GenREF
v1.00

MDOS Reference guide.

MEMORY - CONTENTS

Page

MEMORY MANAGEMENT OVERVIEW

The memory management routines in MDOS are provided to aid a programmer in writing applications which are larger than the 64 Kbytes directly addressable by the CPU's 16 address lines. They also serve the purpose of providing each task with it's own private address space, separate from other the memory accessible to other tasks.

Each task under MDOS can have 2 Mbytes of virtual memory, using 21 address bits. The 21 addresses bits consist of two fields. The first field includes the eight most significant address bits, and is referred to as the virtual page number. The second field consists of the thirteen least significant address bits, and is referred to as the page offset.

The physical memory in the Geneve computer has 21 address lines, for a maximum of 2 MBytes of physical memory. Like virtual memory, each physical address can be thought of as a 21 bit address of two fields, with the first eight bit field called the physical page number, and the final thirteen bit field referred to as the page offset.

NOTE: It can be easy to confuse physical pages with virtual pages, so be careful when reading the opcode descriptions below.

The 16 address lines provided by the 9995 processor can be thought of two fields. The first field is the most significant three bits of the address and is called the "window number". The least significant thirteen bits are the page offset. These 16 address bits can be referred to as the linear address space of the CPU.

The virtual address space and the linear address space are of the most interest to people writing tasks which run under MDOS. The memory management routines provide transparent methods of assigning physical memory pages into your task's virtual address space and transparent methods of viewing any 8k block of a task's virtual address space within one of the seven usable 8k memory windows in the linear address space.

MDOS maintains two arrays to manage the physical memory pages belonging to your task. The first array, which only contains 8 physical page numbers, is part of the Geneve hardware and is called the "mapper". The mapper is used to assign a physical memory page to each of the eight 8k windows addressable in the 9995's linear address space. The second array, which is actually stored as a singly linked list within MDOS, is stored in RAM under control of the MDOS memory management routines and is referred to as "the virtual page list". Each node in the virtual page list consists of a physical page number, and various attributes for that page. Various attributes used in the virtual page list allow for pages to be unassigned (correspond to no useful physical page), for pages to be shared, for pages to be disk-resident (swapped out), and for pages to be private (accessible to only your task.)

CALLING MEMORY FUNCTIONS

The MDOS memory management functions must be called from within a machine code program running as a task under MDOS. You pass arguments to the memory management functions using only a few registers of your program's workspace.

The MDOS memory management functions are invoked from a machine code program when software trap number zero (XOP 0) is called with a library number of 7. The calling program's R0 must contain the opcode of the routine within the memory management library which is to be performed. The following code fragment will allocate memory to your task.

```
        LI      R0,1
        LI      R1,7       56k bytes
        LI      R2,1       starting @>2000
        SETO    R3         try fast pages
        XOP     @SEVEN,0
        MOV     R0,R0
        JNE     MEMERR
* ...
SEVEN   DATA    7
*...
```

AVAILABLE MEMORY

Function You would use this operation in a program when you want to determine how much memory is available for use. It returns the total number of 8k pages installed, the number of zero wait state 8k pages available, and the total number of 8k pages available (both fast and slow pages).

Parameters	R0	= 0 (opcode)
Results	R0	= 0 (no error)
	R1	= total number of free pages
	R2	= number of free zero wait state pages
	R3	= total number of installed pages

ALLOCATE MEMORY

Function This routine allows you to assign physical pages of memory from system list of free pages to virtual pages belonging to your task. You must use this function if you wish to use more memory than your program occupied on disk as a program image file. You must also use this function if you wish to use more than 64k of memory in a program you have written. This routine will not reassign pages which have already been allocated, even if the block of pages you specify overlaps pages which have already been assigned to your task.

On successful return, all pages in the range R2..(R2+R1-1) are available for use by your task.

Parameters	R0	= 1 (opcode)
	R1	= page count
	R2	= starting page
	R3	= speed flag 0 (use first available memory page) <>0 (use zero wait state pages, if available)
Results	R0	= error code
	R1	= new count
	R2	= fast count

Parameter description

Page count This is the number of consecutive memory pages you wish to have for your program, it is not necessarily the number of pages which will be returned to your program. As an example, to allocate 20k bytes of memory for your program, you must actually ask for three 8k memory pages. The number of pages you need to ask for can be calculated from the number of bytes you need as follows:

pages = (bytes + >1fff) / >2000

Starting page This is the virtual page number within your task's memory at which you want to allocate more memory pages. If you were to think of your task's memory as having addresses ranging from >000000 to >1fffff (0 to 2 MB), this number is the address divided by 8192 (remainder is discarded.)

Speed flag If this flag is non-zero, MDOS will attempt to assign zero wait state memory pages to your task. If there are not enough zero wait state pages available to satisfy your request, MDOS will assign slow pages to your task in order to satisfy the request.

If this flag is zero, and your computer has 512k of (one wait state) ram on the motherboard, MDOS will first attempt to assign slow pages to your task. If there are not enough slow pages, MDOS will continue by allocating fast pages to your task.

If this flag is zero, and your computer has 1024k of (zero wait state) ram on the motherboard, MDOS will first attempt to assign fast pages to your task. If there are not enough fast pages, MDOS will continue by allocating slow pages to your task.

The "fast count" returned to you reflects the number of fast pages allocated as a result of the operation, and the "fast count" subtracted from the "new count" returned to you reflects the number of slow pages allocated as a result of the operation.

If the "fast count" returned to you is non-zero, and different from the number of pages you requested, there is no convenient method of determining which pages are fast, and which are slow. The easiest deterministic method of telling which pages are fast and which are slow is to ask MDOS for one page at a time, and look at the "fast count" resulting from each single-page allocation.

Error code

0 = No error. This indicates that the pages you specified can now be used by your task.

1 = Insufficient memory. When you get this error, there were not enough pages free in the system to accommodate your request for more memory. No additional pages have been assigned to your task, even if there were some free memory pages in the system.

(NOTE: Calling the "Available memory" operation to determine the amount of memory available, followed by the "Allocate memory" operation with fewer pages than reported to you from "Available memory" can still fail, since another task may have allocated pages in between your two calls. Do not rely on being able to call the two routines in succession without checking the error code returned from the "Allocate memory" operation.)

7 = Attempt to overwrite shared page. You will get this error if any page in the range R2..(R2+R1-1) is already allocated to your task with a "shared" attribute. No additional pages have been assigned to your task if you receive this error, even if there were enough free memory pages in the system to accommodate your request.

8 = Out of table space. You will receive this error if too many tasks have large gaps of unassigned pages in their memory maps. The current versions of MDOS allow 480 virtual pages between all tasks which are currently executing. Note that there are only 256 possible physical pages, and that there are only 128 physical pages even if you have the 512k expansion ram, so tasks would have to be pretty wasteful (have more gaps than actual pages) in order to use up all 480 virtual pages allowed by MDOS. If you get this error, your program should just give up and tell the user to try later.

New count

This is the number of pages which were newly assigned to your task, and is only valid if you did not receive an error from the "allocate memory" call. This number can be less than the number of pages you requested if some of the pages in the range R2..(R2+R1-1) were already assigned to your task.

Fast count This is the number of fast pages which were newly assigned to your task, and is only valid if you did not receive an error from the "allocate memory" call. You would use this to check if MDOS actually assigned any fast pages to your task.

Example 1.1 **Filling a hole**

R0=1	Opcode
R1=2	Number of pages to get
R2=4	Virtual page number
R3=0	Speed flag

Virtual Page	**Physical Page**	**Physical Page**	
	Before	**After**	
0	>3F	>3F	
1	>3E	>3E	
2	>3D	>3D	
3	(>FF)	(>FF)	hole
4	(>FF)	>33	new page
5	(>FF)	>34	new page
6	(>FF)	(>FF)	hole
7	>3C	>3C	
8	>3B	>3B	
9	>3A	>3A	
10	>39	>39	
11	>38	>38	
12	>37	>37	
13	>36	>36	
14	>35	>35	
15	>34	>34	

NOTE: The pages (>FF) represent holes in the tasks virtual memory map. The physical page >FF is actually part of the boot rom on your computer, and cannot be overwritten by your task.

Example 1.2 **Creating a hole**

R0=1	Opcode
R1=1	Number of pages to get
R2=17	Virtual page number
R3=0	Speed flag

Virtual Page	**Physical Page**	**Physical Page**	
	Before	**After**	
0	>3F	>3F	
1	>3E	>3E	
2	>3D	>3D	
3	(>FF)	(>FF)	hole
4	(>FF)	(>FF)	hole
5	(>FF)	(>FF)	hole
6	(>FF)	(>FF)	hole
7	>3C	>3C	
8	>3B	>3B	
9	>3A	>3A	
10	>39	>39	
11	>38	>38	
12	>37	>37	
13	>36	>36	
14	>35	>35	
15	>34	>34	
16	(null)	(>FF)	new hole
17	(null)	>33	new page

Notice that this routine only **fills** holes, it does not assign a new physical page to a virtual page which is already assigned to your task.

Example 1.3 **Overlaying pages**

R0=1	Opcode
R1=5	Number of pages to get
R2=1	Virtual page number
R3=0	Speed flag

Virtual Page	**Physical Page**	**Physical Page**	
	Before	**After**	
0	>3F	>3F	
1	>3E	>3E	no change
2	>3D	>3D	no change
3	(>FF)	>33	new page
4	(>FF)	>32	new page
5	(>FF)	>31	new page
6	(>FF)	(>FF)	hole
7	>3C	>3C	
8	>3B	>3B	
9	>3A	>3A	
10	>39	>39	
11	>38	>38	
12	>37	>37	
13	>36	>36	
14	>35	>35	
15	>34	>34	

Notice that even though you asked for 5 pages, only 3 were actually assigned, since two of the specified pages had already been assigned.

RELEASE MEMORY

Function You will use this routine to return unused memory to MDOS, this is useful if your program uses lots of temporary data. This is also one of the functions used by MDOS to free memory when your task is terminated. Any page which is released by your task which is also currently mapped into your task's 64k of execution pages will be removed from the execution pages available to your task, and its entry in the mapper will be replaced by page >FF.

You may not release virtual page zero of your task using this function (although page zero may be accessed by your task, it doesn't really belong to your task.)

This opcode cannot be used to free shared pages belonging to a task. Shared memory pages must be freed with opcode #6.

Parameters	R0	= 2 (opcode)
	R1	= page count
	R2	= starting page
Results	R0	= error code

Parameter description

Page count This the number of memory pages you wish to free from your program. It is not necessarily the same as the number of physical memory pages which will actually be freed from your task. Shared pages, and unallocated pages in the range R2:(R2+R1-1) will not be released from your task. No pages will be released if this count is zero.

Starting page This is the page number of the first virtual memory page you wish to have released from your task. This procedure will attempt to release all of your task's virtual memory pages in the range R2:(R2+R1-1) into the free page pool.

Error code 0 = No Error. This indicates that the non-shared pages in the range specified have been released from your task back to the free pages in the system.

2 = Attempt to free page zero. This indicates that you tried to free virtual page zero of your task. No pages were actually released from your task.

8 = Out of table space. MDOS was unable to free a page because there weren't enough virtual pages nodes available to create a new page in the free pool. When you receive this error, it is possible that some, but not all, of the pages in the range R2:(R2+R1-1) have been moved to the free pool. You will receive this error if too many tasks have large gaps of unassigned pages in their memory maps. The current versions of MDOS allow 480 virtual pages between all tasks which are currently executing. Note that there are only 256 possible physical pages, and that there are only 128 physical pages even if you have the 512k expansion ram, so tasks would have to be pretty wasteful (have more gaps than actual pages) in order to use up all 480 virtual pages allowed by MDOS. If you get this error, your program should just give up and tell the user to try later.

Example 2.1 **Making a Hole**

R0=2	Opcode
R1=9	Number of pages to release
R2=2	First virtual page to release

Virtual Page	**Physical Page**	**Physical Page**	
	Before	**After**	
0	>3F	>3F	
1	>3E	>3E	
2	>3D	(>FF)	new hole
3	(>FF)	(>FF)	hole
4	(>FF)	(>FF)	hole
5	(>FF)	(>FF)	hole
6	(>FF)	(>FF)	hole
7	>3C	(>FF)	new hole
8	>3B	(>FF)	new hole
9	>3A	(>FF)	new hole
10	>39	(>FF)	new hole
11	>38	>38	
12	>37	>37	
13	>36	>36	
14	>35	>35	
15	>34	>34	

Note that only five pages were actually released from your task to MDOS, since some of the pages in the specified range were already unassigned.

Example 2.2 **Making list shorter**

R0=2	Opcode
R1=8	Number of pages to release
R2=10	First virtual page to release

Virtual Page	**Physical Page**	**Physical Page**	
	Before	**After**	
0	>3F	>3F	
1	>3E	>3E	
2	>3D	>3D	
3	(>FF)	(>FF)	
4	(>FF)	(>FF)	
5	(>FF)	(>FF)	
6	(>FF)	(>FF)	
7	>3C	>3C	
8	>3B	>3B	
9	>3A	>3A	
10	>39	null	freed
11	>38	null	freed
12	>37	null	freed
13	>36	null	freed
14	>35	null	freed
15	>34	null	freed

The list was truncated, since all of the pages at the tail of the list were unassigned. Also note that we really told it to release pages 10 to 18, but we only had pages up to 15 to begin with. No error is reported when you attempt to release unassigned pages.

MAP MEMORY

Function This routine can be used to place a physical page into the mapper chip for the specified virtual page belonging to your task. You can think of the mapper as providing seven usable 8k "memory windows" into your task's virtual memory space. You tell this routine which of the seven windows to use, and which part of virtual memory to look at. This routine can not be used to overwrite page zero of your task.

You should use this routine for mapping memory if you want your program to be able to use transparent demand paging in future versions of MDOS which support page swapping to hard disk.

(NOTE: If you are using window number seven, the one which is at >E000 in your direct address space; the data from offset >1000 to >1140 in the page will be corrupted by writes to addresses in the range >F000 to >F140. Do not use window number seven unless it is ok for the data in the specified range to be corrupted.)

On successful return, the specified virtual page belonging to your task has been mapped into the window you specified and is available for use.

Parameters	R0	= 3 (opcode)
	R1	= page number
	R2	= window number
Results	R0	= error code
	mapper	= new page

Parameter description

Window number This parameter, in the range 1:7, is used to tell MDOS which of the seven 8k byte windows in the processor's 16-bit address space to use for the specified virtual page belonging to your task.

Processor address = Window_number * >2000

Page number This is the virtual page number within your task that you wish to have mapped into the specified window in the processor's 16-bit address space. Virtual memory in the address range (page * >2000):(page * >2000 +>1fff) will be accessible to your program with the 16-bit addresses (window * >2000):(window * >2000 + >1fff).

Error code

0 = No Error. This indicates that the specified virtual memory page of your task has been mapped into the specified memory window.

2 = Header page mapping violation. You attempted to map a virtual page into window zero, which is reserved for your task's header by MDOS. Alternatively, you attempted to map virtual page zero, your task's header, into some other memory window. Your task's memory map has not been changed if you get this error.

3 = Unassigned virtual page, or Invalid memory window. The virtual page you specified has never been allocated by your task, and contains no valid data (it is a "hole" in your address space.) Alternatively, you specified a window number larger than seven. Your task's memory map has not be changed if you get this error.

Mapper

In MDOS mode, the mapper is 8 bytes long, and located at >F110 in the processor's 16-bit address space. Each byte in the mapper contains a physical memory page number, in the range >00:>FF. Assignments are as follows:

Mapper register = >F110 + Window_number

Note that each mapper register corresponds to a specific 8k block in the processor's 16-bit address space. After successful completion of the "Map Memory" function, the mapper register corresponding to the window you specified will contain the physical page number of the virtual page you specified.

Symbolicly:
mapper[window number] = task_pages[virtual page number]

GET MEMORY LIST

Function This operation returns an array of physical page numbers (each is one byte) corresponding to the virtual pages belonging to your task. You are allowed to specify the address of the first byte in the array, and the maximum number of elements in the array.

This array of physical page numbers is useful if you need to speed up memory paging in your task by use of your own paging code. If your program performs its own paging, you will need to call this opcode after every call to a memory management function which adds or removes pages from your virtual address space. (Opcodes 1,2,6,7)

In future versions of MDOS which support transparent demand paging, this operation will also "lock" **all** of your task's virtual pages into RAM, making them ineligible for paging. To "unlock" your virtual pages, another opcode will be provided for your use. (Normally, only pages which are currently mapped into one of the seven processor windows for your task would be "locked" into RAM.)

On successful return, the array will contain the physical page numbers for the virtual pages in your task.

Parameters	R0	= 4 (opcode)
	R1	= array start
	R2	= array size
Results	R0	= error code
	R1	= array used

Parameter description

Array start You specify the address of the first byte in your array using this parameter. The physical page number of your task's header page, virtual page number zero, is placed in the first byte of the array. This is a 16-bit processor address for a location which is currently mapped into your task's memory windows.

Array size This is the maximum number of physical page numbers which can be returned to your task. The indexes of the array elements can range from zero to (array size)-1.

Error code	0 = No Error. This indicates that the array contains all of the physical page associations for the virtual pages in your task. The number of actual array elements used can be less than the maximum size you specified for the array. 8 = Array not large enough. Your array was not large enough to hold all of the physical page numbers in use by your task. When you get this error, the contents of the array are valid up to the maximum element which you allowed.
Array used	This indicates the number of valid pages returned in the array. When you perform your own paging, you should make sure that you never index into the array after the last valid page (You will end up mapping a page which doesn't belong to your task.)

Sample Code

Assuming that you've already called this opcode, the follow code fragment will map in a data item pointed to by a 32 bit address.

```
 assume: r1,r2 = 32 bit pointer, @paglst are bytes from opcode #4

 movb r2,r1                 ok, since only low 5 bits of r1 are used
 andi r1,>e01f              keep 8 bits, zap the others
 src  r1,13                 rotate to make an index into the page list
 andi r2,>1fff              mask off the high three bits, they're now
*                           in R1 ...
*
 movb @paglst(r1),@mapper+4          put it at >8000
 movb @paglst+1(r1),@mapper+5        put next page at >a000
*
* it is not necessary to place two pages into the mapper if you
* know for certain that the record accessed by the pointer does not
* cross page boundaries, the above code is just a method of playing it
* safe
*
 mov  @>8000+field_offset(r2),r3
*
* this of course assumes that there is some record addressed by the
* initial pointer, and that the record contains fields of some
* data structure.  fields are easy to set up with a DORG statement
* for each record type in use by an application
*
```

DECLARE SHARED MEMORY

Function	This routine is used to declare a range of pages currently belonging to your task as "shared" memory pages, which means that they can be used by other tasks. An example of two tasks sharing memory would be an editor which "shared" all of its text buffer with an assembler, so that the assembler could assemble from RAM rather than from disk. Each group of pages declared as shared has a type, which you assign. When another application wants to share those memory pages with your task, it will ask MDOS to use a certain type of shared pages. An editor buffer could be declared as one specific type, while object code would be declared as a separate type, so that programs would not use the wrong sort of data as input (You wouldn't want a Fortran compiler to use a binary program as its input!) It is recommended that "types" be assigned by the distributor of MDOS, so that incompatible applications do not try to use the same "type" if you decide to use a "type" please correspond with the distributor of MDOS to coordinate your development efforts with others. A "shared" type may only be declared once, and always resides in a group of consecutive virtual pages. If all applications using a "shared" group of pages release those pages, the "type" may be redeclared. (MDOS keeps a count of the number of applications using a shared group, and if the count ever becomes zero, the type is made free for re-use) **NOTE:** It is not possible to declare page 0 to be part of a shared group. Page 0 is **always** private, since it contains the information which MDOS uses to distinguish between tasks.	
Parameters	R0	= 5 (opcode)
	R1	= page count
	R2	= starting page
	R3	= shared type
Results	R0	= error code

Parameter description

Page count	This is the number of consecutive virtual pages belonging to your task which will be declared as a shared page group for use by other tasks.
Starting page	This is the virtual page number within your task of the first virtual page which will become part of the shared page group. Pages in the range (start_page):(start_page+page_count-1) will belong to the group.
Shared type	This must be in the range >01:>FE, and should be a code unique to the format of data which you are sharing with other tasks. It is recommended that you use a common set of source code routines for data access for all of your tasks which use the data.
Error code	0 = No Error. This indicates that virtual pages you specified can now be shared by other tasks running under MDOS.
	3 = Bad page. At least one of the pages in the range (start_page):(start_page+page_count-1) has never been allocated by your task. The shared group has not been defined if you get this error.
	5 = Invalid type code. Your "shared type" parameter was not in the range >01:>FE, or your "shared type" code has already been declared by another task. The shared group does not contain the pages you specified if you get this error.
	7 = Invalid page declaration. At least one of the pages in the range (start_page):(start_page+page_count-1) is unallocated, already declared as shared, or is virtual page number zero. The shared group has not been defined if you get this error.
	8 = Out of table space. MDOS was unable to create the shared type because weren't enough virtual pages nodes available to create a shared page group descriptor list. You will receive this error if too many tasks have large gaps of unassigned pages in their memory maps. The current versions of MDOS allow 480 virtual pages between all tasks which are currently executing. Note that there are only 256 possible physical pages, and that there are only 128 physical pages even if you have the 512k expansion ram, so tasks would have to be pretty wasteful (have more gaps than actual pages) in order to use up all 480 virtual pages allowed by MDOS. If you get this error, your program should just give up and tell the user to try later.

RELEASE SHARED MEMORY

Function	This operation removes all shared memory pages of the specified type from your task's virtual memory list. If your task was the only task using the shared page group, the group will become undefined, and must be redeclared before use. Any page which is released by your task which is also currently mapped into your task's 64k of execution pages will be removed from the execution pages available to your task, and its entry in the mapper will be replaced by page >FF.

Parameters	R0	= 6 (opcode)
	R1	= shared type
Results	R0	= error code

Parameter description

Shared type	This is a shared group type number, in the range >01:>FE, and must have been previously defined by another task.
Error code	0 = No Error. This indicates that all of the pages from the shared group you specified have been released from your task.
	6 = Invalid type. The type you specified was not in the range >01:>FE, or hasn't yet been declared by another task.
	8 = Out of table space. MDOS was unable to free a page because there weren't enough virtual pages nodes available to create a new page in the free pool. When you receive this error, it is possible that some, but not all, of the pages belonging to the shared group have been moved to the free pool. If you get this error, your program should just give up and tell the user to try later.

USE SHARED MEMORY

Function This operation will include the pages from the shared type specified in your task's list of virtual pages. The shared type must be been previously declared by another task. When you call this function, all pages in the range (start_page):(start_page+shared_size-1) must not be allocated by your task, since you are not permitted to overlay shared and existing pages in your virtual page list.

Parameters	R0	= 7 (opcode)
	R1	= shared type
	R2	= start page
Results	R0	= error code

Parameter description

Shared type This is a type code for a shared page group which must have been declared by another task. If your task has enough contiguous available virtual pages beginning with the start page you specified, all of the pages from the shared page group will be mapped in at the specified virtual page address.

Start page This is the virtual page number within your task of the first virtual page which will be used by the shared page group. After calling this operation, you must explicitly map in the virtual pages which have just been assigned, since they will not be automatically placed into your task's mapper registers.

Error code 0 = No Error. The shared page group of the type you requested was already defined and was successfully mapped into your task's virtual page list.

2 = Attempt to overlay page zero. You specified virtual page zero as the start page for the shared memory group. No pages from the memory group have been allocated to your task if you get this error.

6 = Invalid shared type. The type you specified was not in the range >01:>FE or has not yet been defined for use by another task. No pages from the memory group have been allocated to your task if you get this error.

7 = Attempt to overlay shared and private memory. Your task did not have enough contiguous free virtual pages starting with the virtual page specified to map in the pages from the shared group. No pages from the memory group have been allocated to your task if you get this error.

8 = Out of table space. There were not enough free nodes available to extend your task's virtual page list. MDOS is out of table space. At this point, your task should give up and tell the user to try later. No pages from the memory group have been allocated to your task if you get this error.

GET SIZE OF SHARED GROUP

Function This operation reports the number of pages which belong to a shared page group. It should be used by your task before you have the shared page group assigned into your virtual page list, so that you know in advance if your task has enough unused contiguous pages to overlay the shared page group.

Parameters	R0	= 8 (opcode)
	R1	= shared type
Results	R0	= error code
	R1	= shared size

Parameter description

Shared type This is a type code for a shared page group which must have been declared by another task. On successful return, the number of pages in this page group is returned to your task.

Error code 0 = No Error. The shared page group of the type you requested was already defined and its size was returned to you.

6 = Invalid shared type. The type you specified was not in the range >01:>FE or has not yet been defined for use by another task.

Shared size On successful return, this will contain the size, in 8k pages, of the specified shared page group.

FREE TASK

Function	This routine can not be directly used by tasks under MDOS, it is reserved for use by system library functions. This is used to free all memory pages, except the task's header, from the task's list of virtual pages. If the task is using a shared page group, its reference to the group will be removed, and the group itself will be removed if this was the last task using the shared page group.	
Parameters	R0	= 9 (opcode)
	R1	= first node
Results	R0	= error code

Parameter description

First node	This is the MEMLST pointer from the task's header.
Error code	0 = No Error. The pages belonging to the task were freed. >FFFF = Invalid opcode. You attempted to call this from a user task.

GET MEMORY PAGE

Function	This routine can not be directly used by tasks under MDOS, it is reserved for use by system library functions. It is used to get a single memory page, by specific physical page number, or by speed priority.	
Parameters	R0	= 10 (opcode)
	R1	= physical page
	R2	= speed flag
Results	R0	= error code
	R1	= node pointer

Parameter description

Physical page	If this is in the range >00:>FF, MDOS will return a pointer to the memory node for the page, only if the page is currently unassigned. If this is larger than >FF, MDOS will return a pointer to the memory node for the the first free page in the system with the specified speed attribute.
Speed flag	This parameter is used only if the physical page number specified is larger than >FF. If this is zero, MDOS will allocate the first memory page available in the free list. If this is non-zero, MDOS will attempt to allocate the first zero wait state page available from the free list, if there are no zero wait state pages available, MDOS will allocate the first free page it finds.
Error code	0 = No Error. The page was reserved as specified, it is not assigned to any task, and it is not available for use. 1 = Page not available. The specified page was not free, or there are no free pages in the entire system. >FFFF = Invalid opcode. You attempted to call this from a user task.
Node pointer	This is pointer to a 4 byte memory node inside of the memory library's address space.

FREE MEMORY PAGE

Function This routine can not be directly used by tasks under MDOS, it is reserved for use by system library functions.

Adds the specified physical page to the list of pages available for use by user tasks.

Parameters	R0	= 11 (opcode)
	R1	= page number
Results	R0	= error code

Parameter description

Page number This a simply a physical page number to be freed.

Error code	0 = No Error. The page was reserved as specified, it is not assigned to any task, and it is not available for use.
	8 = Out of table space. MDOS was unable to create a free page because there weren't enough virtual pages nodes available to create a new page in the free pool.
	>FFFF = Invalid opcode. You attempted to call this from a user task.

FREE MEMORY NODE

Function This routine can not be directly used by tasks under MDOS, it is reserved for use by system library functions.

This operation will add the specified 4-byte node to the memory nodes available for use by the other memory management library functions.

Parameters	R0	= 12 (opcode)
	R1	= node address
Results	R0	= error code

Parameter description

Node address	This is the address of the 4-byte node within the memory management library's address space.
Error code	0 = No Error. The node was added to the free node list.
	>FFFF = Invalid opcode. You attempted to call this from a user task.

LINK MEMORY NODE

Function This routine can not be directly used by tasks under MDOS, it is reserved for use by system library functions.

This is used to link memory nodes together. It can be used to link page nodes onto a task's virtual memory list, to link page nodes to the system free page list, and to link nodes into the free node list.

Parameters	R0	= 13 (opcode)
	R1	= new node
	R2	= old node
Results	R0	= error code

Parameter description

New node The node to be inserted into a list after the old node.

Old node The node, in a node list, after which the new node is to be inserted.

Error code 0 = No Error. The nodes were linked together.

>FFFF = Invalid opcode. You attempted to call this from a user task.

GET MEMORY LIST (system)

Function This routine can not be directly used by tasks under MDOS, it is reserved for use by system library functions.

This routine will return a task's virtual page list to location >1F00 in system page zero. This is used primary by the DSR routines to locate data pointed to by a task's PAB buffer address.

Parameters	R0	= 14 (opcode)
Results	R0	= -1 (error code)
	R0	= page count (no error)

Parameter description

Error code >FFFF = Invalid opcode. You attempted to call this from a user task.

Page count This is the number of valid pages in the page list at >1F00 in system page zero. This count is also returned at >1FFE.

DSR OVERVIEW

The MDOS device drivers (Device Service Routines) are used to transfer data from your program to a mass storage device, from a mass storage device to your program; and to transfer data between your program and a character oriented device like a terminal, modem or printer. The device drivers are also used to control device characteristics such as baud rate, creating subdirectories, and initializing new media.

Available Devices

DSK1 ... DSK9	DSK1 through DSK9 are block oriented devices with a floppy disk directory structure. Each of these devices is limited to 3200 sectors. The assignment of these names to actual drives/ramdisks can vary depending on your hardware configuration.
HDS1 ... HDS3	HDS1 through HDS3 are block oriented devices with a winchester disk directory structure. See your HFDC owner's manual for more information on these devices.
RS232, RS232/1, RS232/2 RS232/3, RS232/4	RS232 through RS232/4 are serial bidirectional asynchronous character devices, both input and output is buffer with spoolers whose size you configured with your AUTOEXEC file.
PIO, PIO/1, PIO/2	PIO through PIO/2 are parallel bidirectional character devices, usually used to send data to a printer. Both input and output is buffered with spoolers whose size was set with your AUTOEXEC file.
WDS1 ... WDS4	WDS1 through WDS4 are used to address the Personality Card winchester disk system. These devices do not support all MDOS DSR operations.

CALLING DSR FUNCTIONS

The MDOS device drivers must be called from within a machine code program running as a task under MDOS. You pass arguments to the DSR using a small area of memory known as a PAB (Peripheral Access Block.) The primary parts of a

PAB are an opcode, an error flag, and a full filename, other areas of the PAB have meanings which are specific to the file operation you wish to perform.

The MDOS device drivers are invoked from a machine code program when software trap number zero (XOP 0) is called with a library number of 8. The calling program's R0 must contain the 16-bit address of the PAB at the time of the XOP. The calling program's R0 must be located between >A000 and >FFD8 in memory, and should be located in the PAD ram at >F000 if possible. The following code fragment will rename a file on floppy drive #1.

```
          LI      R0,PABADR
          XOP     @EIGHT,0
          MOVB    @PABADR+2,R0    check for error
          JNE     ERROR
*
          BL      @PRINT
          TEXT    'File rename successful'
          BYTE    >0D,>0A,0
*
* ...
*
NEWNAM    TEXT    'NEWFILE'
PABADR    BYTE    >0D        rename opcode
          BYTE    >00
          DATA    0
          DATA    NEWNAM     assumes NEWNAM is mapped in
          DATA    0,0,0,0
          DATA    NAMLEN
NAME      TEXT    'DSK1.OLDFILE'
NAMLEN    EQU     $-NAME
*
```

In the preceding example, three hidden assumptions were made. First, it is assumed that the program's registers are correctly located, with the Workspace Pointer register of the 9995 containing a value from >A000 to >FFD8. Second, it is assumed that PABADR is located on a page which is currently mapped into a memory page which has the same 16-bit address page number as its Virtual address page number (read the section on Memory Management.) The third assumption is that NEWNAM is actually at the virtual address NEWNAM, not in some overlay segment with a different virtual address.

Always check these three assumptions in your own programs!

OPEN

Function For block devices, such as hard disk and floppy disk, this operation must be used to prepare a structured file for access with READ and WRITE opcodes. You must close the file before terminating your task. No other task in MDOS is permitted to open the file while your task has it open.

For character devices, such as RS232 and PIO, this operation is only used to change the operating modes of the device port you specified as part of the file name. Multiple tasks are allowed simultaneous access to the RS232 and PIO ports...user beware. You do not need to open a character device before reading and writing to it.

PAB format Parameters passed to OPEN

byte offset	size	parameter
0	1 byte	opcode = >00
1	1 byte	mode flags
6	2 bytes	records to reserve for newly created file
8	2 bytes	record length
15	1 byte	name length
16	string	file name

Parameters returned from OPEN

byte offset	size	parameter
2	1 byte	error code
8	2 bytes	your record length
12	2 bytes	actual record length

Parameter description

Opcode >00 is the opcode for the OPEN function in the DSR.

This opcode only applies to structured files with the display or internal attributes. You will get an error if you attempt to open a non-structured file in this manner.

Flag byte

Bit	Meaning	
0 (lsb)	0 =	Sequential access file, you must use sequential access for files with variable record lengths.
	1 =	relative access file, this can only be used with fixed record length files.
2,1	00 =	Update mode, can only be used with fixed record length files. You are allowed to READ and WRITE records to the file.
	01 =	Output mode, this is used to create a new file. You are only allowed to WRITE records to this file. If the file already existed on disk, the old contents of the file will be forgotten. You will get an error if there is already a Protected file with the name you specified.
	10 =	Input mode, this is used prepare for reading from an existing file. You will get an error if the file doesn't already exist.
	11 =	Append mode, this is used to prepare for writing to the end of an existing file, or for creating a new file. You will get an error if you try to use this with a fixed record length file.
3	1 =	Use display format data
	0=	Use internal format data This bit has no effect on the data actually stored in the file. It is provided by the programmer as an indication of the type of data in the file. You will get an error if this flag does not match the attributes of a file you are opening for update, input, or append modes.
4	0=	File will use variable record lengths
	1 =	File will use fixed record lengths This flag will affect how data is stored in the file, and whether you can use relative record access within the file.
5,6	00	RESERVED, set to ZERO.
7	0	IGNORED
	1	For RS232 and PIO devices only, setting this bit to one will cause the driver to change the mode of the port as specified in the switches you placed after the device name in the filename. The mode will be set after all data currently in the output spooler for the specified device has been processed.

Error code	
>00	No error occurred, the file is open and ready for reads and writes.
>20	Write Protection violation. A file you were trying to open for UPDATE, APPEND, or OUTPUT could not be created because the floppy disk has a write-protect tab. Alternatively, an existing file which you were trying to open for UPDATE, APPEND, or OUTPUT has the "protected" attribute bit set in its directory entry.
>40	Invalid attributes. You specified a record length which was different than that of an existing file. Or, you tried to open a fixed file in APPEND mode. Or, the attributes you specified in the mode byte do not agree with the file attributes of an existing file on the disk.
>80	Out of space. There are not enough free sectors on the device to create the file you specified; or, the directory on the specified device already has the maximum of 127 entries; or, there are too many files open in MDOS, by your task and other tasks.
>C0	Media error. For some reason, MDOS encountered an unrecoverable error when trying to access the specified device. For floppy disks, this could mean that the drive is empty. For hard disks, this means that there is no device present, or that MDOS was unable to read a needed sector from the hard drive.
>E0	General purpose error. An error which didn't fit any of the previous descriptions. You will get this error if a file you specified for INPUT mode does not yet exist. You will also get this error if any subdirectory specified as part of the filename does not exist on the specified device.

Reserved Records — If this parameter is set to zero during the OPEN call, no space will be reserved for a newly created file. You can reserve sectors for a file when you open the file by setting this parameter to the number of records you expect to put into a newly created file. Reserving space during the OPEN call can greatly decrease the time needed to write records to a new file since MDOS doesn't have to spend time allocating more sectors for the file each time you want to write more records to the file. For files with record lengths less than 255 characters, the number of sectors reserved is calculated with the following formula: Sectors = Records/ INT(256/Record_Size). For files with record lengths longer than 255 characters, the number of sectors reserved is calculated with the formula: Sectors = INT((Records*Record_Size)/256). Note that if you specified a record length of ZERO, a record length of 80 will be used by MDOS.

For variable files with record lengths less than 255, the record size is the record length plus one. For fixed files, the record size is the same as the record length. For variable files with records lengths greater than 255, the record size is the record length plus two.

Record Length

This parameter specifies the default record length for a file. If you are creating the file, and this parameter has a zero value, MDOS will use a record length of 80 characters. If the file already exists, passing a zero value for this parameter will cause MDOS to use the record length of the existing file. If the file already exists, and you specify a non-zero value for this parameter which is different from the record length of the file, you will get an error. Specifying a zero value will always open any structured file.

On return, this will be the same as you set it, unless you initially specified a zero length. If you specified a zero length, and the file already existed, this will have the true record length of the file. If you specified a zero length, and you were creating a new file, this will have a value of 80.

Actual Length

On return, if the file you were trying to access already existed, this will contain the record length of the file. This is returned even if you get an error for specifying an incorrect record length for a file which already exists. If you just created the file, this will contain your record length.

Filename length

This is a count of the number of characters in the filename string.

Filename string

For block devices, such as disks and hard disks, this string must contain the name of the device you wish to access, followed by a list of the subdirectories separated by periods, followed by the name of the file you wish to access. (Example: "HDS1.SOURCE.UTIL.EXAMPLE") The length of the name, including the period separators, must be limited to 40 characters. For the OPEN call, MDOS will not create directories as specified in your pathname. If the specified directories did not exist, you will get an error. Floppy disks are currently limited to one subdirectory level. The individual names for the subdirectories and your filename are limited to ten characters, you will get an error if you try to use more than ten characters.

For character devices, such as RS232 and PIO, this string must contain the name of the device you wish to set the switches for, followed by a list of switches you wish to set separated by periods.

(Example: "RS232/2.BA=9600.DA=8.PA=N.CR") Note that the switches are set only if you set bit seven of the flag byte to a one. The switches will not take affect until all output previously loaded into the spooler for the specified device has been processed.

PIO	The following switch extensions may be used with the PIO ports.
CR	Turn off carriage returns and line-feeds after each variable record sent.
LF	Turn off line-feeds after each variable record sent, each variable record is still followed by a carriage return.
NU	Print nulls after each variable record to allow for a low slew-rate printer.
IB	Reconfigure the spooler to recognize a printer with an Inverted-Busy handshake signal. (If your printer doesn't seem to work with MDOS, try turning this switch on.)
HS	Reconfigure the spooler to perform a full handshake with the printer for each byte sent (instead of just strobe.) (If your printer doesn't seem to work with MDOS, try turning this switch on.)
RS232	The following switch extensions may be used with the RS232 ports.
CR	Turn off carriage returns and line-feeds after each variable record sent.
LF	Turn off line-feeds after each variable record sent, each variable record is still followed by a carriage return.
NU	Print nulls after each variable record to allow for a low slew-rate printer.
BA=baudrate	(110,300,600,1200,2400,4800,9600,19200)
DA=databits	(7,8)
PA=parity	(O,E,N) for (Odd,Even,None), respectively.
TW	Use two stopbits on transmission instead of one stopbit. (If your device doesn't seem to work with MDOS, try turning this switch on.)
CH	Check parity on each input character.

CLOSE

Function

This operation must be performed to close an open file. It will cause MDOS to write any modified file buffers associated with the file; and if the file was modified, MDOS will write out a new copy of the file's directory entry.

You must perform this operation on every file you have open before causing your program to terminate.

This operation has no effect on a character devices such as RS232 and PIO.

PAB format

Parameters passed to CLOSE

byte offset	size	parameter
0	1 byte	opcode = >01
15	1 byte	name length
16	string	file name

Parameters returned from CLOSE

byte offset	size	parameter
2	1 byte	error code

Parameter description

Opcode	>01 is the opcode for the CLOSE function in the DSR.

Error code

>00	No error occurred, the file is now closed, or the file wasn't even open.
>20	Write Protection violation. MDOS was unable to flush the file's buffers to disk. This should only happen if the user switched floppy disks while the file was open.
>C0	Media error. For some reason, MDOS encountered an unrecoverable error when trying to flush the buffers for the specified file. For floppy disks, this could mean that the user prematurely removed the disk from the drive. Alternatively, MDOS was unable to locate the place on the disk where the flushed data was to be written.
>E0	General purpose error. An error which didn't fit any of the previous descriptions.

READ

Function

This operation is used to transfer data from the specified device/file to a buffer you specify. For block devices such as floppy and hard disk, the file must already be open.

You can read from a character device such as RS232 at any time, no OPEN operation needs to be performed.

Reading from a character device is designed to accept input from a user typing at a terminal. For variable record length files, MDOS will interpret certain control characters (described later in this section) input from the device as editing commands for the current input line, and display appropriate changes on the user's terminal.

PAB format

Parameters passed to READ

byte offset	size	parameter
0	1 byte	opcode = >02
3	3 bytes	buffer address
6	2 bytes	record number
8	2 bytes	record length
10	1 byte	CPU/VDP flag
15	1 byte	name length
16	string	file name

Parameters returned from READ

byte offset	size	parameter
2	1 byte	error code
6	2 bytes	next record number
11	1 byte	ZERO
12	2 bytes	character count

Parameter description

Opcode	>02 is the opcode for the READ function in the DSR.
Error code	
>00	No error occurred, the read was successful.
>A0	You attempted to read past the end of the file's previous contents. Your buffer does not contain valid data.
>C0	For block devices, media error. For some reason, MDOS encountered an unrecoverable error when trying read data from the specified file. For floppy disks, this could mean that the user prematurely removed the disk from the drive. Alternatively, MDOS was unable to locate the place on the disk where the data was to be read from. For RS232 character devices, the hardware detected an error such as incorrect parity, byte too long, character buffer overflow (a character was lost.)
>E0	General purpose error. An error which didn't fit any of the previous descriptions. You will get this error if the file you specified is not currently open.
Buffer Address	This is where you specify the address to which data is to be transferred. For transfers to VDP ram, only the lowest 17 bits of the 3 bytes are significant. For transfers to CPU ram, only the lowest 21 bits of the 3 bytes are significant. For CPU ram transfers, the lowest 13 bits are an offset into one of your task's memory pages, and the other 8 bits specify which of your task's pages the transfer starts on. This address is not necessarily the same as the 16-bit CPU address your task will use to examine the data, depending on how your task has altered the map of its execution memory pages (see the section on Memory Management.)
Record Number	The record number is only valid for Fixed record files. You will get an error if you set the record number higher than the highest record ever written to the file. It is possible to read a buffer full of garbage if you specify a record number which has never been written to. On return, the record number has been incremented by one from the value you passed. Note that no distinction is made between Relative and Sequential access for Fixed files. All access

	to fixed files is treated as sequential unless you change the record number within your program.
Record Length	For block devices, your program should not alter this value. For character devices, set this value to the length of your input buffer.
CPU/VDP Flag	If this byte is zero, data will be transferred to a buffer in the memory belonging to your task, at the address specified in the buffer address. If this byte is non-zero, data will be transferred to VDP ram.
Character Count	On return, this contains the number of characters placed into your input buffer. For fixed files on block devices, this is always the same as the record length for the file. If you were reading fixed records from a character device, this value may be smaller than the record length if no input was available.
NOTE:	For variable record length files on block devices, this count can actually be larger than the specified record length, for compatibility with existing applications like TI-Extended Basic. (The notable example is I/V 163 Merge format files, which can have records longer than 163 characters.)
Filename length	This is a count of the number of characters in the filename string.
Filename string	For block devices, such as disks and hard disks, this string must contain the name of a file which you currently have opened. For character devices, this is simply the name of the device, all switches after the name are ignored.

WRITE

Function

This operation is used to transfer data from a buffer you specify to the specified device/file. For block devices such as floppy and hard disk, the file must already be open.

You can write to a character device such as RS232 or PIO at any time, no OPEN operation needs to be performed.

Writing to a character device with fixed record lengths causes the number of characters you specified to be written to the device, no end of line markers (except possibly NULLS) are added to the data. Writing to a character device with variable record lengths causes the number of characters you specified to be written to the device, possibly followed by a carriage return and a linefeed, just a linefeed, or no extra characters, depending on the switches you set with an OPEN call. NULLS may be added to your data if you turned NULLS on with an OPEN call.

PAB format

Parameters passed to WRITE

byte offset	size	parameter
0	1 byte	opcode = >03
3	3 bytes	buffer address
6	2 bytes	record number
8	2 bytes	record length
10	1 byte	CPU/VDP flag
15	1 byte	name length
16	string	file name

Parameters returned from WRITE

byte offset	size	parameter
2	1 byte	error code
6	2 bytes	next record number

Parameter description

Opcode	>03 is the opcode for the WRITE function in the DSR.
Error code	
>00	No error occurred, the write was successful.
>20	Write protection. MDOS was unable to write data to the file. If the file was opened in OUTPUT mode, this means that the user has switched disks since the file was opened, and that the new disk is write-protected. For a file opened in APPEND or UPDATE mode, this probably means that the disk was write-protected from the outset.
>80	Out of space. The disk is full.
>C0	For block devices, media error. For some reason, MDOS encountered an unrecoverable error when trying read the sector usage bitmap information or trying to write data into the file itself. For floppy disks, this could mean that the user prematurely removed the disk from the drive. Alternatively, MDOS was unable to locate the place on the disk where the data was to be written.
>E0	General purpose error. An error which didn't fit any of the previous descriptions. You will get this error if the file you specified is not currently open.
Buffer Address	This is where you specify the address from which data is to be transferred. For transfers from VDP ram, only the lowest 17 bits of the 3 bytes are significant. For transfers from CPU ram, only the lowest 21 bits of the 3 bytes are significant. For CPU ram transfers, the lowest 13 bits are an offset into one of your task's memory pages, and the other 8 bits specify which of your task's pages the transfer starts on. This address is not necessarily the same as the 16-bit CPU address your task will use to initialize the data, depending on how your task has altered the map of its execution memory pages (see the section on Memory Management.)
Record Number	The record number is only valid for Fixed record files. Writing to a record past the end of the current file contents will cause the file to be expanded, possibly causing a disk full error. On return, the record number has been incremented by one from the value you passed. Note that no distinction is made between

	Relative and Sequential access for Fixed files. All access to fixed files is treated as sequential unless you change the record number within your program.
Record Length	For block devices, your program should not alter this value. For character devices, set this value to the length of your output buffer.
CPU/VDP Flag	If this byte is zero, data will be transferred from a buffer in the memory belonging to your task, at the address specified in the buffer address. If this byte is non-zero, data will be transferred from VDP ram.
Filename length	This is a count of the number of characters in the filename string.
Filename string	For block devices, such as disks and hard disks, this string must contain the name of a file which you currently have opened. For character devices, this is simply the name of the device, all switches after the name are ignored.

RESTORE

Function

For block devices, such as hard disk and floppy disk, this resets the file so the the next record read will come from the beginning of the file, or the next record written will go at the beginning of the file. The only exception is fixed record files with the relative mode on, those files will be restored to the record number you specify.

This is the only opcode which makes a distinction between sequential and relative access fixed files.

PAB format

Parameters passed to RESTORE.

byte offset	size	parameter
0	1 byte	opcode = >04
1	1 byte	mode flags
6	2 bytes	record number
15	1 byte	name length
16	string	file name

Parameters returned from RESTORE

byte offset	size	parameter
2	1 byte	error code
6	2 bytes	record number

Parameter description

Opcode

>04 is the opcode for the RESTORE function

This opcode must be used on a file which was opened with the OPEN opcode. It does not apply to character devices such as RS232 and PIO.

Flag byte

Only the least significant bit has meaning to the restore opcode, and only for fixed files. If the least significant bit is set to one, for a fixed record file, the file pointer will be moved to the record you specified in the record number. Otherwise, the file pointer will be moved to the start of the file.

Error code	
>60	Bad opcode. You attempted to use the RESTORE operation on a character device.
>E0	General purpose error. An error which didn't fit any of the previous descriptions. You will get this error if the file you specified is not currently open.
Record number	For fixed files with the relative access mode bit set, this is the record number for subsequent access to the file. This is pretty useless, because you can always set the record number on any read or write. Upon return, this contains zero for sequential access fixed files, and the record number you specified for relative access fixed files.
Filename length	This is a count of the number of characters in the filename string.
Filename string	For block devices, such as disks and hard disks, this string must contain the name of a file which you currently have opened.

LOAD

Function

For block devices, such as hard disk and floppy disk, this operation is used to read a program image file into memory at the buffer address you specified.

For character devices, this operation is not implemented, but there was some thought about using the 1K XMODEM protocol to implement this operation.

PAB format

Parameters passed to LOAD

byte offset	size	parameter
0	1 byte	opcode = >05
3	3 bytes	buffer address
10	1 byte	CPU/VDP flag
11	3 bytes	buffer size
15	1 byte	name length
16	string	file name

Parameters returned from LOAD

byte offset	size	parameter
2	1 byte	error code
6	1 byte	>00
7	3 bytes	image size

Parameter description

Opcode

>05 is the opcode for the LOAD function in the DSR.

This opcode can only be used to transfer an entire program image from a block oriented storage device to memory. You will get an error if you try to load any other type of file.

Error code

>40	Bad attributes. You attempted to load a file which wasn't a program image.
>60	Bad opcode. You attempted to load from a character oriented device such as RS232 or PIO.
>80	Buffer overrun. The image file on disk is larger than the maximum buffer size you specified.
>C0	Media error. For some reason, MDOS encountered an unrecoverable error while trying to locate or read from the specified file. This could mean that a floppy drive is empty, or there

	is a bad sector on the floppy drive.
>E0	General purpose error. An error which didn't fit any of the previous descriptions. You will get this error if the file does not exist on the specified device.
Buffer Address	This is where you specify the address to which data is to be transferred. For transfers to VDP ram, only the lowest 17 bits of the 3 bytes are significant. For transfers to CPU ram, only the lowest 21 bits of the 3 bytes are significant. For CPU ram transfers, the lowest 13 bits are an offset into one of your task's memory pages, and the other 8 bits specify which of your task's pages the transfer starts on. This address is not necessarily the same as the 16-bit CPU address your task will use to examine the data, depending on how your task has altered the map of its execution memory pages (see the section on Memory Management.)
Image Size	These three bytes give the actual size of the image file on disk. This value is returned to you even if the file was not loaded because it was larger than your buffer. Note that image files loaded from the WDSx personality card winchester are limited to 16384 - (buffer_address MOD 8192) bytes in length.
CPU/VDP Flag	If this byte is zero, data will be transferred to a buffer in the memory belonging to your task, at the address specified in the buffer address. If this byte is non-zero, data will be transferred to VDP ram.
Buffer Size	You use these three bytes to inform MDOS of the maximum number of characters you wish to load as an image file. If the image file is longer than the size you specify here, none of the image file will be loaded, and you will get an error.
Filename length	This is a count of the number of characters in the filename string.
Filename string	For block devices, such as disks and hard disks, this string must contain the name of the device you wish to access, followed by a list of the subdirectories separated by periods, followed by the name of the file you wish to access. (Example: "HDS1.SOURCE.UTIL.EXAMPLE") The length of the name, including the period separators, must be limited to 40 characters. This file must already exist on the device you are accessing.

SAVE

Function

For block devices, such as hard disk and floppy disk, this operation is used to write a program image file to disk from memory at the buffer address you specified. If the file doesn't already exist, a new file will be created if there is room on the disk. If the file already exists, it must be an unprotected program image file, or you will get an error. When you are saving an image file, any subdirectories specified in the filename must already exist, they will not be created for you. You will get an error if the specified subdirectories do not already exist.

For character devices, this operation is not implemented, but there was some thought about using the 1K XMODEM protocol to implement this operation.

PAB format

Parameters passed to SAVE for image files.

byte offset	size	parameter
0	1 byte	opcode = >06
3	3 bytes	buffer address
10	1 byte	CPU/VDP flag
11	3 bytes	buffer size
15	1 byte	name length
16	string	file name

Parameters returned from SAVE.

byte offset	size	parameter
2	1 byte	error code

Parameter description

Opcode

>06 is the opcode for the SAVE function in the DSR.

This opcode can be used to transfer an entire program image from memory to a block oriented storage device.

Error code	
>20	Write Protection violation. The file could not be created because the floppy disk has a write-protect tab. Alternatively, the file already exists, and has the "protected" attribute set in its directory entry.
>40	Bad attributes. You attempted to save over an existing file which wasn't a program image.
>60	Bad opcode. You attempted to save an image on a character oriented device such as RS232 or PIO.
>80	Out of space. There are not enough free sectors on the device to create the file you specified; or, the directory on the specified device already has the maximum of 127 entries.
>C0	For block devices, media error. For some reason, MDOS encountered an unrecoverable error when trying read the sector usage bitmap information or trying to write data into the file itself. For floppy disks, this could mean that the user prematurely removed the disk from the drive. Alternatively, MDOS was unable to locate the place on the disk where the data was to be written.
>E0	General purpose error. An error which didn't fit any of the previous descriptions. You will get this error if any subdirectory specified as part of the filename does not exist on the specified device.

Buffer Address

This is where you specify the address from which data is to be transferred.

For transfers from VDP ram, only the lowest 17 bits of the 3 bytes are significant.

For transfers from CPU ram, only the lowest 21 bits of the 3 bytes are significant. For CPU ram transfers, the lowest 13 bits are an offset into one of your task's memory pages, and the other 8 bits specify which of your task's pages the transfer starts on. This address is not necessarily the same as the 16-bit CPU address your task will use to initialize the data, depending on how your task has altered the map of its execution memory pages (see the section on Memory Management.)

CPU/VDP Flag	If this byte is zero, data will be transferred from a buffer in the memory belonging to your task, at the address specified in the buffer address. If this byte is non-zero, data will be transferred from VDP ram.
Buffer Size	You use these three bytes to inform MDOS of the number of characters you wish to save as an image file.
Filename length	This is a count of the number of characters in the filename string.
Filename string	For block devices, such as disks and hard disks, this string must contain the name of the device you wish to access, followed by a list of the subdirectories separated by periods, followed by the name of the file you wish to save into. (Example: "HDS1.SOURCE.UTIL.EXAMPLE") The length of the name, including the period separators, must be limited to 40 characters.

CREATE DIRECTORY

Function

For block devices, this operation is used to create a new subdirectory on the specified device. You can create a new subdirectory by using a filename which ends in a period separator character. Any intermediate subdirectories specified in your filename must already exist, they will not be created for you. A filename of "HDS1.LEVEL1.LEVEL2.LEVEL3." would create a subdirectory named "LEVEL3" only if subdirectories "LEVEL1" and "LEVEL2" already existed on "HDS1". Note that only one level of subdirectories is allowed on a floppy disk device.

For character devices, this operation is not implemented.

PAB format

Parameters passed to CREATE DIRECTORY

byte offset	size	parameter
0	1 byte	opcode = >06
15	1 byte	name length
16	string	file name

Parameters returned from CREATE DIRECTORY

byte offset	size	parameter
2	1 byte	error code
6	1 byte	>00

Parameter description

Opcode

>06 is the opcode for the CREATE DIRECTORY function in the DSR. This is the same opcode as the SAVE function of the DSR, the DSR distinguishes between the two uses by looking for a period character at the end of the filename you specified.

Error code	
>20	Write Protection violation. The subdirectory could not be created because a floppy disk has a write-protect tab.
>60	Bad opcode. You attempted to create a subdirectory on a character oriented device such as RS232 or PIO.
>80	Out of space. There are not enough free sectors on the device to create the subdirectory you specified; or, the directory on the specified device already has the maximum of 114 subdirectory entries.
>C0	For block devices, media error. For some reason, MDOS encountered an unrecoverable error when trying read the sector usage bitmap information or trying to write data into the subdirectory itself. For floppy disks, this could mean that the user prematurely removed the disk from the drive. Alternatively, MDOS was unable to locate the place on the disk where the data was to be written.
>E0	General purpose error. An error which didn't fit any of the previous descriptions. You will get this error if any subdirectory specified as part of the filename (other than the last one in the filename, the one you wish to create) does not exist on the specified device.
Filename length	This is a count of the number of characters in the filename string.
Filename string	For block devices, such as disks and hard disks, this string must contain the name of the device you wish to access, followed by a list of the subdirectories separated by periods, followed by the name of the subdirectory you wish to create, followed by another period. (Example: "HDS1.SOURCE.UTIL.NEWDIR.") The length of the name, including the period separators, must be limited to 40 characters.

DELETE

Function

This operation serves three purposes.

For block devices, such as hard disk and floppy disk, this can be used to delete a file from a directory, returning sectors allocated to the file into available space on the disk. This operation can also be used to remove a subdirectory from a disk if the subdirectory is empty.

For character devices, this operation can be used to flush the output spooler for the device, causing all characters still in the output spooler to be purged.

PAB format

Parameters passed to DELETE.

byte offset	size	parameter
0	1 byte	opcode = >07
15	1 byte	name length
16	string	file name

Parameters returned from DELETE

byte offset	size	parameter
2	1 byte	error code

Parameter description

Opcode — >07 is the opcode for the DELETE function.

Error code

>20 — Write Protection violation. For files, the specified file could not be deleted because the disk is physically write-protected, or the "protected" mode bit is set in the file's directory entry. For subdirectories, the directory could not be deleted because the disk is physically write protected, or the directory still has subordinate files and directories within it (the directory is not empty.)

>C0 — For block devices, media error. For some reason, MDOS encountered an unrecoverable error when trying to update the sector usage bitmap information or trying to write data into the subdirectory itself. For floppy disks, this could mean that the user prematurely removed the disk from the drive. Alternatively, MDOS was unable to locate the place on the disk

	where the data was to be written.
>E0	General purpose error. An error which didn't fit any of the previous descriptions. You will get this error if any subdirectory specified as part of the filename (other than the last one in the filename, the one you wish to create) does not exist on the specified device. You will also get this error if the specified file/subdirectory did not exist.
Filename length	This is a count of the number of characters in the filename string.
Filename string	To delete files on block devices, such as disks and hard disks, this string must contain the name of the device you wish to access, followed by a list of the subdirectories separated by periods, followed by the name of the file you wish to delete. (Example: "HDS1.SOURCE.UTIL.EXAMPLE") The length of the name, including the period separators, must be limited to 40 characters.

To delete a subdirectory from a block device, such as disks and hard disks, this string must contain the name of the device you wish to access, followed by a list of the subdirectories separated by periods, followed by the name of the subdirectory you wish to create, followed by another period. (Example: "HDS1.SOURCE.UTIL.NEWDIR.") The length of the name, including the period separators, must be limited to 40 characters.

To purge the output spooler on a character device such as RS232 or PIO, this string must contain the name of the device. Any switches and other characters following the device name will be ignored.

DELETE RECORD

Function This operation is not currently implemented in MDOS. It would be used to delete a record from a key indexed file (possibly implemented as a B-Tree.)

PAB format Parameters passed to DELETE RECORD.

byte offset	size	parameter
0	1 byte	opcode = >08
15	1 byte	name length
16	string	file name

Parameters returned from DELETE RECORD

byte offset	size	parameter
2	1 byte	error code = >60

Parameter description

Opcode	>08 is the opcode for the DELETE RECORD function.
Error code	
>60	Bad opcode. This opcode is not implemented in MDOS.
Filename length	This is a count of the number of characters in the filename string.
Filename string	This would be the name of a currently open file on a block device.

STATUS

Function

For block devices, such as hard disk and floppy disk, this operation gives you information about the attributes of a specified file, as well as information regarding space available on the disk.

PAB format

Parameters passed to STATUS.

byte offset	size	parameter
0	1 byte	opcode = >09
15	1 byte	name length
16	string	file name

Parameters returned from STATUS

byte offset	size	parameter
2	1 byte	error code
14	1 byte	attribute byte

Parameter description

Opcode

>09 is the opcode for the STATUS function.

This opcode can be used on any file, even if it is open. It does not apply to character devices such as RS232 and PIO.

Error code

>60	Bad opcode. You attempted to use the STATUS operation on a character device.
>C0	Media error. For some reason, MDOS encountered an unrecoverable error while trying to locate or read from the specified file. This could mean that a floppy drive is empty, or there is a bad sector on the floppy drive.
>E0	General purpose error. An error which didn't fit any of the previous descriptions.

Attribute byte

Bit	Meaning	
0 (lsb)	0 =	Not at end of file.
	1=	At end of open file, read is not possible without EOF error. Write is possible if the file was opened for writing.
1	0=	There is room on the disk to expand the file.
	1=	Disk is full, there is no room to make the file larger.
2	0=	If the file is a data file, it has fixed length records.
	1=	If the file is a data file, is has variable length records.
3	0=	If file exists, it is a data file.
	1=	If file exists, it is a program image file.
4	0=	If the file is a data file, it contains display format data.
	1=	If the file is a data file, it contains internal format data.
5		This bit is not used.
6	0=	If the file exists, it is protected against write operations.
	1=	If the file exists, write operations are allowed.
7 (msb)	0=	File exists, all other bits specify file attributes.
	1=	File does not exist, all other bits must be ignored.

Filename length	This is a count of the number of characters in the filename string.
Filename string	For block devices, such as disks and hard disks, this string must contain the name of a file for which you want to obtain information.

FILE ID

Function

The file ID can be transferred from a file to your program via the BREAD function. You can cause MDOS to create a new file with all of the characteristics in a file ID provided by your task by use of the BWRITE function. The file ID is transferred between MDOS and your task is the buffer whose address you specify in the "buffer address" field of the PAB used by BREAD and BWRITE.

The file ID tells you and MDOS everything there is to know about the file other than where it is actually located on the disk.

File ID format in buffer

offset	size	
0	2 bytes	Extended record length for files with records longer than 255 bytes. All 16 bits are significant, so a record whose length is specified here can be from 256 to 65535 bytes long.
2	1 byte	File status flag bits.
(lsb)	0=	Data file.
	1=	Program image file.
1	0=	If data file, contains display format data.
	1=	If data file, contains internal format data.
2	RESERVED	
3	0=	File is not protected.
	1=	File is write protected.
4-6	RESERVED	
(msb)	0=	If data file, contains fixed length records.
	1=	If data file, contains variable length records.
3	1 byte	For data files, this is the number of records which can fit into on one sector. This byte is zero for program images files and files with record lengths longer than 256 bytes.
4	2 bytes	This is the 16 least significant bits of a 20 bit number representing the number of sectors reserved for the file. The four most significant bits are at offset 18 in the buffer.
6	1 byte	This is the number of bytes used in the last sector of the file. A zero value in this byte means that all 256 bytes in the last sector are used. This is used in determining the length of program image files, and to determine the EOF for variable record length files.

7	1 byte	This is the record length for data files with records shorter than 256 bytes. This is zero for program image files and data files with records longer than 255 bytes.
8	2 bytes	These two bytes must be reversed to form a 16 bit number which is the least significant 16 bits of a 20 bit number used to determine the EOF mark for files. The four most significant bits are at offset 19 in the buffer.
		For files with fixed record lengths, this is one greater than the highest record ever written to in the file.
		For program image files, this is the number of sectors actually used in the file. This, in conjunction with the number of bytes used in the last sector, is used to determine the number of bytes in the program image.
		For variable record files, this is simply the number of sectors actually used in the file, and is only used when you want to APPEND data to the file.
10	2 bytes	Date of file creation. This has three fields packed into the 16 available bits. The 7 most significant bits are the last two digits of the year. The next 4 bits are the month. The 5 least significant bits are the day of the month. The date 12 May 1988 would be encoded as (88*512 + 5*32 + 12) = >B0AC
12	2 bytes	Time of file creation. This has three fields packed into the 16 available bits. The 5 most significant bits are the hour, from 0 to 23. The next 6 bits are the minute, from 0 to 59. The 5 least significant bits are the seconds, divided by 2. The time 10:58:13 would be encoded as (10*2048 + 58*32 + 13/2) = >5746
14	2 bytes	Date of file update. Same format as the creation date.
16	2 bytes	Time of file update. Same format as the creation time.
18	1 byte	This contains the most significant 4 bits of the number of sectors reserved for the file.
19	1 byte	This contains the most significant 4 bits of the number of sectors used by the file.

BREAD

Function

For block devices, such as hard disk and floppy disk, this operation is used to read sectors directly from the disk or from within any type of file located on the disk into memory at the buffer address you specified. It can also be used to read subdirectory headers from a floppy disk.

With files on a block device, if you specify a sector count of zero, this can also be used to read the file's ID into the buffer you specified.

The values returned in the PAB are defined to make error recovery routines easy to code.

PAB format

Parameters passed to BREAD.

byte offset	size	parameter
0	1 byte	opcode = >0A
3	3 bytes	buffer address
6	2 bytes	LSW sector offset
10	1 byte	CPU/VDP flag
12	2 bytes	sector count
14	1 byte	MSB sector offset
15	1 byte	name length
16	string	file name

Parameters returned from BREAD.

byte offset	size	parameter
2	1 byte	error code
3	3 bytes	updated buffer address
6	2 bytes	updated LSW sector offset
12	2 bytes	remaining sectors
14	1 byte	updated MSB sector offset

Parameter description

Opcode

>0A is the opcode for the BREAD function in the DSR.

Error code	
>60	Bad opcode. You attempted to read from a character oriented device such as RS232 or PIO.
>A0	Read past end of file. When you are reading sectors directly from a disk, this error is reported when you attempted to read a sector number higher than any sector on the disk. When you are reading from a file, this error is reported to you when you attempt to read a sector past the last sector reserved for file (not at the last sector used by the file.)
>C0	Media error. For some reason, MDOS encountered an unrecoverable error while trying to locate or read from the specified file. This could mean that a floppy drive is empty, or there is a bad sector on the block device.
>E0	General purpose error. An error which didn't fit any of the previous descriptions. You will get this error if the file does not exist on the specified device.

Buffer Address

This is where you specify the address to which data is to be transferred.

For transfers to VDP ram, only the lowest 17 bits of the 3 bytes are significant.

For transfers to CPU ram, only the lowest 21 bits of the 3 bytes are significant. For CPU ram transfers, the lowest 13 bits are an offset into one of your task's memory pages, and the other 8 bits specify which of your task's pages the transfer starts on. This address is not necessarily the same as the 16-bit CPU address your task will use to examine the data, depending on how your task has altered the map of its execution memory pages (see the section on Memory Management.)

On return from the BREAD function with a non-zero sector count, the buffer address will be updated to point to the first byte in memory after the data you just loaded. If there were no errors this will be your initial buffer address + 256 * sectors. If there were errors, this will point to the where the first bad sector encountered would have been loaded.

Sector offset

The two bytes at offset 6 in the PAB, along with the byte at offset 14 in the PAB, form a 20 bit sector offset within the specified file or device. Sector numbering starts at zero, not at one.

On return from the BREAD function, this is updated to contain the offset of the sector in the file after the last sector successfully

	read. If there was an error while reading sectors, this will contain the offset of the sector which MDOS was unable to read.
CPU/VDP Flag	If this byte is zero, data will be transferred to a buffer in the memory belonging to your task, at the address specified in the buffer address. If this byte is non-zero, data will be transferred to VDP ram.
Sector count	To read sectors from a disk or from a file, this must be set to a non-zero number which tells MDOS how many sectors you want to read. To read a file ID from a file into your buffer, you must set this to zero. On return, this will contain the number of sectors not read due to an error condition.
Filename length	This is a count of the number of characters in the filename string.
Filename string	For access to a file on a block device, such as disk or hard disk, this string must contain the name of the device you wish to access, followed by a list of the subdirectories separated by periods, followed by the name of the file you wish to access. (Example: "HDS1.SOURCE.UTIL.EXAMPLE") The length of the name, including the period separators, must be limited to 40 characters. This file must already exist on the device you are accessing. You must specify a sector count of zero if you want to read the file ID into your buffer. For direct access to sectors on a block device, this string must contain the name of the device you wish to access, followed by a period. (Example: "HDS1.") For direct access to the pointers of a floppy disk subdirectory, this string must contain the name of the floppy disk device, followed by a period, followed by the subdirectory name, followed by another period. (Example: "DSK1.SUBDIR.") For this subdirectory, reading from sector zero or sector one will return information unique to that subdirectory in the same format as sector zero and sector one of the main floppy disk. Access to any sector beyond sector one will simply read the corresponding sector from the disk, as if no subdirectory had been specified.

BWRITE

Function

For block devices, such as hard disk and floppy disk, this operation is used to write sectors directly to the disk or into any type of file located on the disk from memory at the buffer address you specified.

With files on a block device, if you specify a sector count of zero, this can also be used to create a new file with the characteristics described in a file ID in the buffer you specified.

The values returned in the PAB are defined to make error recovery routines easy to code.

PAB format

Parameters passed to BWRITE.

byte offset	size	parameter
0	1 byte	opcode = >0B
3	3 bytes	buffer address
6	2 bytes	LSW sector offset
10	1 byte	CPU/VDP flag
12	2 bytes	sector count
14	1 byte	MSB sector offset
15	1 byte	name length
16	string	file name

Parameters returned from BWRITE.

byte offset	size	parameter
2	1 byte	error code
3	3 bytes	updated buffer address
6	2 bytes	updated LSW sector offset
12	2 bytes	remaining sectors
14	1 byte	updated MSB sector offset

Parameter description

Opcode >0B is the opcode for the BWRITE function in the DSR.

Error code	
>60	Bad opcode. You attempted to write to a character oriented device such as RS232 or PIO.
>80	Disk full. When you were creating a new file, there weren't enough sectors left on the disk to create the file. Alternatively, the specified subdirectory already had 127 file entries, and your new file could not be added to the directory.
>A0	Write past end of file. When you are writing sectors directly to a disk, this error is reported when you attempted to write to a sector number higher than any sector on the disk. When you are writing to a file, this error is reported to you when you attempt to write to a sector past the last sector reserved for file (not at the last sector used by the file.)
>C0	Media error. For some reason, MDOS encountered an unrecoverable error while trying to locate the specified file, or while trying to write to the file, or when trying to find more sectors for the file. This could mean that a floppy drive is empty, or there is a bad sector on the block device.
>E0	General purpose error. An error which didn't fit any of the previous descriptions. You will get this error if the file does not exist on the specified device (unless you were just creating the file.)

Buffer Address

This is where you specify the address from which data is to be transferred.

For transfers from VDP ram, only the lowest 17 bits of the 3 bytes are significant.

For transfers from CPU ram, only the lowest 21 bits of the 3 bytes are significant. For CPU ram transfers, the lowest 13 bits are an offset into one of your task's memory pages, and the other 8 bits specify which of your task's pages the transfer starts on. This address is not necessarily the same as the 16-bit CPU address your task will use to initialize the data, depending on how your task has altered the map of its execution memory pages (see the section on Memory Management.)

On return from the BWRITE function with a non-zero sector count, the buffer address will be updated to point to the first byte in memory after the data you just saved. If there were no errors this will be your initial buffer address + 256 * sectors. If there were errors, this will point to the first sector of data in your buffer which was not saved to the block device.

Sector offset	The two bytes at offset 6 in the PAB, along with the byte at offset 14 in the PAB, form a 20 bit sector offset within the specified file or device. Sector numbering starts at zero, not at one. On return from the BWRITE function, this is updated to contain the offset of the sector in the file after the last sector successfully written. If there was an error while writing sectors, this will contain the offset of the sector which MDOS was unable to write data into.
CPU/VDP Flag	If this byte is zero, data will be transferred from a buffer in the memory belonging to your task, at the address specified in the buffer address. If this byte is non-zero, data will be transferred from VDP ram.
Sector count	To write sectors to a disk or to a file, this must be set to a non-zero number which tells MDOS how many sectors you want to write. To create a new file, with the file ID from your buffer, you must set this to zero. On return, this will contain the number of sectors not written due to an error condition.
Filename length	This is a count of the number of characters in the filename string.
Filename string	For access to a file on a block device, such as disk or hard disk, this string must contain the name of the device you wish to access, followed by a list of the subdirectories separated by periods, followed by the name of the file you wish to access. (Example: "HDS1.SOURCE.UTIL.EXAMPLE") The length of the name, including the period separators, must be limited to 40 characters. This file must already exist on the device you are accessing (unless you are in the process of creating the file with your own file ID.) You must specify a sector count of zero if you want to create a new file with the ID information in your buffer. For direct access to sectors on a block device, this string must contain the name of the device you wish to access, followed by a period. (Example: "HDS1.")

PROTECT

Function For block devices, such as hard disk and floppy disk, this operation can be used to remove or set write-protection for the specified file.

PAB format Parameters passed to PROTECT.

byte offset	size	parameter
0	1 byte	opcode = >0C
2	1 byte	protection flag.
15	1 byte	name length
16	string	file name

Parameters returned from PROTECT.

byte offset	size	parameter
2	1 byte	error code

Parameter description

Opcode	>0C is the opcode for the PROTECT function. This opcode should only be used on a file which is not open.
Protection flag	If this byte is zero, the specified file will be made into an unprotected file. If this byte is non-zero, the file will have the write-protect bit set in its directory entry.

Error code

>20	Write Protection violation. For files, the protection of the specified file could not be changed because the disk is physically write-protected.
>60	Bad opcode. You attempted to use the PROTECT operation on a character device.
>C0	Media error. For some reason, MDOS encountered an unrecoverable error while trying to locate or update the directory entry for the specified file. This could mean that a floppy drive is empty, or there is a bad sector on the floppy drive.
>E0	General purpose error. An error which didn't fit any of the previous descriptions. You will get this error if the file you specified does not exist on the specified device.

RENAME

Function

For block devices, such as hard disk and floppy disk, this operation can be used to rename a file or a subdirectory on the device. It can also be used to change the volume label on a block device. After a rename operation of a file or subdirectory, the filename in the PAB will be updated to reflect the new name of the file or subdirectory.

PAB format

Parameters passed to RENAME.

byte offset	size	parameter
0	1 byte	opcode = >0D
3	3 bytes	buffer address
10	1 byte	CPU/VDP flag
15	1 byte	name length
16	string	file name
@buffer	10 bytes	new file name

Parameters returned from RENAME

byte offset	size	parameter
2	1 byte	error code
15	1 byte	name length
16	string	new file name

Parameter description

Opcode

>0D is the opcode for the RENAME function.

This opcode should only be used on a file which is not open.

Error code	
>20	Write Protection violation. The name of the specified file or subdirectory could not be changed because the disk is physically write-protected.
>60	Bad opcode. You attempted to use the RENAME operation on a character device.
>C0	Media error. For some reason, MDOS encountered an unrecoverable error while trying to locate or update the directory entry for the specified file. This could mean that a floppy drive is empty, or there is a bad sector on the floppy drive.
>E0	General purpose error. An error which didn't fit any of the previous descriptions. You will get this error if the file you specified does not exist on the specified device.

Buffer Address

This is where you specify the address from which MDOS will obtain the new name for the file.

For transfers from VDP ram, only the lowest 17 bits of the 3 bytes are significant.

For transfers from CPU ram, only the lowest 21 bits of the 3 bytes are significant. For CPU ram transfers, the lowest 13 bits are an offset into one of your task's memory pages, and the other 8 bits specify which of your task's pages the transfer starts on. This address is not necessarily the same as the 16-bit CPU address your task will use to set the filename, depending on how your task has altered the map of its execution memory pages (see the section on Memory Management.)

New name

The buffer must contain ten characters for the new name of the file or directory. If the name is shorter than ten characters, you must add enough trailing spaces to the name to make the buffer contain ten characters.

CPU/VDP Flag

If this byte is zero, data will be transferred to a buffer in the memory belonging to your task, at the address specified in the buffer address. If this byte is non-zero, data will be transferred to VDP ram.

Filename length

This is a count of the number of characters in the filename string.

On return from the RENAME operation, this contains the length of the new filename.

Filename string

To rename a file on a block device such as disks and hard disks, this string must contain the name of the device you wish to access, followed by a list of the subdirectories separated by periods, followed by the name of the file you wish to rename. (Example: "HDS1.SOURCE.UTIL.FILE") The length of the name, including the period separators, must be limited to 40 characters.

To rename a subdirectory on a block device such as disks and hard disks, this string must contain the name of the device you wish to access, followed by a list of the subdirectories separated by periods, followed by the name of the subdirectory you wish to rename, followed by another period. (Example: "HDS1.SOURCE.UTIL.DIRECT.") The length of the name, including the period separators, must be limited to 40 characters.

To change the name of a volume in a block device, this string must contain only the name of the device, and the buffer must contain the new name for the volume.

FORMAT

Function For floppy disk block devices, this operation is used to initialize blank diskettes, or to reinitialize used diskettes (erasing the previous contents of the disk.)

PAB format Parameters passed to FORMAT.

byte offset	size	parameter
0	1 byte	opcode = >0E
3	1 byte	tracks
4	1 byte	skew
5	1 byte	interlace
6	1 byte	density
7	1 byte	sides
15	1 byte	name length
16	string	device name

Parameters returned from FORMAT

byte offset	size	parameter
2	1 byte	error code
8	2 bytes	sector count
11	1 byte	actual sides
12	1 byte	sectors per track

Parameter description

Opcode >0E is the opcode for the FORMAT function.

This opcode should only be used on a device which does not have any open files.

Error code	
>20	Write Protection violation. The name of the specified file or subdirectory could not be changed because the disk is physically write-protected.
>60	Bad opcode. You attempted to use the FORMAT operation on a character or winchester device.
>C0	Media error. For some reason, MDOS encountered an unrecoverable error while trying to initialize the disk. This could mean that a floppy drive is empty, there is a bad sector on the floppy drive, or the disk drive is defective.
>E0	General purpose error. An error which didn't fit any of the previous descriptions.

Tracks	This is the number of tracks per side to format on the specified device. For double sided disks, this must be either 40 or 80. For single sided disks, this can range from 1 to 40, or 80. Track counts between 40 and 80 can not be used. Track counts larger than 80 cannot be used.
Skew	This is the rotational delay between the last sector on a specified track, and sector zero on the next track, specified in sectors. This delay can be optimized so that sector zero of the subsequent track is ready as soon as a head step operation between tracks is complete, eliminating an extra rotation of the floppy disk after a head step. One disk revolution is 200 milliseconds on a standard 5.25" floppy disk. The optimum delay for a single-track head step is about 35 milliseconds. The optimum skew is generally (35/200) multiplied by the sectors per track, rounded up to the next highest integer. Skew is NOT the spacing between the zero sector on consecutive tracks.
Interlace	Interlace is used to adjust rotational delays between consecutively numbered sectors on the same track. After reading a sector from the disk, MDOS will immediately process data from the sector, returning information from the sector to the calling task. In the time MDOS spends processing the data, the disk is still turning and several sectors may pass the head of the disk drive. If one of those sectors which went by as MDOS was processing the data happened to be the next sector that MDOS needed, MDOS would have to wait for the disk to spin around again before it could read that sector from the disk. Interlace is used to specify how many sectors are likely to go by as MDOS is processing the current sector. Interlace is the number of revolutions of the disk it would take MDOS to read an entire track, assuming that MDOS

could read consecutively numbered sectors from the track without the extra rotational delay. With an interlace of 2, consecutively numbered sectors on a track would have another sector between them, a sector which MDOS will miss as it is processing the data from the first sector.

Density	Setting this byte to >02 will cause MDOS to format the disk with 18 sectors per track on a double-density controller card. Setting this byte to any other value will cause MDOS to format the disk with 9 sectors per track on all controller cards.
Sides	Setting this byte to >02 will cause MDOS to format both sides of a disk in a double-sided drive. If the drive is not double-sided, and you set this byte to >02, only one side of the disk will be formatted. Setting this byte to any value other than >02 will cause MDOS to only format one side of the disk.
Sector Count	On return from the FORMAT operation, this is a 16 bit integer which indicates the number of sectors available on a freshly formatted diskette.
Actual sides	On return from the FORMAT operation, this is the number of sides of the disk that MDOS formatted. For a single sided drive, this will always be >01. For a double sided drive, the number here will depend on the number of sides you asked to be formatted.
Sectors	On return from the FORMAT operation, this byte will contain a >09 for single density disks, and >12 for double density disks.
Device name length	This is a count of the number of characters in the filename string.
Device name	For block devices, such as disks and hard disks, this string must contain the name of the device you wish to format, followed by a period. (Example: "DSK1.")

UTILITY - CONTENTS

Page

UTILITY OVERVIEW

The memory management routines in MDOS are provided to aid a programmer in writing applications which are larger than the 64 Kbytes directly addressable by the CPU's 16 address lines. They also serve the purpose of providing each task with it's own private address space, separate from other the memory accessible to other tasks.

CALLING UTILITY FUNCTIONS

The MDOS utility functions must be called from within a machine code program running as a task under MDOS. You pass arguments to the utility functions using only a few registers of your program's workspace.

The MDOS utility functions are invoked from a machine code program when software trap number zero (XOP 0) is called with a library number of 9. The calling program's R0 must contain the opcode of the routine within the utility library which is to be performed. The following code fragment will return the day of the week to the calling task.

```
        LI      R0,7
        XOP     @NINE,0
        MOV     R1,@WEEKDA
* ...
WEEKDA  DATA    0          day of the week (1-7):(Sun-Sat)
* ...
NINE    DATA    9
* ...
```

VALIDATE TIME

Function	This operation is used to check the time stored in the clock chip for validity. It insures that the minutes and seconds are in the range 0:59, and insures that the hours are in the range 0:23.	
Parameters	R0	= 0 (opcode)
Results	EQ status	

Parameter description

EQ status	The equal status bit will be set if the time is valid, allowing you to perform a "JEQ time$ok" right after the software trap.

READ TIME

Function	This operation reads the time of day from the clock chip, and places it into your string buffer as a formatted string, with colons between the hours, minutes, and seconds.	
Parameters	R0	= 1 (opcode)
	R1	= buffer
Results	Buffer contains time string "HH:MM:SS".	

Parameter description

Buffer

The buffer address you pass for the string is a 16-bit address within your task's linear address space. The buffer must be ten characters long, and the address you pass is the address of the second character in the buffer.

On return, the first character of the buffer (offset 0) will contain a length byte. The next eight characters, starting at the address you specified, will contain the formatted time string. The last character in the buffer (offset 9) will contain a zero byte, for a null terminated string.

SET TIME

Function	This operation will set the clock chip using the time in the formatting string which the calling task passes as an argument.

Parameters	R0 = 2 (opcode)
	R1 = string

Results	EQ status

Parameter description

string — The address you pass for the string is a 16-bit address within your task's linear address space. The address you pass is the address of the second character in the buffer (the first text character in the string.)

The first character in the string buffer must be a length byte, giving the number of text characters in the string. Any leading spaces in the string will be ignored.

The text of the string must have the following format:

[h]h:[m]m[:[s][s]]

EQ status — The equal status bit will be set if the time string is valid, allowing you to perform a "JEQ time$ok" right after the software trap. The clock chip is not altered unless the EQ status has been returned.

VALIDATE DATE

Function	This operation is used to check the date stored in the clock chip for validity. It insures that the month is in the range 1:12, the day of the month is the range 1:MAX_DAYS[month], the year is in the range 0:99, and that the day of the week based on the month-day-year in the clock chip agrees with the day of the week stored in the clock chip itself.
Parameters	R0 = 3 (opcode)
Results	EQ status

Parameter description

EQ status	The equal status bit will be set if the date is valid, allowing you to perform a "JEQ date$ok" right after the software trap.

READ DATE

Function	This operation reads the date from the clock chip, and places it into your string buffer as a formatted string, with a dash between the month, day, and year.

Parameters	R0 = 4 (opcode)
	R1 = buffer

Results	Buffer contains date string "mm-dd-yy".

Parameter description

Buffer

The buffer address you pass for the string is a 16-bit address within your task's linear address space. The buffer must be ten characters long, and the address you pass is the address of the second character in the buffer.

On return, the first character of the buffer (offset 0) will contain a length byte. The next eight characters, starting at the address you specified, will contain the formatted date string "mm-dd-yy". The last character in the buffer (offset 9) will contain a zero byte, for a null terminated string.

SET DATE

Function	This operation will set the clock chip using the date in the formatting string which the calling task passes as an argument.	
Parameters	R0	= 5 (opcode)
	R1	= string
Results	EQ status	

Parameter description

string

The address you pass for the string is a 16-bit address within your task's linear address space. The address you pass is the address of the second character in the buffer (the first text character in the string.)

The first character in the string buffer must be a length byte, giving the number of text characters in the string. Any leading spaces in the string will be ignored.

The text of the string must have the following format:

[m]m/[d]d[/[y][y]] [m]m-[d]d[-[y][y]]

EQ status

The equal status bit will be set if the date string is valid, allowing you to perform a "JEQ date$ok" right after the software trap. The clock chip is not altered unless the EQ status has been returned.

JULIAN DATE

Function	This operation performs the function of a perpetual calendar, and will work on any date after January 1st, 1 AD.	
Parameters	R0	= 6 (opcode)
	R1	= month
	R2	= day
	R3	= year
Results	R1,R2	= julian date

Parameter description

Year	This must be the full year, like "1989", not "89", for the year in which this documentation was written.
Julian date	This is the number of days since January 1st, 4712 B.C.

DAY OF WEEK

Function	Returns the day of the week, from one (Sunday) to seven (Saturday).	
Parameters	R0	= 7 (opcode)
Results	R1	= weekday

Parameter description

Weekday	This is a sixteen bit integer with a value from >0001 (Sunday) to >0007 (Saturday).

PARSE FILENAME

Function This operation will convert a logical filename descriptor to a physical filename descriptor recognized by the Device Service Routines. For disk devices, the conversion may depend on the drive currently set for the task and the current subdirectory on the drive (depending on the ambiguity left in the name by the calling program.)

It is useful when you wish to make your application program independent of which device it was loaded from or when your application must ask a user for a filename.

Parameters	R0	= 8 (opcode)
	R1	= logical name
	R2	= physical name
	R3	= alias flag
Results	R0	= delimiter
	R1	= error code
	EQ status	

Parameter description

Logical name This is the address of the first character in the string to be converted to a physical device name.

At first, the name is compared to the names of all character devices recognized by MDOS. If the string matches the name of any of the character devices, the string will be copied without modification to the specified string output buffer.

There are three separators regonized by this routine as part of a disk pathname: COLON ":", PERIOD ".", and BACKSLASH "\". If the first separator found before a terminating delimiter is a PERIOD, the entire string will be copied without modification to the specified string output buffer.

The following characters indicate the end of the input string if they are not contained inside of double-quote marks: SPACE, COMMA, SLASH, SEMICOLON. The NIL character (>00) always terminates the input string, even if there are unmatched double quotes in the string. These characters are referred to as "terminal characters".

The terminal characters along with the three separator characters are known as "delimiters".

Remaining filenames are parsed as follows:

Part A, drive alias. All characters parsed during this phase are ignored in subsequent phases of the parsing.

caller R3	alias flag <>0	-> null string
"volume:"	+ non-terminal	-> "WDS.volume."
"volume:"	+ terminal character	-> "volume"
"n:"	+ terminal character	-> "alias"
"n:"	+ non-terminal, "\", or "."	-> "alias."
all others		-> "alias."

Part B, current directory. Using the characters remaining in the input string after Part A.

"\"	is first character	-> null string
current dir	is NULL	-> null string
"."	+ terminator	-> "CURDIR"
".."	+ terminator	-> "PARENTDIR"
".\"	+ non-delimiters	-> "CURDIR."
"..\\"	+ non-delimiters	-> "PARENTDIR."
all others		-> "CURDIR."

Part C, file specifier. This is the characters remaining in the string after Part A and Part B have been done. These characters are copied into the output buffer until a BACKSLASH, QUOTE, or terminator is found. When a BACKSLASH is found, it is replaced by a PERIOD. When a terminator is found, parsing of the input string is stopped, and the address of the terminator is returned to the caller. When a QUOTE is found, all characters until the next QUOTE or NIL in the input string are copied to the output buffer (Unless two matching QUOTES were adjacent to each other, in which case a single QUOTE character will be placed into the output buffer.)

The resulting string in the output buffer is "PART A" + "PART B" + "PART C", and is returned to the caller.

Phys. name	This is most useful when it specifies the name length byte in a PAB. Note that the physical name can use the same buffer as the logical name, and will simply overwrite the logical name after parsing is complete. As a caller, you must specify the address of the length byte in your string output buffer with this parameter. Before calling the parse routine, you must set the length byte to the maximum length allowed for the output string, which is returned in the form "<len><chars><nil>".
Alias flag	This flag must be set to zero for normal processing. If this flag is non-zero, no disk drive alias will be prepended to the output filename. (This feature is used only by the CHDIR command of MDOS at present.)
Delimiter	This is the address of the first character in your input string which wasn't processed during the generation of the filename. Note that this is designed in such a fashion that you generally don't need your own routine to parse filenames which are passed to your program as command line parameters; just call the parse routine, check the delimiter, and call the parse routine again for the next command line parameter.
Error code	This is set to zero if no errors were encountered while parsing the filename. This is non-zero under several conditions: the resulting output string was too long for your buffer, a COLON is the first character in the input string, a drive specified with "n:" does not have an assigned alias, or a directory specifier of the form "." or ".." was not followed by a BACKSLASH or a terminator.

LOAD TASK

Function This operation will load a chained program image file into memory, and cause it to execute as a task under MDOS. Invocation of the new task will start at address >0400 with a workspace of >F000, and memory windows 0..6 of the task will initially contain the data loaded from the program image.

Parameters	R0	= 9 (opcode)
	R1	= physical name
Results	R0	= error code
	R1	= child page zero (physical page number)
	>00E8	in child task contains the physical page number of the parent's page zero

Parameter description

Phys. name This is the address of the length byte of a filename stored in the format "<len><chars>".

Error code

0	= no error, task was loaded
1	= insufficient memory
2	= invalid filename
3	= image file found, with invalid header

Image file header

An image file header has the following format, compatible with GenLINK.

byte 0	if >00, last image in chain, otherwise bump filename and load another image in the chain
byte 1	"G" (normal speed) or "F" (use fast memory)
byte 2,3	length of this image file
byte 4,5	load address of this image file
byte 6..len+6	image data bytes

FORK TASK

Function This operation causes the creation of a new task under MDOS. The new task (child task) is an exact copy of the calling task (parent task).

The new task has a virtual memory address space which has one physical memory page for each physical page used by the parent task. If the parent task was using shared memory, the child task can also use the same shared memory and communicate with the parent task.

Further note: The terms "parent" and "child" are used as a convenience in differentiating between the calling task and the newly created task. In MDOS itself, there is no concept of "task tree" or parent-child relationship as there is in some other operating systems. In MDOS, all tasks are peers.

Parameters R0 = 10 (opcode)

Results parent task:

R0	= -1 (error) = otherwise, this is the physical page number of the child task's header page.
PC	= program execution continues with the instruction after the XOP call. This instruction must be a single word instruction such as a Jump instruction.

child task:

R0	= -1
PC	= program execution continues with the 2nd instruction after the XOP call. Note that the instruction after the XOP call must be a single word instruction.
>00E8	in child task contains the physical page number of the parent's page zero

Parameter description

error	An error code of -1 will be returned to the parent task if there is not enough memory available on your system to create a clone of your task.

Example code

```
                LI          R0,10
                XOP         @NINE,0
                JMP         PARENT
CHILD           PRINT       "This is the child speaking..."
                BLWP        @0            exit
PARENT          PRINT       "Child: be quiet!"
                BLWP        @0
```