-Dec-86         FILE: TTI:SR,WINI #2


 2                  *  TTI:SR        10/02/87
 3
*************************************************************************
 4                  *                                                     *
 5                  *        66   888  K   K    PPPP  DDDD   OOO   SSS    *
 6                  *       6    8   8 K  K     P   P  D  D O   O S   S   *
 7                  *      6     8   8 K K      P   P  D  D O   O S       *
 8                  *      6666   888  KK       PPPP   D  D O   O  SSS    *
 9                  *      6   6 8   8 K K      P      D  D O   O     S   *
10                  *      6   6 8   8 K  K     P      D  D O   O S   S   *
11                  *       666   888  K   K    P     DDDD   OOO   SSS    *
12                  *                                                     *
13                  *  TTTTT TTTTT III    DDDD  RRRR  III V   V EEEEE RRRR  *
14                  *    T     T    I      D  D R   R  I  V   V E     R   R  *
15                  *    T     T    I      D  D R   R  I  V   V E     R   R  *
16                  *    T     T    I      D  D RRRR   I   V V  EEEE  RRRR  *
17                  *    T     T    I      D  D R R    I   V V  E     R R   *
18                  *    T     T    I      D  D R  R   I    V   E     R  R  *
19                  *    T     T   III    DDDD  R   R III   V   EEEEE R   R  *
20                  *                                                     *
21
*=*********************************************************************
22                  * Eyring Research Inst. Copyright 1983,1987.
23                  * ALL RIGHTS RESERVED.
24                  *=
25                  *=            Module Name: TTI
26                  *=                 Author: Richard Adams
27                  *=       Revision History:
28                  *=
29                  *=       10/03/86 3.0    Initial release
30                  *=       10/02/87 3.3   PDOS 3.3
31  0/00000000:     TTI      IDNT    3.3     68K PDOS TTI DRIVER
32                  *=
33
*=*********************************************************************
34                  *
35                  * This driver will input files from the terminal. It gets
36                  * characters from the input port (PRT$) of the task that opened it,
37                  * stores them in the buffer (A3), and echoes them to active ouput
38                  * port(s). It supports both XRLF read line and XRBF read block
39                  * primitives. OPEN call simply makes sure that there is an input
40                  * port assigned to the task. Close does nothing. EOF errors are
41                  * returned, along with the byte count, if an escape is entered.
42                  *
43                  * D5.L = Character count (-1 = Line)
44                  * D7.W = Channel status
45                  * (A2) = Driver base + 4
46                  * (A3) = Memory buffer
47                  * (A4) = File slot
48                  * (A5) = SYSRAM
49                  * (A6) = Task TCB
50                  * (A7) = Return address
51                             *
52                     00001000      OPT     PDOS
53                             *
54  0/00000000:      0/00000000       SECTION       0
55  0/00000000:A55B              DTTI DC.W     $A55B          ;DRIVER ID
56  0/00000002:600E              DROP BRA.S    OPEN           ; 2 OPEN
```

```
TTI:SR - 68K PDOS TTI DRIVER          68020 PDOS Assembler 10-Dec-86
PAGE: 2         14:45 17-Dec-86       FILE: TTI:SR,WINI #2


 1  0/00000004:6012            DRCL  BRA.S  CLOS          ; 4 CLOSE
 2  0/00000006:6014            DRRD BRA.S   READ          ; 6 READ
 3  0/00000008:6004            DRWR BRA.S   WRIT          ; 8 WRITE
 4  0/0000000A:7046            DRPS MOVEQ.L  #70,D0       ;10 POSITION ERROR
 5  0/0000000C:4E75                 RTS
 6                             *
 7  0/0000000E:7050            WRIT MOVEQ.L #80,D0        ;ERROR 80, DRIVER ERROR
 8  0/00000010:4E75                 RTS
 9                             *
10  0/00000012:4A2E044F        OPEN TST.B    PRT$(A6)     ;IS THERE INPUT PORT?
11  0/00000016:67F6                BEQ.S WRIT            ;N, SEND ERROR 80
12                             *
13  0/00000018:4240            CLOS  CLR.W  D0            ;RETURN .EQ.
14  0/0000001A:4E75                 RTS
15                             *
16                             ****************************************
17                             *    READ CHARACTERS, BLOCK OR LINE
18                             *
19  0/0000001C:7200            READ  MOVEQ.L       #0,D1          ;GET COUNT, EOF FOR ESCAPE
20                             *
21                             *     DO LINE/BLOCK READ
22                             *
23  0/0000001E:A07A            LINE XGCR                  ;GET A CHARACTER
24  0/00000020:6D1E                BLT.S ESC             ;ESCAPE OUT
25  0/00000022:4A85                TST.L   D5            ;LINE?
26  0/00000024:6A0A                BPL.S @010            ;N, SKIP Cr CHECK
27  0/00000026:0C00000D            CMPI.B   #13,D0       ;Y, CR?
28  0/0000002A:6604                BNE.S @010            ;N, ECHO AND STORE
29  0/0000002C:4213                CLR.B   (A3)          ;Y, TERMINATE LINE
30  0/0000002E:60E8                BRA     CLOS          ;GET BAT OUT
31                             *
32  0/00000030:A086            @010 XPCC                  ;ECHO TO SCREEN
33  0/00000032:16C0                MOVE.B  D0,(A3)+      ;SAVE IN BUFFER
34  0/00000034:5281                ADDQ.L  #1,D1         ;UP COUNT
35  0/00000036:4A85                TST.L   D5            ;LINE?
36  0/00000038:6BE4                BMI.S LINE            ;Y, SKIP COUNT CHECK
37  0/0000003A:B285                CMP.L    D5,D1        ;N, DONE BLOCK COUNT?
38  0/0000003C:6DE0                BLT.S LINE            ;N, GET ANOTHER
39  0/0000003E:60D8                BRA.S   CLOS          ;Y, RETURN .EQ.
40                             *
41  0/00000040:2F410010        ESC  MOVE.L  D1,3*4+4(A7)    ;RETURN COUNT IN OLD D3
42  0/00000044:7038                 MOVEQ.L       #56,D0  ;EOF ERROR RETURN
43  0/00000046:4E75                 RTS
44                             *
45                             ****************************************
46                             *    DRIVER LENGTH CHECK
47                             *
48  0/00000048:                     IFLT     256-(*-DTTI+4)
49                                  FAIL    ** DRIVER LENGTH ERROR! **
50                                  ENDC
51                             *
52  0/00000048:      0/00000000     END     DTTI
```

Installable Device Routines and Utilities

Installable device routines and utilities include UART service routines, disk service routines, file I/O drivers and utilities. These four service routine types may all be installed into the operating system via the PDOS INSTALL utility which is fully described in the *Monitor, Editor, Utilities* manual. You should also reference LINKIN, a system facility which combines installable routines with the PDOS operating system or for ROM.

Programming Conventions

Installable code should use the following conventions:

- Use proper sections.

Section 0	Utility/IO driver ROMable application code
Section 1	RAM data
Sections 2-13	User defined (may be ROM, or RAM)
Section 14	Disk/UART ROMable code
Section 15	Reserved for PDOS

  All data areas should be located in Section 1. A common programming practice under early PDOS versions was to locate the data area after the program space via an OFFSET assembler directive. This should not be done on installed routines.
- Make the code shareable.
  Utilities should be re-entrant and shareable.
- Use the PDOS kernel dispatch table.
  The kernel dispatch table should be used by disk and UART service routines to set up interrupt vectors and other kernel services on the BIOS level.

UART Service Routines

PDOS can handle up to eight UART types. Each type has a table of short branches to the various subroutines for get, put, baud, etc. A pointer to the BRA.S tables for each UART type is located in the BIOS table. If a certain UART type is not used in a system, then its pointer addresses a NULL UART table of routines located in MBIOS:SR, which return .NE. status for all calls. To add another UART type, the INSTALL facility first loads the object code into memory, preserves that memory, calls the initialization routine for the card, reports the number of cards found, and then alters the BIOS table pointer to address the new DSR table.

The UART type code can be written in either assembly or C, and must contain a special structure and initialization code. The object must contain an entry tag that points to the DSR table, or if the file is an "SY" type, the DSR table must be at the very beginning of the code. The DSR table has the same entries as the standard PDOS UART type, with the following additions. The data word just after the DSR table must contain the characters "U0" (U zero), the word just after that must have a BRA.S INIT branch to the card initialize routine. Also, INSTALL assumes that just after the initialize call there is a string, null terminated, which describes the UART type:

```
*****************************************
*       MAIN DISPATCH TABLE:
*
ACMESIO BRA.S   UnDG    ;GET A CHARACTER
        BRA.S   UnDP    ;PUT A CHARACTER
        BRA.S   UnDB    ;BAUD THE PORT
        BRA.S   UnDR    ;RESET THE PORT
        BRA.S   UnDS    ;READ PORT STATUS
        BRA.S   UnHW    ;HIGH WATER
        BRA.S   UnLW    ;LOW WATER
        DC.W    'UO'    ;INSTALL ID
        BRA.S   UnDI    ;INIT
        DC.B    'ACME SIO',0 ;IDENT MESSAGE
        EVEN
        . . .
*
        END     ACMESIO
```

The INIT call is made by INSTALL in supervisor mode. It should count the number of cards using bus errors, perform any one-time initialization of the cards or chips, setup UART interrupt vectors, and return the number of cards found in the system. INIT has the following inputs and outputs:

```
*****************************************
*         INSTALL DRIVER:
*           IN:   (A1) = K1$BEGN > KERNEL DISPATCH TABLE
*                 (A2) = 0 or NEW BASE ADDR
*           OUT:  D0.B = # of cards found
*
*         PRESERVE D1/A4
```

Address register A1 points to the PDOS kernel dispatch table. This facilitates calling the K1$SVEC subroutine to either set the bus error vector when counting the number of cards, or to set the interrupt service routines of the UARTs. The method of calling K1$SVEC is as follows:

```
. . .
MOVEQ.L #2,D0              ;SET A NEW BUS ERROR ROUTINE
LEA.L   @020(PC),A0        ;CONTINUE ADDR
JSR     $20(A1)            ;SET NEW BUS ERROR (A0= OLD BSER)
MOVE.L  A7,D2              ;SAVE OLD STACK
. . .
```

The UART code should contain no PDOS primitives so that it can be linked in at xxDOS:GEN time, if necessary. If you need to write a new installable UART file, carefully examine one of the sample files. Once installed, new UART types cannot be deleted, but you must reboot the system.

I/O Drivers

PDOS maintains a linked list of installed I/O drivers, with the pointer to the first entry in the SYRAM variable DRVL.(A5). Each entry in the list consists of a long word address to the next entry, a 16-bit word containing the size of the driver in number of kbytes, the 32-byte directory entry of the original driver file, followed by the code portion of the driver, without the sector links at the beginning. A zero long word in the pointer to the next list entry indicates the end of the list. A zero in the size word tells INSTALL that the driver was linked at DOSGEN time and cannot be removed.

I/O drivers have been described previously.

Disk Service Routines

PDOS communicates with mass storage drives through four routines:

- Initialize (W$XDIT) which is only called at cold startup,
- Disk Off (W$XDOF) which is called once a second while PDOS is running,
- Read Sector (W$XRSE) and
- Write Sector (W$XWSE).

The routines are accessed by PDOS through four pointers in the BIOS table.

To add another disk controller, the INSTALL facility first loads the object code into memory, preserves that memory, calls the initialization routine for the disk controller, reports the number of cards found, and then alters the BIOS table pointers for W$XDIT, W$XRSE, and W$XWSE to address the new routines contained in the added code. These installable disk files have a specific structure that helps INSTALL to manage them. The disk off routine first performs the timer functions associated with its card, and then jumps to the former BIOS routine address. The read and write sector routines first check to see whether or not the disk number of the call corresponds to one of theirs. If so, then the operation is performed on the new card followed by an RTS. If not, then the new routine simply jumps to the address of the former BIOS routine. The following excerpt from the sample disk file WSAMPLE:SR shows the required call structure:

```
ENTRY   DC.W    'W0'    ;IDENTIFIER
        BRA.S   WINIT   ;INIT DISK
        BSR.L   XDOF    ;DISK OFF
        JMP     $0.L    ;OLD DISK OFF ROUTINE
        BSR.L   XREAD   ;READ SECTOR
        JMP     $0.L
        BSR.L   XWRIT   ;WRITE SECTOR
        JMP     $0.L
        DC.B    'W Sample',0 ;ID MESSAGE
        EVEN
*
*****************************************
*       INIT DISK
*         IN:   (A1) = K$1BEGN
*               (A2) = optional card base address
*                D7  = optional disk #
*         OUT:   D0  = # of cards installed
*
* This code is executed once at installation time.
*
WINIT   XPMC    MES1    ;OUTPUT INIT MESSAGE
        . . .
        END     ENTRY
```

See the description of BIOSW for further information about device service routines.

Shared Utility

PDOS maintains a linked list of installed utilities, with the pointer to the first entry in the SYRAM variable UTLL.(A5). Each entry in the list consists of a long word address to the next entry, a 16-bit word containing the size of the utility in number of kbytes, the 32-byte directory entry of the original file, followed by the code portion of the utility. A zero long word in the pointer to the next list entry indicates the end of the list. A zero in the size word informs INSTALL that the utility was linked at DOSGEN time and cannot be removed.

To install a shareable utility, set the first parameter to "X" and the second parameter <filename> to the name of the "SY" or "OB" type file that contains the utility. Any size file can be used. Utilities which have been altered so that they are installable have the characters "X0" (zero) as their second word:

```
START BRA.S    @010
      DC.W     'X0'
      DC.B     '<message>',0
      EVEN
@010  . . .
```

If the utility doesn't have the "X0", INSTALL prints a warning message, but still installs the file into the list. This list is searched by the PDOS Monitor just before it chains to the command with XCHF. Disk numbers on the file are ignored so that any reference to the file name invokes the installed version of the utility. Utilities must be re-entrant.

Interrupts

Interrupt service routines for new cards in a PDOS system are usually added to the xxBIOS:SR file, with a new entry in the BINTB table. Some general rules are:

- Be sure that the interrupt acknowledge daisy-chain is complete across the back plane (for VMEbus only).
- If response time is critical, select an interrupt level that is higher than the system clock interrupt. If it is really critical, select a level higher than the UARTs (you may drop characters).
- Save all registers that you use upon entry, and restore them before exiting.
- Do not assume any registers are set (e.g. A5, A6) or passed from a task.
- The system stack is not infinite. You may be interrupting during another task or during the SWAP routine.
- Avoid wait loops in ISRs.
- Avoid using PDOS primitives in ISRs.
- Preferably just set an event and get out with either an RTE instruction or an XRTE primitive.

The best way to handle device interrupts from a task is to set the device to interrupt and then suspend the task on both a timeout local event and the event (EVNT) associated with the ISR. Be sure the EVNT is reset before suspending on it or you will come right back, before the interrupt. The ISR should set EVNT directly in SYRAM with code similar to the following:

```
        XREF    EVTB.
*
ISR     MOVE.L  A5,-(A7)        ;SAVE REG
        MOVE.L  B$SRAM,A5       ;GET SYRAM POINTER
        BSET     #~EVNT,EVNT/8+EVTB.(A5)  ;SET EVENT IN SYRAM TABLE
        MOVEA.L  (A7)+,A5
        XRTE                    ;RETURN AND SWAP TO TASK
```

The tilde (~) on EVNT simply converts PDOS event numbers (where 0 is most significant) into 68000 bit numbers (where 7 is most significant). Dividing EVNT by 8 yields the byte index of event EVNT in the event bit table of SYRAM. The XRTE primitive executes an RTE instrucion, after setting a flag for the PDOS swap routine to execute a swap as soon as possible.

If faster interrupt response is needed or if some immediate calculation of data is required, then you need to insert more code at the ISR itself. Remember not to block interrupts for too long, unless you must.

PDOS Error Definitions

Only PDOS system errors (50-99) are discussed here. Assembler errors (300-399) and linker errors (500-599) are discussed in their respective manuals. The BIOS errors can be found in the *Installation and Systems Management* guide for your hardware system. Language errors are discussed in the reference manual for each specific language. Errors are returned through data register D0 on all assembly primitives.

PDOS Error Summary

50	Bad File Name
51	File Already Defined
52	File Not Open
53	File Not Defined
54	Bad File Attribute
55	Too Few Contiguous
56	End of File
57	File Directory Full
58	File Writ/Del Prot
59	Bad File Slot
60	File Space Full
61	File Already Open
62	Bad Message Ptr Call
63	Bad Object Tag
64	
65	Not Executable
66	Bad Port/Baud Rate
67	Bad Parameter
68	Not PDOS Disk
69	Out of File Slots
70	Position > EOF
71	AC File Nesting > 2
72	Too Many Tasks
73	Not Enough Memory
74	Non-existent Task
75	File Locked
76	
77	Not Memory Resident
78	Msg Buffer Full
79	Bad Memory Address
80	Bad Driver Call
81	
82	
83	Delay Queue Full
84	
85	Task Abort
86	Suspend on Port 0
87	Exception
88	
89	

90 Illegal K2 module primitive
91 Illegal K3 module primitive
92 Illegal F module primitive
93 Illegal W module primitive
94 Illegal N module primitive
95 Illegal D module primitive
96 Illegal M module primitive
97 Illegal B module primitive
98
99

PDOS Error Ranges

1- 49 BASIC error numbers
50- 99 PDOS system error numbers
100-200 BIOS error numbers (disk)
300-399MASM error numbers
400-499 C error numbers
500-599QLINK error numbers
600-699 Pascal error numbers

PDOS Error Numbers

ERROR 50

Bad File Name. Valid file names consist of an alphabetic character followed by up to 7 alpha-numeric characters. An optional extension and disk number may follow. An extension consists of a colon followed by 1 to 3 characters. A disk number is delineated by a slash and a number ranging from 0 to 127.

```
x>DKDKDKDKF
PDOS ERR 50 Bad File Name
x>
```

ERROR 51

File Already Defined. Each file name is unique to a disk file directory. There is one directory per disk number.

```
x>DF FILE1
x>DF FILE1
PDOS ERR 51 File Already Defined
x>
```

ERROR 52

File Not Open. An attempt to access a file which has not been opened, results in error 52.

```
x>EX
FILE 1,1;3,I
*ERROR 52 File Not Open
```

ERROR 53

File Not Defined. If the file name does not exist in the disk directory, an error 53 occurs.

```
x>SF FILE2
PDOS ERR 53 File Not Defined
x>
```

ERROR 54

Bad File Attribute. Valid file types are AC, BN, OB, SY, BX, EX, TX, DR, *, and **. All others result in error.

```
x>SA FILE1,TR
PDOS ERR 54 Bad File Attribute
x>
```

ERROR 55

Too Few Contiguous. Error 55 results from attempting to define a contiguous file on a disk unit which does not have enough room or is fragmented so that there is not a big enough contiguous block of sectors.

```
x>DF FILE2,10000
PDOS ERR 55 Too Few Contiguous
x>
```

ERROR 56

End of File. Error 56 results from an attempt to read past the end of file index of a file.

```
x>EX
*READY
OPEN "#PAUL",F
FILE 1,F;3,I
*ERROR 56 End of File
```

ERROR 57

File Directory Full. The file directory size is set when the file is initialized. Any attempt to define another file after the directory has been filled results in error 57.

```
x>DF FILE3
PDOS ERR 57 File Directory Full
x>
```

ERROR 58

File Write/Delete Protected. An attempt to delete a file with a delete or write protect flag results in error 58.

```
x>SA TEMP,*
x>DL TEMP
PDOS ERR 58 File Writ/Del Prot
x>
```

ERROR 59

Bad File Slot. A valid file slot number is returned from PDOS on all open commands. A file slot consists of the the disk number in the left byte and the slot index in the right byte.

```
x>EX
*READY
FILE 1,F;3,I
*ERROR 59 Bad File Slot
```

ERROR 60

File Space Full. An attempt to extend a file or define a file after the disk space is filled results in error 60.

```
x>CF TEMP,LIST
PDOS ERR 60 File Space Full
x>
```

ERROR 61

File Already Open. A file can be opened only once in sequential (XSOP) and random (XROP) modes. Read only open (XROO) and shared random open (XNOP) can be executed more than once on the same file.

```
x>EX
*READY
OPEN "LIST",F
OPEN "LIST",F
*ERROR 61 File Already Open
```

ERROR 62

Bad Message Pointer Call.

```
x>ER 62
PDOS ERR 62 Bad Message Ptr Call
x>
```

ERROR 63

Bad Object Tag. Only hex object tag characters are legal.

```
x>SA TEST:SR,OB
x>TEST:SR
PDOS ERR 63 Obj err
x>
```

ERROR 64

ERROR 65

Not Executable. Only section 0 is executable under PDOS.

```
x>TEMP
PDOS ERR 65 Not Executable
```

ERROR 66

Bad Port/Baud Rate. Only numbers 1 through 15 are legal ports. Valid baud rates are 110, 300, 600, 1200, 2400, 4800, 9600, and 19200. The baud rate of 38400 is also available on some systems.

```
x>BP 2,1250
PDOS ERR 66 Bad Port/Baud Rate
x>BP 20,9600
PDOS ERR 66 Bad Port/Baud Rate
x>
```

ERROR 67

Bad Parameter. Most monitor commands check parameters for valid ranges and types.

```
x>LS 1
PDOS ERR 67 Bad Parameter
x>
```

ERROR 68

Not a PDOS Disk. An initialized PDOS disk has the constant >A55A at location >0028 of the header sector (sector 0). If the constant is not found on a disk read, error 68 results.

```
x>LS /2
PDOS ERR 68 Not PDOS Disk
x>
```

ERROR 69

Out of File Slots. A maximum of 32 files can be open at a time. These correspond to the 32 file slots.

```
x>CF TEMP,TEMP1
PDOS ERR 69 Out of File Slots
x>
```

ERROR 70

Position >> Error. Error 70 results from a position command beyond the end of file index.

```
x>EX
*READY
OPEN "#PAUL",F
FILE 1,F;4,0
*ERROR 70 Position >> EOF
```

ERROR 71

AC File Nesting >> 2. Error 71 results for nesting procedure files too deep.

ERROR 72

Too Many Tasks. The task list is defined when the PDOS system is generated.

```
x>@CF LIST,$TTA
PDOS ERR 72 Too Many Tasks
x>
```

ERROR 73

Not Enough Memory. An attempt to create a task with more memory than the current task or available memory in the system memory bit maps results in error 73.

```
CT ,40,,1
PDOS ERR 73 Not Enough Memory
>
```

ERROR 74

Non-existent Task. Error 74 occurs when referencing either a task not in the task list or task 0.

```
x>KT 5
PDOS ERR 74 Non-existent Task
x>
```

ERROR 75

File Locked. Once a file has been locked (XLKF), it cannot be accessed until unlocked (XULF).

```
x>CF FDATA,TEMP
PDOS ERR 75 File Locked
x>
```

ERROR 76

ERROR 77

Not Memory Resident. If PDOS BASIC is not resident in the system, all "BX" and "EX" files will not execute. Also, the interpreter cannot be entered with the "EX" command.

```
x>EX
PDOS ERR 77 Not Memory Resident
```

ERROR 78

Message Buffer Full. There are 32 message buffers in the PDOS system. Too many messages results in error 78.

```
x>SM 4,ANOTHER MESSAGE
PDOS ERR 78 Msg Buffer Full
x>
```

ERROR 79

Bad Memory Address. This error results from a XFUM primitive with invalid arguments.

ERROR 80

Bad Driver. Driver dependent.

ERROR 81

ERROR 82

ERROR 83

Delay Queue Full. Too many delayed events have been requested.

ERROR 84

ERROR 85

Task Abort. If a task is aborted by the scheduler, error 85 results.

ERROR 86

Suspend on Port 0. A task has made a call to get character without any possibility of getting a character.

ERROR 87

Exception.

ERROR 88

ERROR 89

ERROR 90

Illegal K2 Module Primitive. Run module error where a kernel #2 primitive has been executed and the module was not generated in the PDOS system.

ERROR 91

Illegal K3 Module Primitive. Run module error where a kernel #3 primitive has been executed and the module was not generated in the PDOS system.

ERROR 92

Illegal F Module Primitive. Run module error where a file manager primitive has been executed and the module was not generated in the PDOS system.

ERROR 93

Illegal W Module Primitive. Run module error where a R/W module primitive has been executed and the module was not generated in the PDOS system.

ERROR 94

Illegal N Module Primitive. Run module error where a floating point module primitive has been executed and the module was not generated in the PDOS system.

ERROR 95

Illegal D Module Primitive. Run module error where a debugger module primitive has been executed and the module was not generated in the PDOS system.

ERROR 96

Illegal M Module Primitive. Run module error where a monitor module primitive has been executed and the module was not generated in the PDOS system.

ERROR 97

Illegal B Module Primitive. Run module error where a BASIC module primitive has been executed and the module was not generated in the PDOS system.

Index

M

N

O

P

T